Hardware Acceleration of JPEG2000 Image Compression for Low-Power Embedded Applications

Darren Freeman **Greg Knowles**

School of Informatics and Engineering
Flinders University of South Australia,
PO Box 2100, Adelaide, South Australia 5001,
Email: `dfreeman@ieee.org` `gknowles@infoeng.flinders.edu.au`

Abstract

JPEG2000 is ideal as the next-generation still image compression standard, and it is expected to replace JPEG for embedded applications such as digital cameras. These typically require low-power operation, so by combining a low-power microprocessor with a hardware accelerator to form a System on Chip (SoC), the burden of rapidly coding JPEG2000 images is moved to the accelerator. The performance requirements of the processor are significantly relaxed, leading to lower power consumption and potentially cheaper fabrication.

An introduction to the benefits and inner workings of JPEG2000 is given. In particular, the ideas behind wavelet compression, arithmetic coding and subband sample coding are discussed. In the case of JPEG2000, the MQ Coder functions as the arithmetic coder while subband samples are coded using the Embedded Block Coding with Optimal Truncation (EBCOT) algorithm.

Both the MQ Coder and EBCOT are implemented in the hardware accelerator, with a complete System on Programmable Chip (SoPC) prototype demonstrated here. The prototype runs on an Altera ARM-Based Excalibur chip, type EPXA10F1020C1, tested using the Embedded Linux operating system and custom software. Performance estimates are derived from simulation and synthesis results and they are scalable with the number of block coders used. For the EPXA10 chip we obtain 5 megasamples per second, per coder, with a maximum of 20 coders.

Future work includes adding functionality such as a Discrete Wavelet Transform (DWT) engine, optimising for low power, and fabrication of an Application-Specific Integrated Circuit (ASIC) which we expect would give between two to five times the performance provided here, depending upon the silicon process used.

Keywords: Image Compression, JPEG2000, EBCOT, Embedded Systems, SoC, SoPC.

1 Introduction

The flexibility and performance of JPEG2000 make it ideal as the next-generation still image compression standard. If a digital camera used JPEG2000 for it's image compression, it would have the ability to efficiently decrease image quality without having to decompress each image and recompress it. In fact, JPEG2000 allows a multitude of image manipulations without altering much of the compressed data. This can save a substantial period of time and reduce the additional compression artifacts introduced by most other popular compression schemes. Even with very low bit-rates that were previously considered unacceptable, JPEG2000 can produce results that are soft and visually pleasing.

Copyright ©2003, Australian Computer Society, Inc. This paper appeared at the Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, Australia. Conferences in Research and Practice in Information Technology, Vol. 16. Michael Oudshoorn, Ed. Reproduction for academic, not-for profit purposes permitted provided this text is included.

The last couple of years have seen the introduction of a handful of special-purpose chips designed to improve the performance of embedded applications utilising JPEG2000. Some are separate components while others are sold in the form of soft Intellectual Property (IP) modules to be incorporated into a central System on Chip (SoC) or System on Programmable Chip (SoPC).

In this paper a SoC (or SoPC) hardware accelerator for JPEG2000 will be presented. With a low-power microprocessor and an accelerator on a single chip such as the one described in this paper, the device will be able to encode and decode JPEG2000 images with superior speed and yet lower overall power consumption. This aim is achieved by reducing the performance of the microprocessor and moving the computationally expensive operations into the accelerator. When not in use, the accelerator can then be placed into a low-power standby mode.

2 Background

2.1 JPEG2000

JPEG2000 is a still image compression standard intended to replace many existing standards. It is the newest international standard to be produced by the Joint Photographic Experts Group (JPEG) and it has recently been published as an ISO/IEC standard and as an ITU-T recommendation, which is royalty-free (Taubman and Marcellin 2002, pp. 399). It is capable of replacing most of the currently used standards including the original JPEG, usually with superior compression for the same image quality, or alternatively with superior image quality for the same file size.

In addition to supporting both lossy and lossless encoding, JPEG2000 also gives superior image quality with low bit rates. This means that the decoded image can be anywhere from a perfect reconstruction of the original, to a very blurred rendition. Note that the blocky artifacts seen in JPEG compression are not present in JPEG2000, so the image degrades in a more visually pleasing manner. In addition to the ease of adjusting the compression parameters such as bit-rate, JPEG2000 also allows many of these parameters to be changed after encoding, with minimum processing. For example, it is now possible to view a JPEG2000 image on a Web page in thumbnail form, which was generated in real-time by the machine serving the image to the Web browser. This process, called *transcoding*, allows the server to extract the relevant information from the full quality image without the usual decoding and re-encoding. If the image is enlarged by the user, the remaining coded image data is added to the data already received. Instead of starting again, the server continues to send additional bits to improve the image quality

gradually until it reaches the maximum quality of the original image. (Taubman and Marcellin 2002)

