DESIGN AND IMPLEMENTATION OF AN EMBEDDED NIOS II SYSTEM FOR JPEG2000 TIER II ENCODING

Thesis

Submitted to

The School of Engineering of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree of

Master of Science in Electrical Engineering

By

John M. McNichols, B.S.C.E, B.E.E.

UNIVERSITY OF DAYTON

Dayton, Ohio

August, 2012

DESIGN AND IMPLEMENTATION OF AN EMBEDDED NIOS II SYSTEM FOR JPEG2000 TIER II ENCODING

Name: McNichols, John Michael

APPROVED BY:

ABSTRACT

DESIGN AND IMPLEMENTATION OF AN EMBEDDED NIOS II SYSTEM FOR JPEG2000 TIER II ENCODING

Name: McNichols, John Michael
University of Dayton

Advisor: Dr. Eric Balster

Image compression standards continually strive to to achieve higher compression ratios while maintaining image quality. In addition to these goals, new applications require expanded features and flexibility as compared to existing compression standards. JPEG2000 is the latest in the line of image compression standards, offering higher compression ratios than its predecessor JPEG while maintaining comparable image quality. In addition, JPEG2000 offers an extended range of features including bit-rate control, region of interest coding and file-stream scalability with respect to resolution, image quality, components and spatial region. However, these additional features come with associated costs, primarily in the form of computational complexity. Due to the increased computational costs, JPEG2000 has not achieved the same wide-spread usage as JPEG. However, there are a number of specialized applications such as medical imaging and wide-area surveillance which demand the extended features offered by JPEG2000. These applications generally deal with high resolution imagery, resulting in extremely long encoding times when using consumer off the shelf platforms. As a result, many hardware implementations of the most computationally complex portions of JPEG2000, namely Tier I encoding, have been proposed. This thesis proposes using an embedded soft-core processor on a Field Programmable Gate Array (FPGA) for JPEG2000 code stream organization, known as Tier II. The soft-core processor chosen, Altera's NIOS II core, is coupled with existing Discrete Wavelet Transform (DWT) and Tier I implementations on a single FPGA to realize a fully embedded JPEG2000 encoder. Results show the feasibility of using an embedded soft-core processor on a FPGA to perform Tier II processing for JPEG2000.

ACKNOWLEDGMENTS

I would like to thank all who have helped to support me through these years as I begin my career, especially:

• **My family:** For always supporting and encouraging me, and for giving me every opportunity to succeed.

• **My Friends:** For providing support and a distraction from the rigors of graduate school.

• **Dr. Eric Balster:** For giving me this opportunity and for providing knowledge, experience, and motivation throughout my graduate education.

• **Dr. Frank Scarpino:** For serving as an example for what it takes to be a successful engineer.

• **Dr. John Weber:** For serving on my thesis committee.

• **Kerry Hill, Al Scarpelli, and the Air Force Research Laboratory at Wright-Patterson Air Force Base:** For providing the funding and support which has made this research and my education possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

The use of digital technology in everyday life and specialized applications continues to increase with each passing year. In particular, the use of digital imagery continues to grow as the technology to create this media becomes cheaper and more accessible. In addition, technological advances in many fields, such as robotics, medicine and defense, continue to increase the need for digital imagery. As more applications demand higher resolution imagery and rely on it for critical information, the need to store this data efficiently while maintaining data integrity is paramount. To illustrate this need, consider a standard smart phone featuring an 8 megapixel camera. If an image is taken in monochrome with 8 bits of precision for each pixel, this requires 8 MB of memory prior to compression. While this does not seem like much, considering hard drives are available with terrabytes of storage space, consider a video sequence captured using the same camera at a rate of 30 frames per second. This requires a total of 240 MB of space per second and over 14 GB of space for a single minute. Clearly, storing images in a raw format is not feasible, especially when considering applications which use color images or higher resolution cameras.

1.1 Image Compression

Image compression is the answer to the problem posed by high resolution digital imagery. Image compression is the process of manipulating an image with the goal of reducing the number of bits required to represent a digital signal while it resides in storage or is being transmitted. This process can be thought of as reducing the amount of redundancy inherent within the data [1]. Generally, the manipulation can be performed in two different ways: a lossless approach which allows the original data to be recovered exactly or a lossy approach where the decompressed data is distorted when compared to the input data. While lossless compression offers the ability to fully recover the input data, lossy compression is capable of achieving much higher compression ratios with the tradeoff being an increased amount of distortion for higher compression ratios.

