Accelerating CABAC Encoding for Multi-standard Media with Configurability

Oskar Flordal†, Di Wu, and Dake Liu
Department of Electrical Engineering, Linköping University
SE-581 83, Linköping, Sweden
oskar@flordal.net,{diwu,dake}@isy.liu.se

Abstract

This paper presents the study of how to accelerate CABAC encoding for emerging heterogeneous multimedia applications. The latest image and video compression standards such as JPEG2000 and H.264 both have adopted Context Adaptive Binary Arithmetic Coding to achieve performance enhancement. However, CABAC requires high computing power. After investigating computational complexity of CABAC coding, firstly, instruction level acceleration is elaborated. Secondly, a configurable accelerator for CABAC encoding in multiple standards is proposed. Benchmarking performance and implementation cost is also addressed.

1. Introduction

Entropy coding such as Variable Length Coding (VLC) and Arithmetic coding (AC) has been widely adopted in modern information compression systems in order to achieve a higher compression ratio. Context-based Adaptive Binary Arithmetic Coding (CABAC) has been adopted in latest multimedia coding standards such as JPEG2000 and H.264. Compared to VLC, arithmetic coding has its advantage in compression ratio while it introduces higher complexity.

JPEG2000: JPEG2000 is the latest still image compression standard which brings significant improvement in compression performance compared to the JPEG standard. Motion JPEG000 is a multiple frames (e.g. 30fps) extension of JPEG2000 which works almost in the same way as standard JPEG2000 in the sense that each frame is coded independently without motion estimation. This technique is useful for applications with high demand for quality such as digital cinema (e.g. with resolution of 4096x3112) and medical imaging. Motion JPEG2000 generates a large amount of data for these applications which requires the encoder to sustain real-time coding performance at very high bitrate.

H.264: H.264 (MPEG-4 part 10) is the latest video compression standard. By adopting innovative algorithms, both high compression rate and error robustness is achieved which is critical for IP based networks. Compared to MPEG-2 and MPEG-4, H.264 greatly promotes coding efficiency while it introduces much higher computational complexity. For high-end applications, a powerful hardware platform is required to provide real-time performance.

2 Algorithm Features and Comparison

2.1 Arithmetic Coding

Arithmetic coding is one of the entropy coding methods which converts a sequence of data symbols into a single fractional number. By dividing a range, for example between 0 and 1, iteratively depending on the probabilities of different symbols, arithmetic coding interprets a given set of symbols to a value in the range that can only be coded by that specific sequence of symbols.

For example if at the beginning the range is 0 to 1 and there is four symbols (for example A,B, C and D in Figure 1) to encode with the probabilities 1/2, 1/6, 1/6 and 1/6 respectively. Whenever a symbol is to be encoded, the current range will be divided according to the size of the probabilities and the new range will be the interval that represents the symbol that is coded, for example 0-0.5 if the symbol A, with probability 1/2, was coded. This new range becomes the basis for the next symbol to be encoded as depicted in Figure 1 and in there will be a range that is unique for the coded symbols and will be coded with bits. The amount of bits required to code the smallest value in the final range is then usually smaller than the amount of bits required to code the original message. Compared to other entropy coding methods such as Huffman coding, arithmetic coding almost reaches the optimal presentation for coded symbols in

†Oskar Flordal is now with Axis Communications AB, Lund, Sweden.

1-4244-0054-6/06/$20.00

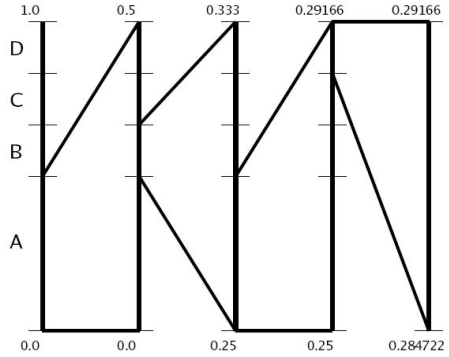

Figure 1: Explanation of Arithmetic Coding

number of bits, which means it is more efficient. For example, compared to variable length coding, binary arithmetic coding provides bit rate reductions of $5-15\%$ in H.264.