In addition to progression by quality, given in the example above, there are many other progression orders available to the JPEG2000 encoder, depending upon the application. For example, a Web page may prefer progression by quality while a scanner or printer may require progression based on stripe scanning across the image. With JPEG2000, even the progression order may be easily changed by transcoding. Instead of decoding the image and re-encoding it in a different order, the bytes of the file are simply reordered and the necessary parameters are changed to let the decoder know of the new ordering. Also, the quality of the image may be reduced after encoding, which would allow a digital camera to reduce the quality of images already stored, to make room for new pictures. With JPEG, this is impossible. The camera would instead need to decompress the many images and recompress them, usually with more artifacts introduced. The time taken to achieve this could be considerable. (Taubman and Marcellin 2002)

Another useful addition to JPEG2000 is the Region of Interest (ROI). Many regions can be drawn onto the image during encoding, which specify which areas should receive a priority when allocating bits to the image. In this way, very important regions such as faces or commercial products, can receive more emphasis and have a higher quality, while less important regions such as sky, grass and traffic, will receive more blurring. (Taubman and Marcellin 2002)

2.2 Wavelet Compression

One of the salient features of the JPEG2000 standard, giving it resolution scalability, is the use of the two-dimensional Discrete Wavelet Transform (DWT) to convert image samples into a more compressible form. Unlike others using the Discrete Cosine Transform (DCT), this scheme is able to vary the resolution of the reconstructed image when decoding, rather than when re-encoding. It also gives superior compression performance by taking advantage of redundant information spread over a much larger area than the 8×8 blocks typically used with the DCT, such as in the original JPEG standard. Hence the compression artifacts are spread over a correspondingly larger area, reducing visual impact.

The image canvas is divided into regular tiles, making the DWT process more manageable in terms of memory requirements, although the entire image may be treated as a single tile. After a possible colour transform, the individual components of the tiles are treated separately, and each is wavelet transformed. The first octave of the wavelet decomposition is depicted in Figure 1, for a black square on white background. The result of the 2D DWT is four separate subbands, each containing one quarter of the source samples. The high-pass subbands are depicted here using black for large absolute value and white for values near zero. Given these subbands, the Inverse Discrete Wavelet Transform (IDWT) achieves perfect reconstruction.

Note that the process of performing a DWT in one dimension is essentially to low-pass and high-pass filter the input, giving two output subbands. Every other sample is then discarded, to keep the number of samples the same. In the case of the 2D DWT, this process is performed on the rows and columns (in either order) to produce four subbands. These subbands are labelled here as lowpass (LL), horizontally-highpass (HL), vertically-highpass (LH) and diagonal-highpass (HH). Note that the LL subband is just a filtered and down-sampled version of the original tile-component, while the HL subband

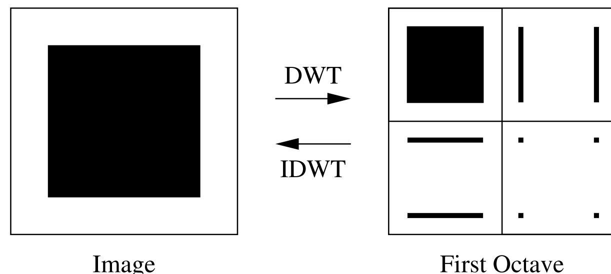

Figure 1: The 2D Discrete Wavelet Transform

contains mostly small values with vertically oriented edges remaining. Similarly the LH subband contains horizontal edges while the HH subband contains diagonal edges.

Since the three highpass subbands contain mostly small values with occasional edges running in predictable directions, they are more suitable for compression than the original tile-component. However, the LL subband is no easier to compress. The logical choice is to recursively transform the LL subband to produce the second octave, and again on the next LL subband to produce the third, etcetera. After the desired number of octaves, most of the samples are small values, with predictable edges remaining. Figure 2 depicts the second octave of the 2D wavelet decomposition, where the DWT and IDWT act now on the LL subband from the first octave.

