IMPROVED THROUGHPUT ARITHMETIC CODER FOR JPEG2000

Michael Dyer, David Taubman, Saeid Nooshabadi

The University of New South Wales, Sydney, Australia

ABSTRACT

Increasing the throughput of the JPEG2000 block coder requires bit-plane and arithmetic coders capable of concurrent symbol processing. Previously described pipelined MQ coders are capable of consuming 1 symbol or less per clock cycle. We develop a new pipelined MQ coder that can process exactly two symbols per clock cycle. The technique is implemented on a FPGA, and is compared with our "Hypothesis Testing" arithmetic coder and a reference one symbol per cycle coder. Our implementation gives an increase in throughput of 1.9 times, at the cost of 1.7 times as much hardware, when compared to the reference coder. It also has 1.2 times the throughput, while consuming only 70% of the hardware associated with the Hypothesis Testing coder.

1. INTRODUCTION

The block coding engine of a JPEG2000 encoder typically consists of a bit-plane coder that produces context-data (CXD) pairs. These are then entropy coded by an arithmetic coder. Current implementations are capable of processing at least one bit per clock cycle. Interest in creating bit-plane coders capable of processing more than one bit per clock cycle has been limited by the lack of arithmetic coders able to encode more than one CXD pair per cycle [1].

To improve the throughput of an individual block coder would require an arithmetic coder capable of consuming more than one CXD pair per clock cycle. Previous authors [2] investigated loop unrolling in the Q and QM coders to code multiple CXD pairs. "Hypothesis Testing" [3] is a generalization of this idea to code multiple CXD pairs from differing contexts. When we implemented a pipelined version of this technique, throughput increased to around 1.7 CXD pairs per clock cycle. This method has the drawback of a variable symbol rate, which complicates the interface. It also requires 2.3 times more hardware resources than a single CXD per cycle arithmetic coder.

In this paper we examine another technique for improving the throughput of the arithmetic coder. This proposed coder will always consume 2 CXD pairs per cycle. The proposed design is implemented on an Altera 20KE600 FPGA, and is compared with our previous Hypothesis Testing coder, as well as a high throughput reference single CXD per cycle coder [4].

2. ARITHMETIC CODING

JPEG2000 uses the MQ variant of arithmetic coder. This is based on the QM coder used by JBIG [5], but uses the byte emission technique of the Q coder [6]. Both coders are multiplier free; all arithmetic is implemented using adders (or subtractors).

Like its predecessors, the MQ coder is context based. Each symbol has an associated context that determines its probability estimate. There are 18 contexts specified in JPEG2000. For the purposes of symbol probability estimation, each context has a state and the value of the Most Probable Symbol (MPS) associated with it. The state holds an index into a probability estimation table (PET), whose entries identify the probability of the Least Probable Symbol (LPS), amongst other things. There are 46 possible states.

The MQ Coder can be separated into several fundamental operations. The first is context state update control and probability estimation. Given a context, this operation will produce the probability of the symbol being an LPS. This information is then used by a series of arithmetic operations that update two register values, A and C. A represents the length and C the lower bound of an interval in $[0, 1)$. A is updated according to

$$A = \begin{cases} A - \overline{p} & \text{if MPS} \\ \overline{p} & \text{if LPS} \end{cases} \quad (1)$$

where $\overline{p}$ is the LPS probability estimate. C is updated according to

$$C = \begin{cases} C + \overline{p} & \text{if MPS} \\ C & \text{if LPS} \end{cases}$$

It should be noted that these are simplified equations that do not show conditional exchange, which may switch the roles of the MPS and LPS. The proposed implementation in Section 6 uses conditional exchange, which prevents the A register falling below half its value when coding an MPS.

Because of the finite precision of the A and C registers, if the value of A falls below $0x8000$, it is left shifted until $A \geq 0x8000$. C is left shifted by the same amount. This is termed a renormalisation event. Each register has a 16 bit resolution. The bits that are left shifted out of the C register form the codeword, which is output in byte-oriented fashion. During renormalisation, the state information for the current context is updated. Each state has two possible next states, the choice depends on whether the renormalisation was caused by an LPS or an MPS.