Typical data compression schemes follow a three step process which is illustrated in Figure 1.1.

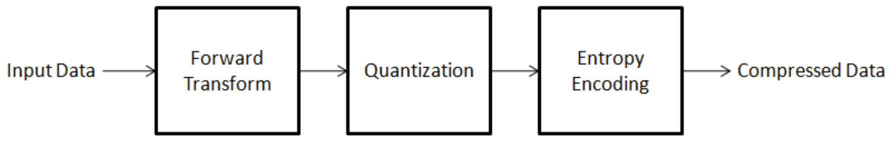


Figure 1.1: Principle Stages of a Generic Compression System

The forward transform is responsible for reducing the redundancy withing the input signal. In the case of natural imagery, neighboring pixels are generally highly correlated with one another. The forward transform decouples the pixel data in the spatial domain, potentially resulting in a more compact representation of the input data [1, 10]. In the case of lossless compression, this step is completely reversible, meaning that the input image can be completely recovered from the compressed image. Certain image compression techniques, such as JPEG, transform the input data from the spatial domain to the frequency domain. In the case of JPEG2000, the DWT results in a representation of the input data in both the frequency and spatial domains. Transforming an image to the frequency domain allows the system to throw away data without and impact on the perceived visual quality of the decompressed image.

The second stage, quantization, is where most of the compression gains are made throughout the compression process. Quantization throws away data in order to represent the original data with fewer number of bits. Since this process eliminates data from the input, it is an irreversible process which is not applied in lossless applications such as medical imagery. Generally, this stage is applied in some fashion, with the amount of quantization dependant on the specific application and its resilience to distortion.

Once the data has been transformed and optionally quantized, the final step is to encode the data. Encoding is a process in which the input data is mapped to a series of symbols which are subsequently encoded. Symbols are encoded by mapping them to codewords based on the probability of occurance of a given symbol. The codeword length can be either fixed or variable length, depending on the encoding method used. Variable-length coding assigns shorter length codewords to symbols which occur more frequently. The average number of bits required to encode a group of symbols is determined as the entropy of those symbols. The entropy of a set of symbols can be found using Equation 1.1,

$$[!h]E = -\sum_{i=1}^{N} p(a_i) log_2 p(a_i), \tag{1.1}$$

where $p(a_i)$ is the probability of occurance of a particular symbol, a_i, and N is the total number of symbols. The probability of occurance can be obtained from the source data but is generally obtained from a model. The final encoding stage helps to reduce redundancies in the data and is a lossless procedure [1].

Once the encoding process has been completed, the decompressed image can be obtained by simply inverting the compression process. In the case of lossless compression, it is possible to recover the original image exactly. However, in the lossy case, data is lost during the quantization process and can never be fully recovered. The decompression process attempts to estimate this lost data, but the resulting decompressed image is simply an estimate of the original image.

1.2 Thesis Objective and Organization

The objective of this thesis is to realize a fully embedded JPEG2000 encoder by designing and implementing an embedded soft-core processing system to perform JPEG2000 Tier II processing on an FPGA. The processing system is coupled with an existing DWT and JPEG2000 Tier I encoder to form a fully embedded JPEG2000 encoder. The existing encoding modules (DWT and Tier I) are the result of an ongoing collaboration between the Air Force Research Laboratory (AFRL) and the University of Dayton Research Institute (UDRI). The embedded soft-core processor chosen for this application is Altera's NIOS II core. Since the NIOS II processor features an reduced instruction set, it will undoubtely perform Tier II processing slower than a traditional consumer off the shelf processor. Therefore, optimizations are made to increase the throughput of the system. The benefits of offloading this processing to the FPGA to realize a fully embedded encoder outweigh a slight increase in system latency.

Chapter 2 of this thesis provides an overview of the principle components of the JPEG2000 compression algorithm. Chapter 3 provides a brief overview of Altera's NIOS II processing core and Chapter 4 describes the design and implementation of a NIOS II system for JPEG2000 Tier II processing. Chapter 5 presents the results of the implemented system and the optimizations applied to the system to increase throughput. Chapter 6 presents a pipeined version of the implemented system. Finally, Chapter 7 presents the conclusions drawn from the work presented in this thesis, while Chapter 8 provides some proposals for future work which could increase the throughput of the design.