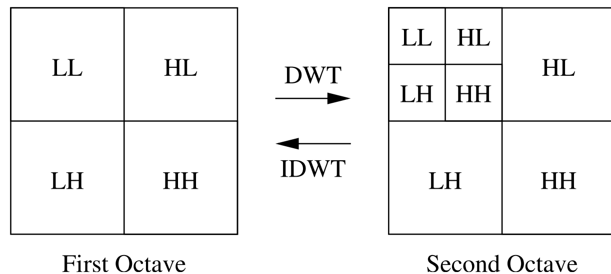

Figure 2: Wavelet Decomposition (Second Octave)

The exact results depend upon the wavelet kernel chosen, which determines the filters used. In the case of JPEG2000 part one, two kernels are specified, known as the 5/3 and the 9/7 kernel. The 5/3 transform works with integers and is used for lossless compression, while the 9/7 transform works with floating- or fixed-point quantities and is used when better lossy compression is desired and more resources are available. If the 9/7 wavelet decomposition is performed, the resulting samples are quantised using a dead-zone quantiser in which the zero bin is twice the width of all other bins. The bin width is chosen so that the integer result has greater bit-depth than will eventually be encoded. Unlike the original JPEG algorithm, the quantiser is not normally used to achieve lossy compression, instead extra bits are produced and then some are discarded after coding, during subsequent bit-rate adjustments.

2.3 MQ Coder

The MQ Coder, created by IBM, is the low-level workhorse of the JPEG2000 compression system. On symbols with well-estimated probabilities, the MQ Coder is able to achieve coding efficiencies nearing 100%. It is a variation of the arithmetic coder, in which an *a priori* unknown number of binary sym-

bols is encoded into a variable-length binary codeword. This codeword effectively represents a binary fraction from the interval $[0, 1)$ obtained in the following way, for a basic arithmetic coder.

Start with the coding interval $[0, 1)$ on the real line. This interval represents "no information" and the probability of it occurring is 1. Then encode the first binary symbol by subdividing the interval into an upper subinterval and a lower subinterval based on some predefined assignment. For example, if the symbol is a '0' then we may choose the latter.

This resulting subinterval should have a length equal to the probability of occurrence, p, multiplied by the original length (1 in this case). To code subsequent symbols the coding interval is recursively subdivided until the operation is complete, with the resulting interval having a length $l = \prod_i p_i$. To signal the final coding interval to the decoder we may choose any binary fraction lying within the interval, and the length of the smallest such fraction will be approximately $-\log_2 l$ bits. The amount of information (*entropy*) coded by each symbol is defined as $-\log_2 p_i$ bits, with the total entropy then given by:

$$\begin{aligned} I &= \sum_i -\log_2 p_i \text{ bits} \\ &= -\log_2 \prod_i p_i \text{ bits} \end{aligned}$$

This is just $-\log_2 l$, so the number of bits in the resulting binary fraction will be very close to the total entropy coded. This is why arithmetic coding can be so efficient when the probabilities p_i are well-estimated.

Even without knowledge of the total number of symbols encoded, the first symbols may be decoded by noting that the binary fraction lies inside the final coding interval, and hence inside all earlier coding intervals. Because the decoding operation doesn't make use of future probability estimates, these may be determined as needed from previously decoded symbols.

The MQ Coder makes use of many carefully chosen approximations to eliminate the use of a multiplication when updating the coding interval, and to use a probability estimator with only 47×2 states for each adaptation context. It turns out that these approximations are in fact possible without greatly sacrificing compression efficiency, and in return the computational resources required are drastically reduced in comparison with earlier arithmetic coders.

The symbols are supplied to the MQ Coder along with *context labels*, which determine what the symbols represent. It adaptively estimates the symbol probabilities based upon these labels and the state of the corresponding adaptation context. Armed with this probability estimate, the MQ Encoder incrementally constructs the codeword, which is output gradually as the symbols are received. Likewise the MQ Decoder must be given the same context labels in order to recover the symbols, since the probability estimates must be the same as those used in encoding. Unfortunately this introduces a performance limitation since the MQ Decoder is kept waiting for the next context label while the previous symbol is processed by the external system, however encoding doesn't suffer this dependency.

2.4 The EBCOT Algorithm

In order to efficiently encode the subband samples of the wavelet decomposition, some means must be employed to scan the samples and produce symbols and context labels for the MQ Encoder. It must also be possible to recover the context labels as the decoder reconstructs the samples. One clever way to accomplish this is known as zero-tree coding. In this scheme, samples from the same image region but different octaves are coded together, thus taking advantage of redundancy between the octaves. (Lewis and Knowles 1991 and 1992, Shapiro 1993, Said and Pearlman 1996)