3. MULTIPLE SYMBOL ARITHMETIC CODING

Arithmetic coding is usually seen as a sequential operation. However, it is possible to despatch multiple symbols per clock cycle. We identify at least two methods for doing this.

In updating A according to equation 1, if the result is less than $0x8000$, renormalisation is required to bring it to a value $\geq 0x8000$. However, if renormalisation did not occur, we could continue to operate on A as a cascade of arithmetic operations,

0-7803-8554-3/04/$20.00 ©2004 IEEE.

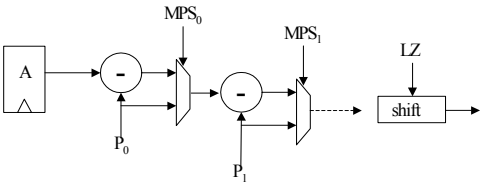

Fig. 1: Unrolled A Arithmetic

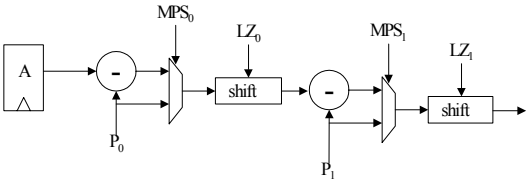

Fig. 2: A Arithmetic for two symbol processing

until renormalisation is required[1]:

$$A = A - (\overline{p_0} + \overline{p_1} + \overline{p_2} + ...)$$

In this case, we can simply cascade a series of subtractors (or adders in the C arithmetic), as shown in figure 1. Cascading arithmetic increases delay (if we ignore the multiplexors) by $1\frac{1}{16}$ for each extra subtractor. The method proposed in [2] uses multiple arithmetic to reduce the number of 'iterations' required to encode a stream of LPS symbols. This requires access to a lot of symbols, which is possible within the JBIG framework. It is less suitable for tight coupling to a bit-plane coder. The method of [7] simply tries to code as many symbols as possible each clock cycle. In this method, the bit plane encoder would produce multiple symbols and present them to the arithmetic coder. If only a partial number of these symbols can be consumed, the unprocessed symbols must be queued. If this occurs too often, the bit-plane coder must reduce the rate at which it produces symbols. This method appears attractive because of the simple arithmetic required to implement it. Consider, however, a pipelined system where probability estimation, queuing and arithmetic processing form separate stages. If renormalisation occurs in the arithmetic stage, the probability estimate for the context causing the event will change. This change will effect not only the stored state in the probability estimation stage, but also any symbols belonging to this context that have been queued for processing. This increase in complexity makes the Hypothesis Testing method unappealing.

A second method (the one proposed here) allows coding across renormalisation events, by providing mechanisms that allow renormalisation to occur during the arithmetic operations. The main difference between the two methods is that shifting networks need to be provided after each segment of arithmetic (see Figure 2). The output of each multiplexor is left shifted by the number of leading zeros (LZ) present.

In addition to the added shifting networks, another mechanism is required to handle renormalisation when both symbols belong to the same context. If an earlier symbol causes a renormalisation, the probability estimate for the subsequent symbol will be incorrect. This is addressed by modifying the PET, such that each entry is augmented to contain the probability estimates for each next state. Then, if a symbol causes renormalisation, the next probability estimate can immediately be forwarded to the symbol processing arithmetic.

Allowing coding across a renormalisation event complicates the byte-out procedures. Additional hardware is required to extract bytes from multiple shifted versions of the C register, a problem which we describe in more detail in Section 5. Interestingly, in spite of this extra complexity, this method requires less hardware to implement than the hypothesis testing coder, as it requires no queuing. It has the additional benefit of always consuming the same number of symbols.

[1]The term "hypothesis testing" refers to the need to test the hypothesis that additional symbols can be decoded without producing a renormalization event.

4. LEADING ZERO FORWARDING

Calculation of leading zeroes is required in order to determine how far to shift the A and C registers during renormalisation. The output of the leading zero calculator is fed into the shifting network. This long chain of logic can result in considerable delay [4].