CHAPTER 2

THE JPEG2000 COMPRESSION STANDARD

Since its adoption as an international standard in 1992, JPEG for still image compression has been very successful in the global marketplace. This is due in large part to the advent of the Internet and its penetration of aspects of everyday life. Additionally, advances in communications and multimedia technology have helped to compound the growing demand for efficient encoding of data. While JPEG has been successful, studies have shown that 90% of users simply use baseline JPEG while the rest of the standard is left unused [1]. This, combined with the need for better compression while retaining visual quality, led to the development of the JPEG2000 standard. JPEG2000 offers superior visual quality at equivalent high compression rations when compared to JPEG (Figure 2.1) while also offering a wide array of additional features over JPEG. The use of superior transformation and encoding techniques used by JPEG2000 are responsible for the improved image quality. An overview of the JPEG2000 compression process is shown in Figure 2.2. The following section describes what is known simply as baseline JPEG2000, corresponding to Part I of the JPEG2000 standard.

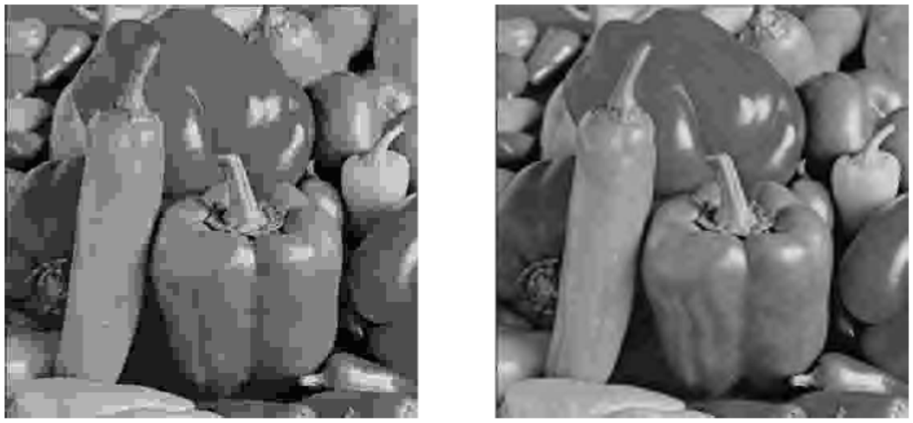


Figure 2.1: Distortion of compressed image at 0.1 bpp using JPEG (left) and JPEG2000 (right)