An alternative first proposed by Taubman (2000), and used in JPEG2000, is the algorithm known as Embedded Block Coding with Optimal Truncation (EBCOT). In this technique, the subbands are coded individually, in manageable code-blocks of a regular size and without a significant loss in compression performance due to independent coding of the subbands. An advantage of the EBCOT algorithm is that the resulting bit-stream is locally embedded, meaning that each codeword may be truncated at one of many locations, to yield a reduction in quality. This reduction occurs over the entire code-block, and has a finer granularity than one bit-plane, due to its use of three passes per magnitude bit-plane. Together with a method for calculating the optimal truncation points, this algorithm is able to tightly control the resulting bit-rate whilst ensuring that the desired quality metric is optimised. (Taubman and Marcellin 2002)

After block encoding has been performed, the resulting codewords are sequenced into a JPEG2000-compliant code-stream, which may optionally be encapsulated by the JP2 file format. The codewords need not be presented in any particular order, and they may be split into many fragments, each contributing to a given quality layer. In this way the resulting code-stream can be made globally embedded, meaning that truncation of the code-stream results in a nearly optimal coding at a reduced bit-rate. Hence the code-stream may be efficiently transcoded into a lower bit-rate by reducing the quality or resolution.

The encoding operation of the Embedded Block Coder is as follows. Its input is a two dimensional code-block of integer samples, whose binary representations are sign-magnitude. Its output is a binary codeword produced by the internal MQ Coder. Samples are coded one magnitude bit-plane at a time, starting from the most significant bit-plane, and sign bits are coded when the samples first become significant (i.e. their first nonzero bit is coded). Deferring coding of the sign bits in this way allows them to be discarded appropriately during truncation.

Each bit-plane is scanned in a stripe-based scan order, depicted in Figure 3. The block is divided into horizontal stripes, four samples high. Stripes are scanned in a raster scan, first down a column of four samples and then down the next column, from left to right. After the top-most stripe comes the one immediately below it, and so on down to the bottom-right corner where the next pass begins again from the top-left corner.

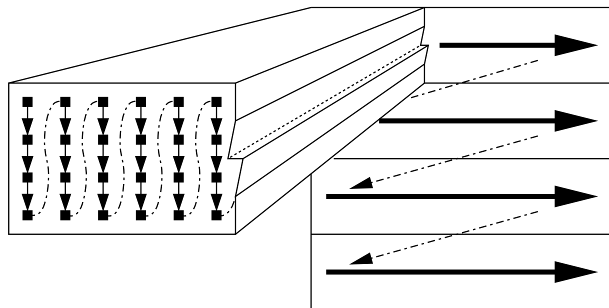

Figure 3: Stripe-Based Scan

The bit-planes are coded using three separate passes, known as Significance Propagation, Magnitude Refinement and Cleanup. Significance Propagation only processes those samples which are currently insignificant but for which one or more immediate neighbour is significant. Magnitude Refinement processes only those samples that were significant before the current plane, and codes their magnitude bit. Cleanup processes those samples remaining, which were insignificant in the previous plane and have no significant neighbours. It codes their significance and includes a run-length mode for better efficiency when coding regions of zeros.

The first magnitude plane is coded using only the Cleanup pass, since there is no significance information to utilise, but after that each plane is coded using the three passes in the order given. The encoder may be asked to terminate after any pass, although often the decision will be deferred until many blocks have been encoded. Hence truncation is the primary mechanism of controlling the bit-rate, and truncation is allowed after any pass. The use of these three passes allows finer control than simply generating fewer bits from the quantiser, as those samples likely to become significant are coded earlier and thus may receive finer quantisation than samples likely to remain zero (by a factor of two).

The context labels needed by the MQ Coder are generated from the states of the immediate neighbours in a 3×3 window centred on the sample being coded. Since these states are synchronised between the encoder and decoder, they are both able to generate the same context labels. There are 18 adaptive contexts and one non-adaptive one that assumes a uniform distribution. Nine of these adaptive contexts are used for significance coding, one represents whether a run is encountered in Cleanup pass, five are sign coding and three are magnitude refinement.

3 Development Platform

The prototype Block Coder (with MQ Coder) was developed in VHDL using the Altera Quartus and Synopsys tools.

The final prototype embedded system was demonstrated on a development board featuring the new Altera ARM-Based Excalibur chip, type EPXA10F1020C1. This integrated circuit features a large (10^6 gates typical) Programmable Logic Device (PLD) and an embedded stripe with "ARM 922T 32-bit RISC processor core operating at up to 200 MHz" (Altera 2002a).

To ease development and testing, an embedded GNU/Linux operating system was installed on the board. It enables the software to be developed and tested using a standard PC running GNU/Linux and then simply recompiled for the target board.