2.2 CABAC in Multi-standard

2.2.1 Binary Arithmetic Coding

Binary Arithmetic Coding (BAC) where only two symbols are used, for example 1 and 0, is in many cases the most convenient and efficient way to do arithmetic coding. In principle the encoder parts of the CABACs in JPEG2000 and H.264 both employ the binary arithmetic coding method for coding. However, several practical differences exist in the implementations. As a binary encoder, bits are encoded and shifted one at a time. The symbols that are coded are referred to as Most Probable Symbol (MPS) and Least Probable Symbol (LPS) and whether MPS is 0 or 1 is determined by the context. In both implementations the convention of a low value that is the base of the current range is used (referred to as C in the JPEG2000 standard[1]) and a range value that is the length of the coding range (called A in JPEG2000). The division of these according to LPS and MPS is carried out in different ways. In H.264, LPS is coded in the higher values of the range (Range - LPS, probability would be added to Low)[2]. While in JPEG2000, the LPS field is in the lower part of the range except when the LPS range is actually larger than the MPS field in which cases the fields are reversed.

2.2.2 Context Modeling

Besides binary arithmetic coding, a large amount of work is done in other steps of CABAC which includes binarization and context modeling. The context modeling selects the correct context for bits which have been decided by the binarization and is perhaps the heaviest task in the CABAC.

JPEG2000: In JPEG2000, context modeling is conducted as follows: A bit in a bitplane is mapped to the bits in the same coefficient and bits closest in the bitplane. A bit can be encoded in three different passes (**Significance Propagation**, **Magnitude Refinement** and **Cleanup**) depending on surrounding bits and how bits were coded in previous bitplanes for that coefficient. Based on neighboring states, the most suitable probability model is chosen to encode the symbol.

H.264: There are a number of different ways to build contexts in H.264 and they can be divided into four main categories. The first type gets its context from previously coded blocks above and to the left, which is suitable for syntax such as motion vectors where blocks nearby have a tendency to move the same way. The second type is a model which is used to code macroblock information and build its context based on previously coded bits in this state. Both the third and fourth category can only be applied to residual data. The fourth category is like the third only used for residual data and the contexts are based on accumulated encoded values from earlier encodings, while the third one does not use past data. Some data is also encoded without a particular model. [2]

The different context indexes describe the way to encode information such as motion vectors (there are different context depending on the length of the vectors and so on) and coefficients. There are in total 399 different context values and these in turn point to 64 different (6 bit) probability states with a current MPS (another bit). More details of CABAC can be referred to [1],[2] and [5].

2.3 Complexity Analysis

The bit rate of coded H.264 video stream with HDTV quality (1080p/30fps) is approximately 10 Mbps. The demands of other resolutions can be found in Table 1 as reference. The GNU profiler (gprof) has been used as a tracer to perform profiling. The number of invocations of each subroutine is traced and recorded in the log file which provides enough precision to expose performance bottlenecks and help make decisions on SW/HW partition. However, reference codes for both standards have been rewritten in order to be optimized and mapped to a single issue DSP platform. Thus the final estimation of MIPS cost is reasonable and accurate enough based on this DSP platform.

Resolution	QCIF (176x144)	CIF (352x288)	4CIF (704x576)	HDTV (1920x1080)
Bitrate	0.5 Mbps	1 Mbps	3 Mbps	10 Mbps

Table 1: Bitrate for Various Quality

JPEG2000: In order to perform profiling of JPEG2000, a reference toolkit called JasPer[7] is used which includes a software implementation of the JPEG2000 CODEC. According to profiling done on this software implementation,

CABAC coding is around 60% of the total MIPS cost. The number of bits that are processed in the CABAC based on profiling of various test images are depicted in Table 2. The invocation number (InvoNum) tells how many times the arithmetic coder is called during the encoding of one frame of the image at a specified size multiplied by 30. Mbps states how many bits per second would go through the encoder if the image was used in a video sequence with 30fps (frames per second). To achieve an approximate MIPS cost, Mbps has then been multiplied by an approximate value of cycles required to code a bit on average according to our DSP platform. The test images (Boat and Wharf) are from the CD that comes with [6] and have been scaled to different resolutions.

Test Sequence	Resolution	InvoNum	Mbps	MIPS
Boat (30fps)	QCIF	12538200	13	260
	CIF	47528790	48	960
	4CIF	185618130	186	3720
	HDTV	929133330	929	18580
Wharf (30fps)	QCIF	11956890	12	240
	CIF	45671940	46	920
	4CIF	170628720	171	3420
	HDTV	821745540	822	16440

Table 2: Profiling of JPEG2000

H.264: For H.264, reference software from JVT which is called JM[8] has been used to perform profiling. The different clips in Table 3 are taken from a set of standard 1080p clips and standard clips in smaller sizes. The invocation numbers(InvoNum) in this case is calculated over 50-250 frames as the frames are dependent on each other in H.264. The Mbps is based on frame rate of 30fps.