Large improvements can be made by observing that most of the time (due to the skewed symbol probabilities in JPEG 2000), the arithmetic unit will set $A = \overline{p}$. Since $\overline{p}$ is stored in the probability estimation table; it is easy to extend the data stored to include the number of zero MSBs of each $\overline{p}$. This value can then be forwarded down the pipeline, removing the need for leading zero determination during when renormalising after substitution.

When substitution does not occur, the arithmetic will set $A = A - \overline{p}$. We know $\min(A) = 0x8000$ (due to renormalisation) and $\max(\overline{p}) = 0x5601$. Thus, $\min(A) = 0x29FF$ which has 2 leading zeros. For this case, we need only consider the two MSBs of A when calculating the number of leading zeros, which results in substantially less delay.

5. BIT STUFFING AND BYTE EMISSION

Emitting compressed data presents a challenge for MQ coder design. The shift of the C register can be such that it requires 0, 1 or 2 bytes to be emitted. For a two symbol coder, where the C register is updated twice, up to 4 bytes may be emitted. Previous implementations [8] have used a loop based approach for emitting bytes sequentially. In this case, the C register is left shifted incrementally, until all bits have been emitted. This technique can result in less than one symbol per clock cycle being consumed when a byte is to be emitted.

It is possible to output multiple bytes concurrently [4], so that the coder pipeline will never stall during the byte-out procedure. The final stage of our proposed coder (see Section 6) implements this form of byte emission. Arithmetic processing of the C register results in a 17 bit unsigned number (if the carry result is appended as the MSB). This result is first masked so that only the top bits, corresponding to the number of leading zero bits in the A register, remain (see Figure 3). The number is then right shifted by the number of bits already in the bit buffer, so that the carry bit overlaps with the last stored bit. This shifted result is then added and merged to the existing bits. The bit buffer contains at most 18 bits,

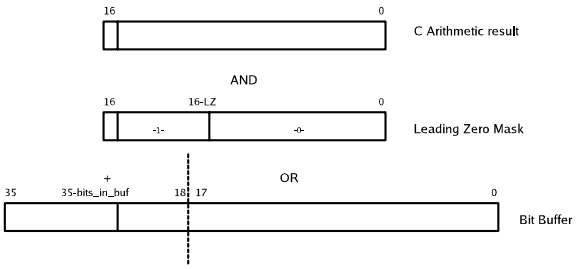

Fig. 3: Merging C arithmetic into bit output buffer

so only the top 18 bits require an adder. The lower bits can simply be copied.

If the number of bits stored is now greater than 27, two bytes are removed. If the first byte contains $0xFF$, a bit stuff is performed on the second byte, meaning only 7 more bits will be removed. If the 2nd byte out is $0xFF$, a zero is inserted at the start of the bit buffer, and the remaining bits are left shifted up to it. If less than 27, but greater than 19 bits are available, only one byte is removed. If it is $0xFF$, a zero is inserted at the start of the bit buffer, and the remaining bits are left shifted up to it.

Because of multiple byte processing, the C register may now have more than one shift operation performed on it. In order to extract bytes and perform bit stuffing in this system, the above procedure must be replicated and performed after each shift. Cascading the byte-out processing allows the bit buffer to remain the same size as for the single shift case.

6. TWO SYMBOL CODER IMPLEMENTATION

Figure 4 shows the block diagram for our proposed coder. This design has advantages over our previous Hypothesis Testing implementation, in that each stage in the pipeline can always cope with two sets of data. This simplifies the design, as no queueing of unused symbols is required. It also presents a simplified interface to earlier processing stages in the JPEG2000 coder.

The first stage implements state and probability estimation lookup. As mentioned in Section 3, the probability estimation table must be extended to include the probability estimates of each next state. Due to leading zero forwarding (Section 4), the number of leading zeros present in the extra probability estimates must also be stored. Each PET entry now contains 68 bits, consisting of 3×15 bits of probability information, 3×4 bits of leading zero information and 2×6 bits of next state information. The state memory must also have two write ports, to handle the case where two symbols of different contexts each cause renormalisation. In the case where two symbols from the same context cause renormalisation, the priority control block selects the most 'recent' next state information. Observe that multiplexors are provided to forward recent renormalisation data straight into the PET ROM, as this information is not written into the state memory until the next clock edge.