4 Architecture

The original architecture used to implement the Embedded Block Coder in hardware was suggested by Taubman and Marcellin (2002, pp. 654). In a simplified form it is presented here in Figure 4. With this architecture as a starting point, the Block Coder was designed and successfully tested in a behavioural simulation. It has been designed to encode and decode blocks of a fixed size chosen at manufacture, typically 32×32 or 64×64 which are the sizes specified in profile-0 (ISO/IEC 2000). The sample size is also chosen at manufacture, normally equal to the largest sample size to be processed plus guard bits to allow for the increase in precision required by the image transformations. For RGB colour images with 8-bit samples per channel, lossless compression with up to five octaves of the wavelet decomposition would require 12 bits[1] per sample, while lossy compression with the 9/7 wavelet would require only 9 bits[2] for any number of octaves (Taubman and Marcellin 2002, pp. 668). These code-block samples are stored in sign-magnitude form with separate sign and magnitude memory, so 9 bits per sample is actually a convenient size.

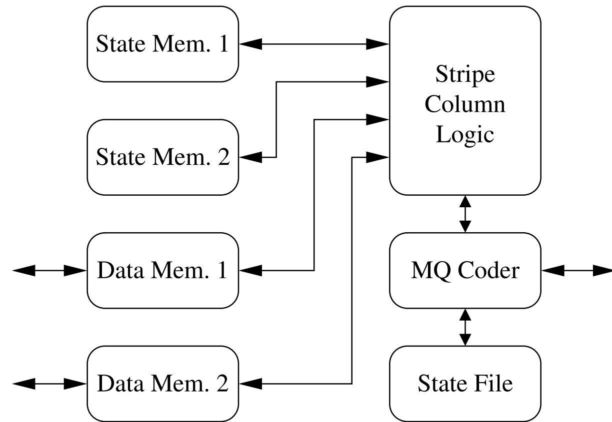

Figure 4: Simplified Architecture of Block Coder (Taubman and Marcellin 2002, pp. 654)

The operation of this block diagram for encoding is as follows. The code-block samples are loaded into the two data memories, with Data Memory 1 holding the sign bit-plane and Data Memory 2 holding one or more magnitude bit-planes. The two data memories output their contents one column at a time, in stripe order. Since all magnitude bit-planes except the first are processed in three passes, Data Memory 2 loops over the same plane until signalled to move to the next-most-significant plane.

The two state memories each contain enough storage to hold two state bits per code-block sample, and they are initialised to zeros. Their behaviour is identical for both encoding and decoding operations, and their contents must be the same at each step of the encoding and decoding operation to allow the correct context labels to be generated for the MQ Coder. State Memory 1 provides a 6×3 (vertically oriented) context window around the current stripe column, and the bits in the current column may be written to. The two bits per sample represent the significance state and a copy of the sign bit (since Data Memory 1 doesn't provide a context window). State Memory 2 only outputs the bits associated with the current column and hence its interface is made considerably simpler. Again these bits may be written to, and these two bits represent the Significance Propagation pass membership and the significance from the previous plane.

The Stripe Column Logic is a Finite State Machine (FSM) that is responsible for performing the coding passes and generating the appropriate context labels for each symbol. The logic processes each column in a variable number of cycles, generating or receiving symbols depending upon whether encoding or decoding. The state memories are updated once each cycle and if decoding the data memories may also be written to. When the current stripe column is processed,

[1] plus one more guard bit for assuring perfect reconstruction beyond five octaves

[2] more bits are available but the quantiser need not output more than one guard bit and eight sample bits for full quality

the data and state memories are signalled to advance to the next column, and processing continues until all coding passes are completed.

The MQ Coder contains a probability estimation FSM with Context State File, representing the current probability estimates for each context label. This file is updated by the FSM as symbols are received or generated, hence keeping it synchronised between the encoder and decoder. There are 47×2 possible states for each of the 19 context labels, and each state has associated with it a 16-bit scaled probability estimate and two possible next states based upon the JPEG2000 prescribed transition table.

Given the probability estimate, the MQ Coder updates its internal state variables to encode or decode the next symbol. The two main registers, named A and C, represent the length and lower limit of the coding interval respectively. Their implementation is that of a shift register with parallel access, since A is continually becoming smaller and C is being shifted in or out as the codeword is processed. After one extra bit of information has been added to or stripped from the evolving codeword, the registers shift, and after 8 bits (or 7 with bit-stuffing) one byte will be transferred between C and the external codeword buffer.