Test Sequence		Paris	Highway	Pedestrian
Resolution		352x288	352x288	1920x1080
Frames		250	250	50
InvoNum	Context	1914198	1208284	6790618
	MQ-coder	5551384	3129586	15618854
Mbps (30fps)		0.67	0.38	9.37
MIPS (30fps)		20	12	271

Table 3: Profiling of H.264

3 Hardware Acceleration

3.1 Design Consideration

From complexity analysis elaborated in Section 2.3, it is obvious that pure SW implementation of CABAC is not efficient enough and therefor hardware acceleration is required. Generally, application based instruction level acceleration (ILA) and dedicated hardware acceleration (DHA) can both promote performance to different degrees based on different hardware cost.

3.2 Instruction Level Acceleration

ILA is about finding ways to add instructions and architectural features in order to accelerate specific tasks. Based on the study of CABAC in multiple standards, a set of instruction extension which accelerates the processing of CABAC on general DSP has been proposed. ILA is tightly defined to the processor core and generally can be applied to a larger set of tasks with similar computational features. Compared to DHA, it has lower silicon cost and higher flexibility while only limited performance enhancement can be achieved. For mobile applications, only relatively low resolution (e.g. 352×288) is considered. Thus the bit rate is relatively low. For the sake of silicon cost, power consumption and programmability, to apply ILA to a general DSP is preferred.

3.2.1 ILA Design

It has been disclosed that CABAC coders in different standards are very branch heavy while containing relatively few operations in each branch. In order to accelerate the coding process, one set of improvement that has been suggested is to use features like delayed execution, address register arithmetic and, most interesting, if-then-else decisions. The first two are already common architectural features and need not be discussed further. If-then-else instructions or conditional execution is interesting in the cases where branches are evenly matched and could be considered in some cases.

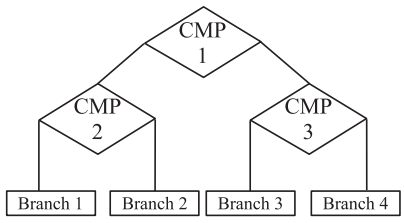

Figure 2: Branch Tree

A multi-branch instruction has been proposed to accelerate control flow by calculating multiple branching steps in one cycle. A reasonable limit would be three comparisons in parallel and to find the correct branch in a binary tree as is depicted in Figure 2. The instruction compares a few pre-loaded constants in flexible combinations with at most two other registers (which is the maximum most bus structures would allow). If this multibranch instruction is used on the LPS/MPS branch in JPEG2000 that consists of three possible compares, they could be executed together in one cycle with a little rewrite. Similarly the same function could efficiently deal with the H.264 renorm (renormalization) process and the encoding process, but it would not be

as efficient on those parts. The problem with the instruction is that it would need some pre-loading/configuration and is not fully orthogonal to most instruction sets. But with quite simple hardware, three comparators and a few muxes and short registers it would be useful. The initialization of these four registers takes four cycles only once and will be updated in runtime until they are reconfigured for the next frame. Another optimization is to incorporate some sort of jump back functionality that works with the Program Counter. At a few certain values the Program Counter would unconditionally be set to a common value that is the location where the execution is supposed to continue after the branches. This way the jump instruction that is needed on most branches could be skipped and the branching would in principle be free if the first branch could be done in the background.