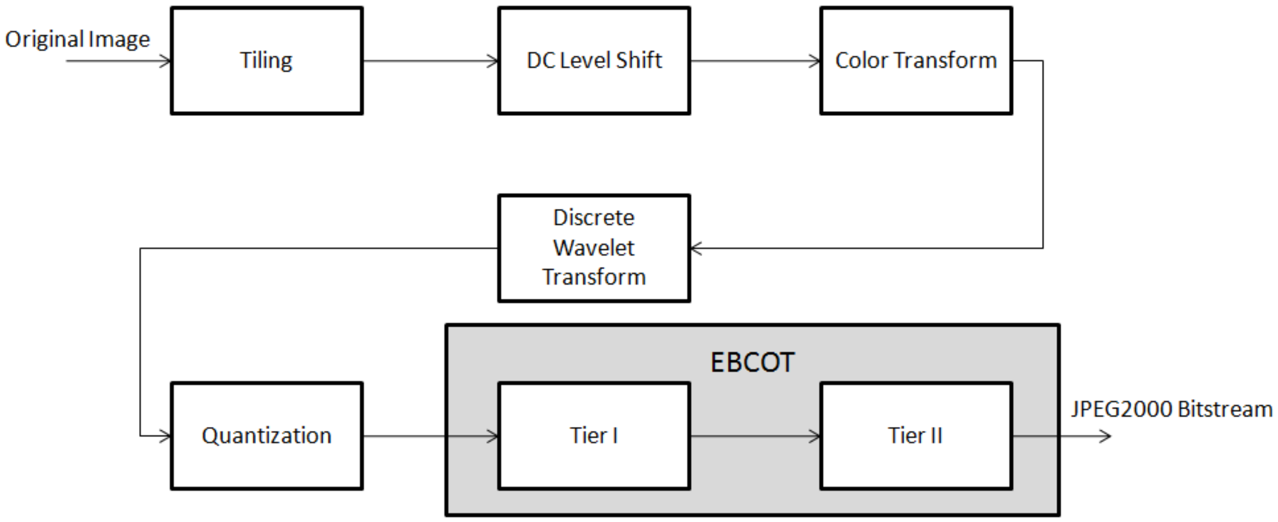


Figure 2.2: Block diagram of JPEG2000 compression standard

2.1 Tiling

Prior to compression, a number of optional preprocessing steps are performed, the first of which is image tiling. This step takes the input source image and partitions it into a number of rectangular nonoverlapping blocks, each of which is called a tile. If the image consists of multiple components, each component is tiled seperately. All tiles are the same dimensions, except those tiles which lie on the boundary of an image whose dimensions are not integer multiples of the tile size. The tile size is an arbitrary number which can be as large as the image itself, typically in the range of 256x256 to 1024x1024. Each tile is then compressed independently. Since the tiles are compressed independently, boundary artifacts may become visible due to heavy quantization at very low bit-rates (in the case of lossy compression). Tiling becomes necessary in applications where memory costs are a chief concern, such as an embedded system, since only a single tile must be stored in memory at any time as opposed to the entire image. Tiling is also an excellent way to facilitate parallel processing, as each tile is able to be compressed independently. Further information regarding tiling, specifically details regarding partial tiles and the JPEG2000 image canvas can be found in [11] and [15].

2.2 DC Level Shift

Following tiling, image components have an optional DC level shift applied, also refered to as a level offset. Since pixel values are stored as unsigned integers, the DC level shift is used to shift pixel values so that they are distributed around zero. This is accomplished by subtracting 2^{B_i-1} from each pixel value in each component, where B_i is the bit-depth of the ith component. Equation 2.1 shows the mapping of an image pixel $I(x, y)$ to a new DC shifted value.

$$I_{DCShifted}(x, y) \leftarrow I(x, y) - 2^{B_i - 1} \tag{2.1}$$

The DC level shift yields a range of pixel values centered at zero and this range can be seen in Equation 2.2.

$$-2^{B_i - 1} < I_{DCShifted}(x, y) < 2^{B_i - 1} \tag{2.2}$$

The DC level offset is used since the majority of samples produced by the DWT are the result of high-pass filtering. Since high-pass filtering yields values centered around zero, the samples produced only by low-pass filtering would become an exception as they would not be centered around zero. While this step is considered optional, using unsigned samples could result in a bit-stream which is incompatible with certain decoder implementations [15].

2.3 Color Transform

The color transform is optionally applied after the DC level offset. The color transform reduces any correlations between the components of a multicomponent image, resulting in increased compression performance due to a decrease in redundancy. The color transform is optionally applied to the first three image components which can be interpreted as three color planes (R, G, B). While these components do not have to represent RGB data, they must be the same bit-depth and have the same dimensions to apply the color transform. The color transform produces a luminance and two chrominance components, denoted YCbCr.

The standard defines two color transforms: a reversible color transform (RCT) for lossless compression and an irreversible color transform (ICT) for lossy compression.

The RCT is used in conjunction with the lossless 5/3 wavelet for the DWT [11]. Equation 2.3 gives the forward RCT, while Equation 2.4 gives the inverse RCT.

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.25 & 0.5 & 0.25 \\ 0.00 & -1.0 & 1.00 \\ 1.00 & -1.0 & 0.00 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{2.3}$$

$$\begin{gathered} G = Y - .25Cb - .25Cr \\ \begin{pmatrix} R \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 & 1.0 \\ 1.0 & 1.0 & 0.0 \end{pmatrix} \begin{pmatrix} G \\ Cb \\ Cr \end{pmatrix} \end{gathered} \tag{2.4}$$

For lossy compression, the ICT is used to convert RGB to YCbCr color space and only in conjunction with the DWT's 9/7 wavelet [11]. The forward ICT is shown in Equation 2.5 and uses non-integer coefficients, leading to rounding errors making the transform irreversible. The reverse ICT is given by Equation 2.6.

$$\begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} = \begin{pmatrix} 0.29900 & 0.58700 & 0.11400 \\ -0.16875 & -0.33126 & 0.50000 \\ 0.50000 & -0.41869 & -0.08131 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{2.5}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.00000 & 1.40200 \\ 1.0 & -0.34413 & -0.71414 \\ 1.0 & 1.77200 & 0.00000 \end{pmatrix} \begin{pmatrix} Y \\ Cb \\ Cr \end{pmatrix} \tag{2.6}$$