An example of the MQ encoder's operation will be given here to demonstrate how hardware-friendly the MQ Coder is. The probability estimator maps the context label to the 16-bit probability estimate p of the least probable symbol (LPS), and s the identity of the most probable symbol (MPS). The symbol to encode is x. The A register is 16 bits while C is 28 bits when encoding.

```
if A < 2p                         Conditional exchange
        s ⇐ s̄                              MPS ⇔ LPS
if x = s                              Select subinterval
        A ⇐ A − p               MPS = upper subinterval
        C ⇐ C + p
else
        A ⇐ p                   LPS = lower subinterval
```

The conditional exchange step is taken if the MPS actually receives the smaller subinterval due to the multiplier-free approximation, which may happen when the probability estimate is near 50%. The remedy is to swap MPS and LPS for this symbol. After the above steps are performed, A is checked to verify that its most significant bit (MSB) is still set to '1'. If not, both A and C are left-shifted until this condition is met. During this process it may be necessary to output another codeword byte. Also, whenever any shifts are required the probability estimator is signalled to update its estimate based on whether the symbol was an MPS or LPS. This behaviour, rather than updating the estimate on every symbol, was chosen to increase the performance of software implementations, although it has little impact on hardware. The MQ decoder performs a similar sequence of operations.

The architecture discussed above served as a functional starting point for the development of the Embedded Block Coder with MQ Coder.

5 Prototype Chip

In order to demonstrate the Embedded Block Coder working in hardware, it was necessary to create many more support modules to form a complete embedded system. The ARM-Based Excalibur chip provides the Embedded Stripe, a basic embedded system for the user to extend in the PLD. Contained within this stripe is an ARM922T microprocessor, some single- and dual-port SRAM, an SDRAM controller, a UART and many more modules to create a single-chip computer capable of running a modern OS. The PLD is programmed typically during boot-up, and it communicates with the Embedded Stripe via two Advanced High-performance Bus (AHB) bridges and via dual-port SRAM (Altera 2002a).

5.1 Block Diagram

Refer to Figure 5 for a simplified block diagram of the demonstration of our JPEG2000 accelerator. It consists of an array of Embedded Block Coders which are interfaced to the Embedded Stripe via the two AHB bridges, one being a Stripe-to-PLD bridge, the other being PLD-to-Stripe (bridges not shown). The ARM processor is able to perform Memory-Mapped I/O (MMIO) via the Register File connected to the Stripe-to-PLD bridge. Once coding has been initiated via MMIO, the data is moved between the coders and system memory using Direct Memory Access (DMA). The authors have included an option of connecting the coders directly to the dual-port SRAM of the Embedded Stripe, for applications where the PLD-to-Stripe bridge provides insufficient bandwidth. In this case, codewords may be located anywhere in system memory but the sample data is constrained to lie within the 128 kB of dual-port SRAM available.

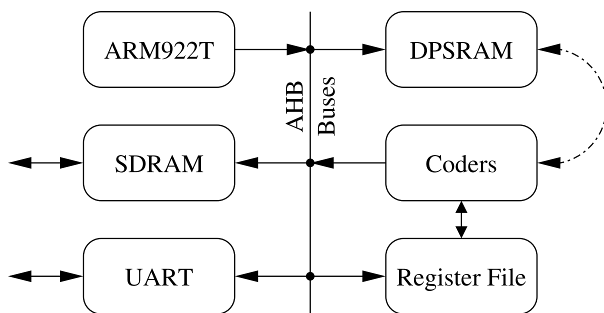

Figure 5: Prototype Block Diagram

5.2 Hardware / Software Interface

To make the prototype testable and useful, a software driver has been developed concurrently with the hardware development effort. This driver is divided into user-space and kernel-space, with the user-space code written in C++ and intended to be OS independent. The kernel-space code is responsible for accessing the hardware directly, and is currently implemented as a Linux kernel module (necessarily written in C). This module presents a Unix-style interface for user-space applications in `/dev` and it supports software simulation of the hardware and thus it is possible to develop the application software without access to the accelerator in hardware form.

Testing has been performed on this software simulation by incorporating the user-space code into a privately modified version of a fully functional JPEG2000 implementation, Kakadu 2.2.3 by Dr D. Taubman, which was also written in C++. Perfect reconstruction has been achieved using this combination, and subsequently the same drivers allowed Kakadu to take advantage of the prototype accelerator. Once our drivers were verified against Kakadu we developed a new JPEG2000 implementation to take advantage of the parallel processing offered by our accelerator.

5.3 Simulation Results

Using simulation tools some performance estimates were made, presented here in Table 1. These estimates predict the throughput per cycle for each module. Estimation of the throughput of the Block Coder is difficult because the results are dependent upon the image data being processed. Since the timing of the Block Coder at the single cycle level is similar to the "concurrent membership testing" model discussed by Taubman and Marcellin (2002, pp. 658–661), their investigation is used to obtain these estimates. Two test scenarios are considered here, one for lossless compression, and the other for lossy compression at a bit-rate of 1.0 bit per sample. Both scenarios use 11-bit code-block samples and therefore the maximum number of passes is 28. Since the number of passes for lossy compression is unknown prior to encoding the block, the throughput estimate given here relies upon a good software algorithm for predicting this[3]. Decoding will of course only process the required number of passes.