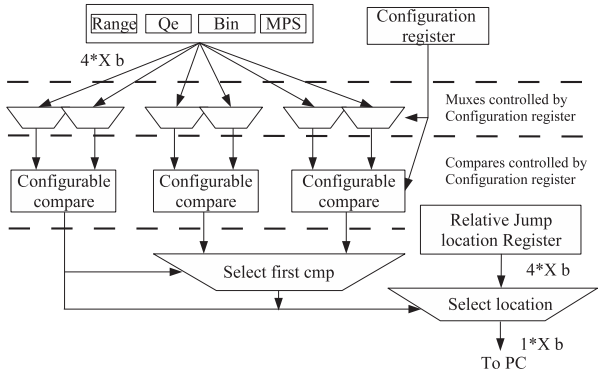

Figure 3: Multi-branch HW

3.2.2 HW Implementation

As depicted in Figure 3, four operands are fetched from four registers, these could either be all hardwired or a mix of hardwired registers and bus registers. To select which operands to use for the three compares there are two muxes for every compare. The first compare selects which of the other two compares should be used by controlling a mux to where both the other compares are connected. The bit coming from the first compare together with the bit that got through the mux decides which location the PC should go to. These four locations are stored in four 8-bit registers that contain relatives addresses. The standard logic around the PC will make the addition necessary for it to be a relative jump. The cost of this ILA lies mainly in registers and in the case that all values need to compared dynamically, registers have to use an additional bus so the values could be loaded without any overhead. The jump destinations of this branch also need to be stored, which requires one more register. Generally, 8-bits relative addresses is enough to provide flexibility with low HW cost.

3.2.3 Sample Code

A version of the JPEG2000 encoder without renorm is depicted in Figure 4. Note that this changes the program flow as depicted in the standard to suit the multi-branch better by moving the A (recursive probability interval) and 0x8000 comparison to the branch where $A \geq Qe$ (estimated probability of LPS). Both LPS branches can obviously be shortened if 32 bits could be loaded from memory to two separate registers or one 32 bit register with high and low part. Otherwise even with a fast memory, six to eight cycles are required for an encoding process. The example code requires that the multi-branch has been configured already by writing the destinations of mpsa, mpsb, lpsa and lpsb to four special purpose registers. Together with this each compare unit has to be configured on what type of compare ($<, >$ or $=$ and so on) it will do between its two operands. Furthermore it expects hardwired registers which are pointed to by Qe, Range, MPS and (outside this code snippet) Bin that describe which bit is being coded.

```
start:
    loaddm state,qe    ;load qe first in dm
    sub range,qe,range ;range -qe
    multbr      ;multibranch: we now end up in
                ;mpsa,mpsb,lpsa or lpsb
                ;depending on the comparissons
                ;of range,qe,bin and mps
    nop;        ;avoid delayed execution
                ;it would be ok to do the next
                ;instruction instead, but
                ;added for clarity

mpsa:           ;range<qe and bin==mps
    loaddmr state,mpsoffset,state
    addi 0,qe, range
mpsb:           ;range>=qe and bin==mps
    cmp range,quarter
    brge done   ;Do not do renorm
    add low,qe,low
    loaddmr grp15,mpsoffset,state

lpsa:           ;range<qe and bin!=mps
    add low,qe, low
    loaddmr state,lpsoffset,state
    loaddmr state,switchoffset,grp16
    xor mps,grp16,mps

lpsb:           ;range>=qe and bin!=mps
    addi 0,qe, range
    loaddmr state,lpsoffset,state
    loaddmr state,switchoffset,grp16
    xor mps,grp16,mps
    ;this could be loaded instead making
    ;this instruction redundant with
    ;a little extra memory
renorm:
```

Figure 4: Sample Code of ILA

3.2.4 Benchmarking Result

It is hard to evaluate the performance of various ILA methods in a fair way, without actually implementing them in a real processor and analyzing memory latencies and setup

time to get accelerators running and so on. In order to benchmark proposed instructions, a simulator of a single issue DSP designed by our division was used and modified to accommodate ILA. It is assumed in every loop that registers like state, MPS and so on are loaded previously which takes cycles and could be argued to be the job of the encoder. As the DSP has memory fetches and writes in one cycle, the only real penalty is pipeline flushes due to missed branch prediction and data dependencies in the pipeline. In our implementation it cuts down the cycle count almost 50% in the encoder before renorm and also automatically makes it less prone to branch misses compared to the case with multiple branches. The benchmarking was done by combining assembly based subroutine implementation and statistics of subroutine invocation frequency collected from the reference software. This gives us a quite good estimation of MIPS cost. For H.264, when using a specialised instruction for picking out 2 bits from the range and a multibranch for both encoder and renorm, four cycles can be saved in the encoder and another four from the renorm part which reduces the MIPS cost by 27%.

Resolution	QCIF (176x144)	CIF (352x288)	4CIF (704x576)	HDTV (1920x1080)
MIPS Cost	10	21	63	210

Table 4: ILA Benchmarking Result

As shown in Table 4, MIPS cost of ILA implementation at various resolutions has been estimated. For low-end applications, such as mobile video, ILA ensures real-time performance with flexibility and low design cost. While for high-end applications such as HDTV, the MIPS cost is still too high which means Dedicated Hardware Acceleration (DHA) is required.