2.4 Discrete Wavelet Transform

To achieve data decorrelation, JPEG2000 uses the Discrete Wavelet Transform (DWT). By high and low pass filtering an image in both the horizontal and vertical dimensions, the DWT decomposes an image into different spatial frequency subbands and resolutions. Figure 2.3 shows the DWT process for a single decomposition level, which actually contains two resolution levels. The original image (left) is first low pass filtered along the rows and then downsampled by a factor of two along the columns. This process is then repeated with a high pass filter, yielding two

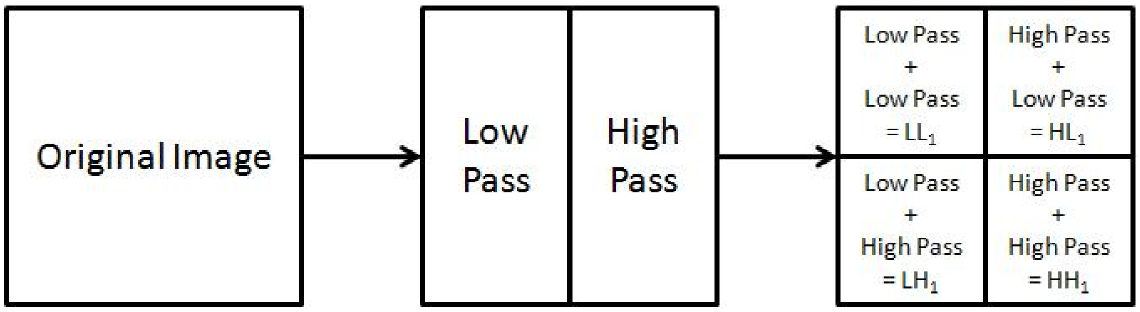

Original Image
Low Pass
High Pass
Low Pass + Low Pass = LL1
High Pass + Low Pass = HL1
Low Pass + High Pass = LH1
High Pass + High Pass = HH1

Figure 2.3: Original Image (left); Row processed and downsampled (center); Column processed and downsampled image (right)

image halves, shown in the center of Figure 2.3. Then, each image half is low pass filtered along the columns and downsampled by two along the rows. The same process is performed with a high pass filter. Finally, the images are merged into a single image, yielding the four subbands pictured in the right of Figure 2.3. Each of the four quadrants of the resulting image are refered to as different frequency subbands. The LL_1 subband is a coarse approximation of the original image and considered to be a low resolution representation of the original image. The other 3 subbands (HL_1, LH_1, and HH_1) contain the residual information which, when combined with the LL_1 subband, form the original image. For this reason, the DWT is considered a multi-resolution transform and can contain an arbitrary number of decomposition levels. To obtain subsequent decompositions, the DWT is applied to the LL_1 subband produced in the first decomposition. This process is repeated for each level. Figure 2.4 shows an example of an image transformed with three decomposition levels. A DWT'd image will yield N_L+1 resolution levels and $3N_L+1$ subbands, where N_L is the number of decomposition levels specified. For natural images, 5 decomposition levels have been shown to provide the best compression performance [16].