Module Name	Throughput per cycle
Block Coder (lossless)	0.13 samples
Block Coder (lossy)	0.39 samples
MQ Coder	almost 1 symbol
Block DMA (AHB)	0.5 samples
Block DMA (DPSRAM)	2 samples
Codeword DMA	32 bits

Table 1: Estimated throughput for each module

Although throughput per cycle is essentially the same for encoding and decoding in this architecture, maximum clock frequencies may differ. Block DMA achieves half of its theoretical maximum throughput due to a known bug in the Excalibur AHB bridges, which requires an idle cycle between every burst (Altera 2002b). Future revisions of the Excalibur chip should allow close to maximum.

First we consider the use of DMA to fetch/store samples from/to system memory. Since it takes a significant time to move samples to and from the data memories of each block coder, and during this time the coder is idle, the average throughput for each coder will be slightly lower. This may be calculated as follows (assuming all modules share one clock):

Coder cycles per sample	$1 \div 0.13 = 7.7$
Add idle cycles per sample	$7.7 + 2 = 9.7$
Average samples per cycle	$1 \div 9.7 = 0.10$

If instead we use dual-port SRAM to hold the samples, we may fetch two 16-bit words from different locations at the same time in the Excalibur chip (Altera 2002a). Hence only half of a cycle is consumed per sample when transferring the code-blocks. Repeating the calculation yields:

Average samples per cycle	$1 \div 8.2 = 0.12$

5.4 Synthesis Results

Using the synthesis tool (LeonardoSpectrum-Altera), and specifying our Excalibur as the target technology, it was possible to obtain estimates of the clock frequencies allowed by each module. Synthesis was performed with a block size of 32×32 and 9-bit samples to make optimum use of the memory in the PLD. The resulting clock frequency estimates are given in Table 2 and when combined with the previous throughput estimates per cycle they allow us to derive per-second performance estimates for the prototype implemented using the Excalibur chip. Note that these timing estimates are subject to increase as the work progresses, and that a fabricated ASIC can be anticipated to operate two to five times faster than a programmable chip, depending upon the silicon process used.

Module Name	Clock / MHz
Embedded Block Coder	35
encode only	47
MQ Coder	49
Stripe Column Logic	42
Register File	84
Block DMA (AHB)	49
Block DMA (DPSRAM)	N/A
Codeword DMA	102

Table 2: Estimated clock frequencies for each module

Note that the MQ Encoder may be operated at a different clock frequency from the rest of the Block Coder, and the DMA units operate at a common bus frequency, which is independent of the main AHB bus frequencies due to the use of AHB bridges. Since decoding is a slower operation than encoding, the Embedded Block Coder clock frequency can be boosted during encode operations.

Let's derive some per-second throughput estimates for lossless encoding operations on a single coder, using Block DMA. Due to buffering around the MQ Encoder, we may use the throughput estimate given for the entire Block Coder. Assuming that a common clock frequency of 50 MHz is used, we obtain a per-second throughput of:

$$0.10 \times 50 \text{ MHz} = 5.0 \text{ Msamples s}^{-1}$$

Assuming a three-component colour image compressed in YCbCr (YUV) 4:1:1 format:

$$5.0 \text{ Msamples s}^{-1} \div 1.5 = 3.3 \text{ Mpixels s}^{-1}$$

Now examine the bus throughput to find the maximum number of coders that can be kept near full capacity with Block DMA (assuming 80% of maximum bus throughput):

$$\begin{aligned} 80\% \times 0.5 \times 50 \text{ MHz} &= 20 \text{ Msamples s}^{-1} \\ 20 \text{ Msamples s}^{-1} \div 1.5 &= 13.3 \text{ Mpixels s}^{-1} \end{aligned}$$

This would require 4 coders working simultaneously to achieve, while the Excalibur EPXA10's PLD could hold up to 20 (constrained by the SRAM requirements of 1.8 kB per coder). Hence the Excalibur could achieve far greater performance if the bus bottleneck were alleviated, and for applications requiring more coders we next consider replacing Block DMA with Block to DPSRAM using the same arguments:

$$\begin{aligned} 0.12 \times 50 \text{ MHz} &= 6.0 \text{ Msamples s}^{-1} \\ 6.0 \text{ Msamples s}^{-1} \div 1.5 &= 4.0 \text{ Mpixels s}^{-1} \end{aligned}$$

Now we calculate the throughput of the Block to DPSRAM unit, assuming that the data in DPSRAM is always available:

$$\begin{aligned} 2 \times 50 \text{ MHz} &= 100 \text{ Msamples s}^{-1} \\ 100 \text{ Msamples s}^{-1} \div 1.5 &= 67 \text{ Mpixels s}^{-1} \end{aligned}$$

This would require 17 coders to achieve. This is now close to full utilisation of the Excalibur chip.