3.3 Dedicated Hardware Acceleration

For high-end applications such as HDTV and camcorders, bit rate may be higher than 10Mbps. In order to achieve real-time coding performance of CABAC, DHA is required. Generally an accelerator is designed to fulfill CABAC coding without frequently interrupting the DSP processor. Several dedicated CABAC accelerators have been implemented specifically for JPEG2000[3] and for H.264[4] compression systems. A novel architecture with Inverse multiple branch selection (IMBS) method was proposed in the design of DHA for JPEG2000 by Pastuszak[3]. In our design, based on the study of characteristics of CABAC in multiple standards, this architecture was extended for both JPEG2000 and H.264. Thus when achieving performance improvement, hardware reuse was also achieved.

As shown in Figure 5, the CABAC coding process consists of the following four stages.

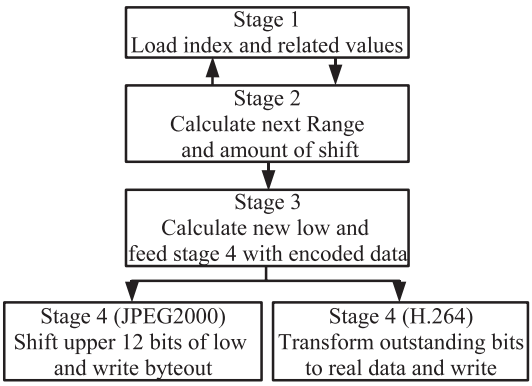

Figure 5: Stages of DHA

Stage1(Pre-load): The first stage prepares index values and symbols for the calculation in stage 2. As Qe values are selected differently there will be a few changes here. Most notably is the memories that contain Qe values as they have to have more rows and be longer due to having four Qe values per row. The Qe memory has to be around 256 bytes in H.264 as compared to roughly 100 bytes in JPEG2000. The largest problem here is to localize the storage of the context table. That is as in JPEG2000 only 20 contexts are used which makes it quite easy to put them all in a flexible register file, however in H.264, since 399 context values are to be stored, it is impractical to allocate these 399 values into one register file.

As two values need to be both read and written in one cycle one memory is not enough. Even if the memory would be dual port, there would still only be time for two of four jobs to be finished every cycle. If dual memories are used they would also need to be synchronized so it is not as trivial as to just add another memory.

As depicted in Figure 6, the solution proposed by us is to use a cached architecture where the most used context values would be kept in the cache. Statistics acquired by running cache based solutions resulted in an estimation that a 64 post fully associative cache gets around 15% cache misses. Such a cache will however be as large as the original register file with surrounding logic so another solution is necessary. A cache with eight rows containing four posts each for simple memory transfer achieves, without optimising the memory, 65% hits. Doubling the size of the cache results in 22% misses which is closer to the fully associative as it is the same size, and a quite good compromise. An interesting feature when using a cache is that the structure could easily be reused for JPEG2000 were all contexts would fit in the cache. This would eliminate the need for using a register file or similar, especially for JPEG2000.