The A register arithmetic stage is implemented as per Section 3. If the two symbols are from the same context, and the first symbol causes renormalisation, the next probability estimate is extracted from the extended PET data and supplied to the arithmetic for the second symbol. Additional next state information is not

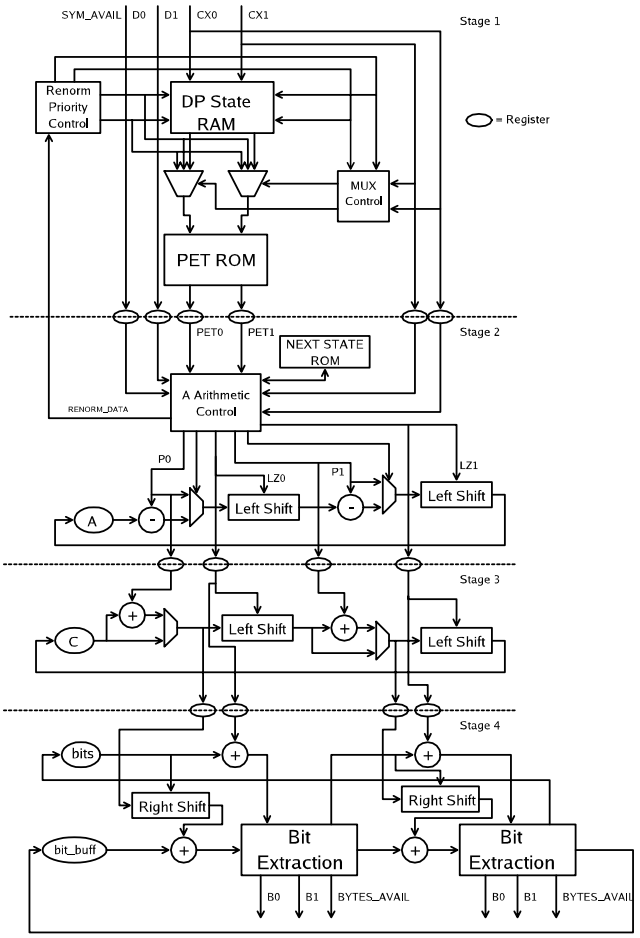

Fig. 4: Two Symbol Arithmetic Coder Implementation

provided in the extended PET data. In the case where both symbols belong to the same context and both cause renormalisation, the required next state information is missing. A small amount of additional ROM is provided to enable the new next state lookup. This could be achieved by augmenting the PET lookup table entries, however this method would require more storage space. Extending the PET lookup up would add an extra 4×6 bits, whereas each entry in the Next State ROM is only 2×6 bits. The addition of this extra ROM does not increase the delay of the arithmetic stage, as next state lookup is performed concurrently with the second symbol's arithmetic.

The third stage implements C register arithmetic, using the same data that was fed to the arithmetic units in the previous stage. Control signals can also be pipelined from the previous stage. Observe that C uses only 16 bits, because the non-renormalised data is fed directly to the fourth stage for byte-out processing. Any bits that would have been shifted out during renormalisation will be stored in this stage. C arithmetic could be performed concurrently to the A arithmetic stage, reducing the latency of the system by a clock cycle.

The byte-out processing stage consists of a bit buffer and bit count register. These are fed into two cascaded bit extraction blocks,

that perform the byte-out procedure described in Section 5.

7. IMPLEMENTATION RESULTS

The hardware utilisation and throughputs of the proposed coder, along with those of our previous Hypothesis Testing coder and the single symbol high throughput reference coder [4] are shown in Table 1. Memory storage space is approximately tripled relative to the reference coder. This is due to the lack of dual port storage in the FPGA. The cell count is more indicative of the overall size of the design; this has increased by a factor of 1.66, relative to the reference coder. At the same time, the throughput has been increased by a factor of 1.89.