[3] Kakadu 2.2.3 contains such an algorithm

Note that in a digital camera application the number of samples to decode will be a small fraction of those samples that were originally encoded because the display is a much lower resolution than the image sensor, hence it may be worthwhile to make some of the coders only capable of encoding, to reduce chip area and power consumption.

6 Future work

Continuing improvements are being made to the prototype discussed here. The performance estimates and comparisons presented above are expected to improve as more work is performed. Now that a working chip has been demonstrated, it may be possible in the future to commercialise this design and integrate additional functionality. One important undertaking would be to completely verify compliance with every aspect of ISO/IEC (2000).

One goal of this project is to optimise the design for low-power operation. Further to that goal the authors intend to add an on-chip DWT engine, which will work with the block coders to automatically convert a tile of image samples located in dual-port SRAM into a string of codewords in system memory (and vice-versa), all with minimum microprocessor overhead. Eventually it should be possible to put the processor into a power-saving sleep mode while the tiles are being processed.

The JPEG2000 application software and drivers can be further improved to take advantage of the additions mentioned above.

7 Conclusion

In this paper a hardware accelerator for JPEG2000 image compression has been presented. It is intended to relieve the system microprocessor of the coding tasks which are most expensive to perform in software, but which can be performed more efficiently in hardware. By incorporating this accelerator into a System on Chip (SoC) or System on Programmable Chip (SoPC), the performance requirements of the core processor are significantly relaxed, leading to a design with lower power consumption and potentially cheaper fabrication.

The overall architecture of the prototype chip has been described, and performance estimates obtained have been presented along with their justification. Even in programmable logic, the prototype design is capable of high performance. With further improvements, the design will be suitable for fabrication on an Application-Specific Integrated Circuit (ASIC).

In conclusion, this project has developed a hardware design that has been tested and demonstrated on a development board. The design is now ready for the next stage of development, to optimise for low-power operation.

8 Acknowledgements

The authors wish to thank Dr David Taubman for helpful correspondence, and Professor Michael Marcellin for co-authoring the book with David, referenced below. This book has proven to be our most valuable reference to JPEG2000. This project is supported by the Australian Research Council (ARC) Discovery Grant DP0210399 entitled “Low-power architectures for the wavelet transform and JPEG2000”.

References

Altera (2002a), *Excalibur devices, hardware reference manual, July 2002, version 3.0.* [Online, accessed 2 September 2002]. URL: `http://www.altera.com/literature/manual/mnl_arm_hardware_ref.pdf`

Altera (2002b), *Excalibur EPXA10 devices, errata sheet, June 2002, version 2.1.* [Online, accessed 25 June 2002]. URL: `http://www.altera.com/literature/ds/es_epxa10.pdf`

ISO/IEC (2000), *ISO/IEC 15444-1 Information technology - JPEG2000 image coding system - part 1: core coding system.* ISO/IEC ref. no. SO/IEC 15444-1:2000(E)

Lewis, A. S. & Knowles, G. (1991), ‘A 64 kB/s video codec using the 2D wavelet transform’, *Proceedings of Data Compression Conference, Snowbird, Utah*, IEEE Computer Society Press

Lewis, A. S. & Knowles, G. (1992), ‘Image compression using the 2D wavelet transform’, *IEEE Transactions on Image Processing* 1, 244–250.

Said A. & Pearlman, W. (1996), ‘A new, fast, and efficient image codec based on set partitioning in hierarchical trees’, *IEEE Transactions on Circuits and Systems for Video Technology* 243–250 June 1996

Shapiro, J. M. (1993), ‘Embedded image coding using zerotrees of wavelet coefficients’, *IEEE Transactions on Signal Processing* **41**(12), 3445–3462.

Taubman, D. S. (2000), ‘High performance scalable image compression with EBCOT’, *IEEE Transactions on Image Processing* **9**(7), 1158–1170.

Taubman, D. S. & Marcellin, M. W. (2002), *JPEG2000: image compression fundamentals, standards and practice.* Massachusetts USA, Kluwer Academic Publishers.