Stage2(Update Range): Main function of this stage is

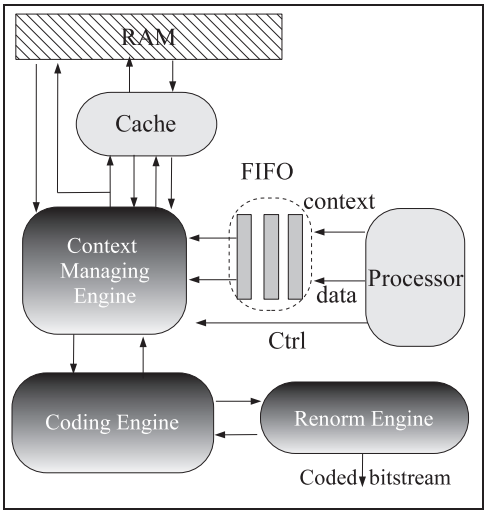

Figure 6: Accelerator Architecture

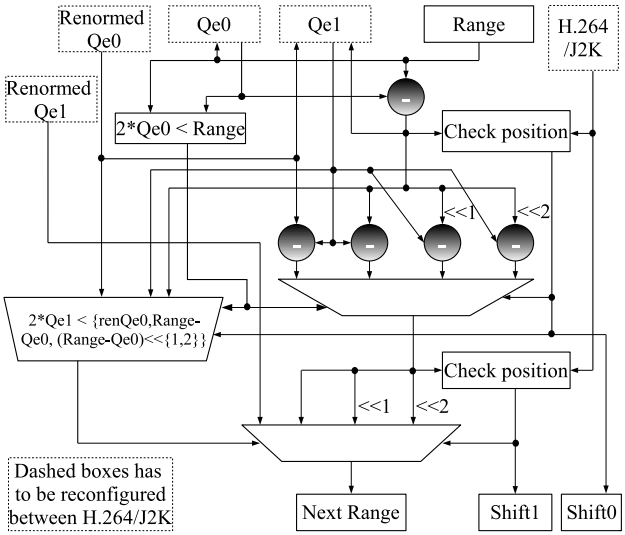

Figure 7: DHA Stage 2:Update Range

to update range based on the coding process. According to IMBS (inverse multiple branch selection) proposed in [3], parallelism can be achieved if the coder pre-calculates all possible outcomes of the first range when updating the second bit and then select the result that was correct based on the outcome of the first bit. To avoid a special renorm of the data the fact that the value after the first encoding is either a renormed Qe0 or $Range - Qe0$ in both CODECs can be utilized. If the encoding results in $Range - Qe0$ the renorm will shift the data either zero, one or two times otherwise a pre-calculated renormed Qe0 will be used. It is known that there are at most two shifts as Qe0 is at most less than 3/4 the size of range in both CODECs. If a branch is made for each of these shifts and the renormed Qe all the possible new range values have been covered. Which one to use is selected by figuring out if what is coded is an MPS symbol, if $Qe0 < Range - Qe0$ and by finding out how many times to shift. Finding out how many times to shift is easily done when there are so few possibilities and separate logic can be used to do it for H.264 and JPEG 2000. The same technique is used to pre-calculate $2 \times Qe < Range$ for the second bit.

The extra precision granted to H.264 by having more lookup values must also be taken care of in this step as this is the first opportunity to know the second range value. Having to select this value here is far from optimal as it stalls some of the branches until correct selector bits can be calculated from the first range, nullifying most of the advantage gained from IMBS.

As depicted in Figure 7, the thing that has to be configurable between the different CODECs in this stage is primarily which bits to look at to find out how many shifts should be done as well as the device for getting the extra precision in H.264. Another thing is that the switching mechanism is disabled in H.264 as an interval swap is never done in H.264. These are all minor changes and this stage in general makes for great hardware reuse.

Stage3(Update Low): As it is shown in Figure 8, this stage updates the lower boundary of the coding range. To cope with the slightly different selection where H.264 can set Low to $Low = Low + Range - Qe$ and JPEG2000 besides keeping the old Low value can have $Low = Low + Qe$, a mux has to be added to both those places. The addition can be done using the same adder with a mux or done for clarity as in Figure 8 with one adder for both possible additions. The resulting value is then shifted the same amount of times the range value was shifted in the previous stage. The big difference between the CODECs comes after the shift as H.264 only uses at most 10 bits of Low and in case every previous shift in the renorm has not produced a '1' it is only 9 bits. These three steps are done twice, once for each bit.

To get the proper bits for stage 4 in H.264 a device to merge the correct bits from the renorm has been devised. It shifts the bits from the first coding to the left of the bits from the second coding to turn them into the same stream. As the renorm in H.264 can code either bit 8 or bit 9 in the first place and so on there has to be muxes that decide which bit that should actually be used for every position. To understand this consider an example with the bits $1_9 1_8 0_7 1_6 1_4 0_3 0_2 0_1 0_0$ and suppose 5 bits should be shifted. Bit 7 will in in this case be ignored as when the bits have been shifted twice the front most bits will be 01. This indicates that an outstanding bit should be coded and then the second of the bits will be removed. After the second bit has been removed there is no longer a chance for a 1 to end up in the ninth bit place (as possible new ones will be eliminated at the second position) and from now on the only possible

outcomes are 0 or bit outstanding for the rest of the shifts.