The scalability of the proposed design is limited. Adding additional symbols would cause PET memory sizes to increase exponentially. For example, consider adding resources to process a third symbol concurrently. The state memory would need three write and three read ports. Under such conditions it is likely that the memory would be mirrored, doubling the number of stored bits. The PET ROM would need to not only forward the next two possible probability estimates for the second symbol, but an additional four for the third, to handle the case where all three symbols belong to the same context and the first two cause renormalisation. This may be reduced by employing an additional ROM in the arithmetic stage to look up the new probabilities, in a similar fashion to the way next states are retrieved in our current coder. Additional symbols would substantially increase the delay in the critical path. Each additional bit would add another 18 bit adder to the critical path, combined with a large amount of combinatorial logic and multiplexers to extract the compressed bytes from the bit buffer.

8. CONCLUSIONS

We have demonstrated that it is possible to implemented an arithmetic coder capable of coding across a renormalisation event. Our example architecture can consume two CXD pairs every clock cycle. It achieves a 1.89 times improvement in throughput at a cost of 1.66 times the hardware, when compared to a reference high throughput coder. When compared to the previous Hypothesis Testing coder, throughput is improved by 1.2 times, while using less hardware. Somewhat surprisingly, the complexity of concurrently coding across renormalisation events turns out to be less than the data queuing cost incurred by our previous Hypothesis Testing coder. Moreover, since this new coder can always consume two symbols per clock cycle, it presents a simpler interface to the bit-plane coder. This should allow tight coupling within the complete block coding engine, making it a more appropriate arithmetic coder for JPEG2000 than other high throughput methods.

Our ongoing work investigates the overall cost of using this arithmetic coder to improve block coding throughput. The cost of creating a multi-symbol bit-plane coder is yet to be identified, and needs to be compared with the cost of simply replicating the entire block coding engine. Providing multiple complete block-coders allows concurrent processing of code-blocks, representing an alternate means of increasing the encoder throughput. Indeed, it may be possible to combine both approaches, providing both speed improvements and hardware cost reductions.

System	Cells	RAM blk.	Clk. Rate	Throughput	Crit. Path
2 Symbol	1811	100	26.29 MHz	52.58×10^6	byte-out
Hypo. Test.	2549	93	28.40 MHz	37.72 to 43.45×10^6	queue
Reference	1089	31	27.78 MHz	27.78×10^6	arithmetic

Table 1: FPGA utilisation, clock rate and throughput (sym/sec)

9. REFERENCES

[1] Chung-Jr Lian, Kuan-Fu Chen, Hong-Hui Chen, and Liang-Gee Chen, "Analysis and architecture design of block-coding engine for EBCOT in JPEG 2000," in *IEEE Transactions on Circuits and Systems for Video Technology*, 2003, pp. 219 – 230.

[2] Gennady Feygin, Patrick Glenn Gulak, and Paul Chow, "Architectural advances in the VLSI implemntation of arithmetic coding for binary image compression," in *Proc. Data Compression Conference (DCC 94)*, 1994, pp. 254 – 263.

[3] William B. Pennebaker and Joan L. Mitchell, *JPEG Still Image Data Compression Standard*, chapter 13, pp. 231–232, Van Nostrand Reinhold, 115 Fifth Avenue, New York, New York 10003, 1993.

[4] Keng-Khai Ong, Wei-Hsin Chang, Yi-Chen Tsenf, Yew-San Lee, and Chen-Yi Lee, "A high throughput low cost context-based adaptive arithmetic codec for multiple standards," in *Proc. Of the International Conference for Image Processing (ICIP 02)*, 2002, pp. I–872 – I–875.

[5] "ISO/IEC international standard 11544:ITU-t rec t.82, coded representation of picture and audio information - progressive bi-level image compression," 1993.

[6] W. B. Pennebaker, J. L. Mitchell, G. G. Langdon Jr., 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, 1988.

[7] David Taubman, Erik Ordentlich, Marcelo Weinberger, Gadiel Seroussi, Ikuro Ueno, and Fumitaka Ono, "Embedded block coding in JPEG2000," in *Proc. International Conference on Image Processing (ICIP 02)*, SEPT 2000, vol. 2, pp. 33–36.

[8] Yun-Tai Hsiao, Hung-Der Lin, Kun-Bin Lee, and Chien-Wei Jen, "High-speed memory-saving architecture for the embedded block coding in JPEG2000," in *IEEE International Symposium on Circuits and Systems*, May 2002, vol. 5, pp. V–133 – V–136.