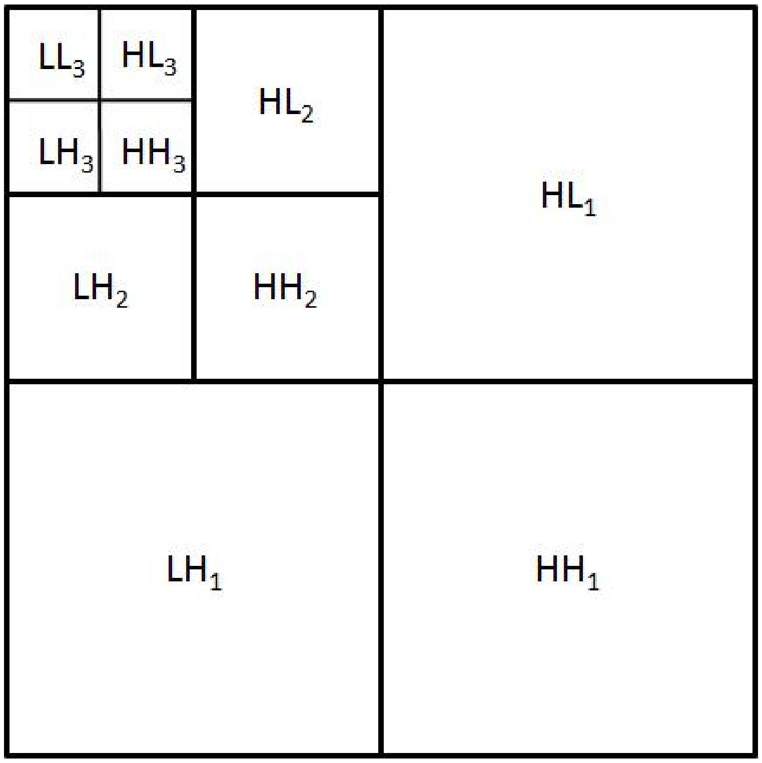


Figure 2.4: Example of DWT with three decomposition levels

Depending on whether lossless or lossy compression is desired, JPEG2000 employs one of two different wavelet filter types. For lossy compression, the Daubechies 9/7 filter is used while the 5/3 filter coefficients are used for lossless compression. The numbers used to refer to each set of filters refers to the number of taps in the low and highpass filters, respectively. The wavelet coefficients used for each type can be found in [2]. Further details regarding the DWT and possible implementations are not included, as they are beyond the scope of this thesis. For more information regarding the DWT, refer to [10, 15].

2.5 Quantization

Once the DWT has been performed, the precision of the resulting subband coefficients are is recuded to decrease the total amount of data prior to entropy encoding [1]. A reduction in data size prior to entropy encoding yields a smaller compressed code-stream. Since quantization discards data, it is an irreversible process and is not performed for lossless compression, allowing for an average compression raio less than 3:1 for lossless compression [8].

Quatization is performed by dividing each of the wavelet coeffecients in a subband by a parameter Δ_b, called the quantization step size, where b refers to the coefficients subband. Equation 2.7 is used to calculate the quantized wavelet coefficient, $q_b(i,j)$.

$$q_b(i,j) = sign(y_b(i,j)) \left\lfloor \frac{|y_b(i,j)|}{\Delta_b} \right\rfloor \tag{2.7}$$

Equation 2.8 is used to calculate Δ_b,

$$\Delta_b = 2^{-\varepsilon_b} \left(1 + \frac{\mu_b}{2^{11}}\right) \tag{2.8}$$

where μ_b and ε_b are the mantissa and exponent for subband b, respectively. Both μ_b and ε_b are non-negative integers in the ranges of $0 \leq \mu_b < 2^{11}$ and $0 \leq \varepsilon_b < 2^5$ [15]. These parameters can be specified for each subband, termed expounded quantization, or for only the LL_{N_L} subband, called derived quantization. More information regarding quantization can be found in [1, 11, 15].

2.6 Embedded Block Coding With Optimized Truncation

Following quantization, the wavelet coefficients are entropy encoded, which is a lossless process. The algorithm used by JPEG2000 for entropy encoding is refered to as Embedded Block Coding with Optimized Truncation (EBCOT) [15]. EBCOT is a very computationally expensive task, resulting in JPEG2000 being around 30 times more expensive than baseline JPEG [1]. EBCOT is broken into two stages: Tier I and Tier II. Tier I is the most intensive portion of EBCOT and is responsible for encoding the wavelet coefficients. Tier II is left to assemble the code-stream from the encoded data as well as optimized truncation of the bit stream to meet specified file sizes.

2.6.1 Tier I

As mentioned, Tier I is responsible for entropy encoding the wavelet coefficients and is the most computationally expensive portion of EBCOT. Tier I is also broken into two stages: bit plane coding and arithmetic encoding. Prior to Tier I encoding, each wavelet subband is divided into sections of coefficients, refered to as code blocks. Code block do not span across subbands, and if a code block boundary does not line up with a subband boundary, is partial code block is used. Typical code block sizes are 32x32 or 64x64 samples. Details regarding the sizes of code blocks and their boundaries can be found in [11]. Code blocks are encoded independently, meaning that Tier I is a perfect candidate for parallel processing implementation.