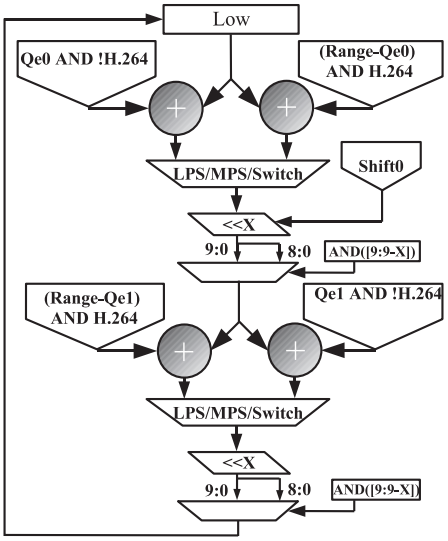

Figure 8: DHA Stage 3:Updating Low

Stage4(Renorm): Renorm is sequential in nature and would be hard to implement as a combinatorial net. The simple solution is to implement it as a sequential net that updates the Range and Low by shifting a bit at a time. But as a parallel coder will generate bits at a high speed it has to be done in once cycle to avoid constant pipeline stalls. As the renorms are quite different, a separate solution is needed for the different standards.

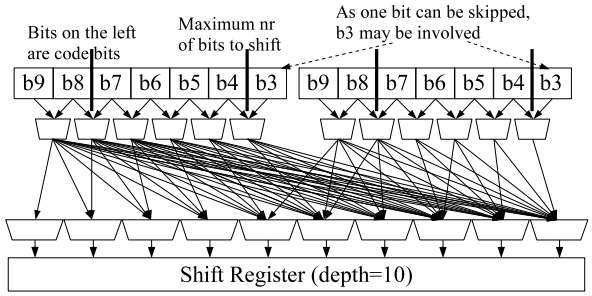

Figure 9: Output Generator of Renorm Stage (H.264)

As depicted in Figure 9, bits loaded from both low values are shifted into a register that is large enough to contain the maximum amount of bits that can be added in one cycle. This should be approximately 25 from previous bits outstanding plus five more as the maximum of renorm cycles is six in once cycle plus an additional four from the other renorm as a maximum when the previous cycle coded a renorm which in total makes 34. An additional buffer is also required for wire speed. The register now contains as many bits as will be put out, but a bunch of them will contain values based on outstanding bits which needs to be corrected. The way the bits have been packed in the previous stage there is now a register with coded zeroes and ones and another register which marks which bits are yet undecided. By using a chain as described in Figure 9 all outstanding bits that have a known bit to the right (which means they are coded later) of them can now be corrected.

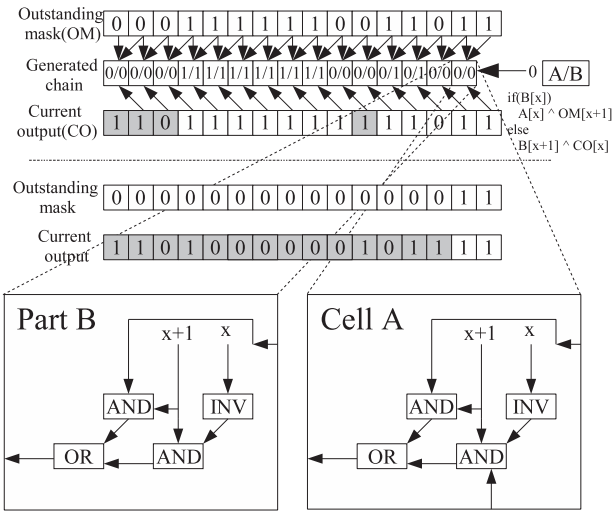

Figure 10: DHA Stage 4: Renorm

The solution prepared to solve this is actually not optimal in its simplest form as it contains a carry chain of 50 or so gates (if the maximum propagation chain would cover 25 bits) but there are ways around this. Values at the end of a sequence of outstanding bits are propagated along the propagation chain shown in Figure 10. The first bit in a series of outstanding bits gets the value of the bit to the right on the end of the outstanding bits, that is the first value that is known which is normally coded first. The other bits including the far right one at the same time gets the inverted value of the leftmost bit that was just coded. This way the renorm can be done in one step, which makes things more efficient and also simpler in a way.

3.3.1 Configuration

Selecting which CODEC to code for is done by setting a bit that indicates if it is H.264/JPEG2000 while coding and by feeding a proper value to the Range register when resetting the accelerator. This could for example be done through a special purpose register.

4 Benchmarking and Prototyping

The performance of this implementation is determined by the amount of stalls that have to occur. A stall will have to occur when both bits to be coded are supposed to

be coded in the same context, as the context need to be updated for the next cycle in that case. This seems to occur 17% of the dual encodings in H.264 and 11% in JPEG2000 which suggests well over 1.5 bits will be coded per cycle if the pipeline is kept full. As argued in [3], this of course comes at the cost of potentially lower frequency, especially when using a single cycle renorm as suggested here. The other problem is hardware cost. In H.264 the pattern of ordinary encodings is interrupted by equal probability codings, which brings the problem that the codings works slightly different and the output has to be stalled and flushed if this machinery is not added to the last two steps (it only affects Low and of course output). Benchmarking is based on statistics of how often a stall occurs by counting the amount of time both contexts is same during the encoding process. The result was 1.82 bits per cycle, to reach those speeds the large register file discussed before would have to be used. If the 64 entry cache were used with unoptimised memory, 25% miss rate will be achieved using policy where the first context is always checked in memory at the same time it is checked in the cache, thereby making sure at least one context is available. If dual port memory is used, number of bits coded per cycle would be $1.615 = 0.82 \times (1 - 0.25) + 1$ bits(0.82 is the probability in case that $contex0! = context1$), which makes the estimation of more than 1.5 bits per cycle rather safe. Note that the second context will still end up in the cache the next cycle when it is instead considered the first context. To achieve these values, an equal probability encoder need to be built. According to the benchmarking scheme elaborated above, estimation of MIPS cost based on video sequences with different resolution is given in Table 5.

The DHA model was first implemented in Verilog and simulated in Modelsim. Then it was prototyped and tested on an Avnet Virtex-II FPGA board, by testing with various corner cases, the functionality has been proven correct. Finally, the design was synthesized with UMC $0.18\mu m$ process. The DHA (including a cache with 64 entries) consumes $0.039mm^2$ silicon area and reaches 190MHz as working frequency.

Resolution	QCIF (176x144)	CIF (352x288)	4CIF (704x576)	HDTV (1920x1080)
MIPS Cost	0.3	0.6	1.8	6

Table 5: DHA Benchmarking Result

5 Conclusion

In this paper, both ILA and DHA for CABAC has been investigated. Configurable solutions have been proposed for multi-standard media processing. Benchmarking has been conducted and comparison of different solutions have been made. ILA has been proven to be practical for low-end applications while DHA achieves enough performance for high-end applications with silicon efficiency. HW reusability has also been explored and it has been proven that the configurable accelerator proposed by us can accommodate high-end CABAC coding for the latest heterogeneous video coding standards with both flexibility and silicon efficiency.

6 Future Work

Context-based Adaptive Variable Length Coding can also utilize hardware components in our CABAC accelerator. Design of an entropy coding accelerator in charge of CAVLC, CABAC and general Huffman coding is ongoing as future work.

References

[1] JBIG and JPEG, *JPEG2000 Part I Final Committee Draft Version 1.0*, ISO/IEC JTC1/SC29 WG1(ITU-T SG8), March 2000.

[2] Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, *JVT-G050,Draft ITU-T Recommendation and Final Draft International Standard of Joint Video Specification (ITU-T Rec. H.264 ISO/IEC 14496-10 AVC)*, March 2003.

[3] Pastuszak, G, *A novel architecture of arithmetic coder in JPEG2000 based on parallel symbol encoding*, International Conference on Parallel Computing in Electrical Engineering, 2004. 7-10 Sept. 2004 Page(s):303 - 308

[4] Roberto R. Osorio and Javier D.Bruguera, *Arithmetic Coding Architecture for H.264/AVC CABAC Compression System*, Euromicro Symposium on Digital System Design, 2004.

[5] Marpe, D.; Schwarz, H.; Wiegand, T, *Context-based adaptive binary arithmetic coding in the H.264/AVC video compression standard*, Circuits and Systems for Video Technology, IEEE Transactions on Volume 13, Issue 7, July 2003 Page(s):620 - 636

[6] David S.; Taubman and Michael W. Marcellin. *JPEG 2000: Image Compression Fundamentals, Standards and Practice*, Kluwer Academic Publishers, Norwell, MA, USA, 2001. ISBN 0-79237519-X.

[7] Michael Adams. *The JasPer Project Home Page* (http://www.ece.uvic.ca/ mdadams/jasper/)

[8] Karsten Shring. *H.264/AVC Software Coordination* (http://iphome.hhi.de/suehring/tml/)