Once the wavelet coefficients are divided into code blocks, each code block is then sliced horizontally between bits to create bit planes. A graphical representation of a code block divided into bit planes is shown in Figure 2.5. This results in all the bits for a specific bit level grouped into a single, two-dimensional, array. The bit plane containing the most significant bits of a code block is refered to as the most significant bit-plane.

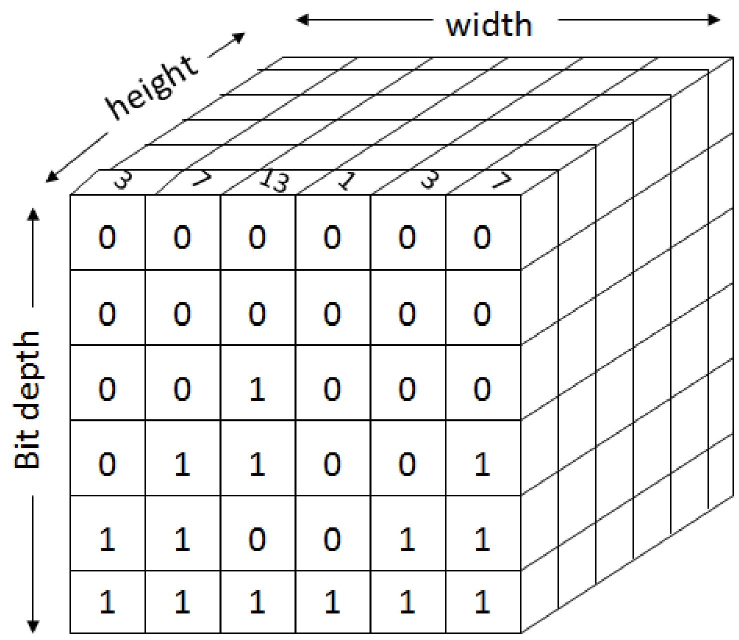


Figure 2.5: Code block divided into 8 bit planes (for a bit depth of 8)

Once divided into code blocks and bit planes, the wavelet coefficients are then converted from twos complement representation to a sign-magnitude representation. Then, beginning with the most significant bit plane, each bit plane is scanned to find the first plane to not contain all zeros. This bit plane is refered to as K and is the first non-all-zero bit plane in the code block. A sequence of three coding passes are then applied to the bit planes, beginning with K and ending with bit plane 1, the least significant bit plane. The three coding passes are: Significance Propagation Pass (SPP), Magnitude Refinement Pass (MRP), and Cleanup Pass (CUP) [15]. These coding passes are applied in the order of SPP, MRP, and CUP. However, only the CUP is applied to bit plane K, as the other coding passes would yield no output [1]. The coding passes generate context labels and decision pairs for each bit as inputs to the arithmetic encoder. Each bit, processed in groups of four called stripes, is only encoded once by a single coding pass.

The coding passes are associated with 3 state tables which track which bits have been coded and how context-decision pairs are assigned to each bit. The three coding passes are: σ, σ_D, and π. The state tables are the same size as the code block and locations within these tables correspond to locations within the code block. σ denotes the significance of a coefficient, meaning whether or not a one bit has been coded for the coefficient, and is set to one when the first one bit of a coefficient is coded. σ_D is called the delayed significance and denotes whether a bit of a coefficient is coded during the MRP. Both σ and σ_D are set to zero after each code block. π refers to the coding pass membership and denotes whether zero coding has been performed for the current bit. Further details of the bit plane coding fall outside the scope of this thesis. Further detail regarding the bit plane coding procedure can be found in [1, 15].

The second portion of Tier I coding is arithmetic encoding of the context-decision pairs generated during the bit plane coding process. The arithmetic encoder used in JPEG2000 is the MQ coder, a binary arithmetic entropy coder, which lossless encodes a set of symbols. The symbols encoded by the MQ coder are decision bits generated during bit plane coding. The context data generated in the bit plane coding is used to determine whether a bit is a more or less probable symbol. These contexts are used too determine whether a one or zero is the more probably symbol and are coded accordingly. More detail on the MQ coder can be found in [1, 15].

2.6.2 Tier II

The final portion of EBCOT, Tier II, is responsible for forming the final JPEG2000 file from the individual compressed code blocks and tiles. Tier II is tasked with efficiently representing layer and block information for each code block. Tier II must represent to which bitstream layers each code block contributes compressed codewords, and the lengths of these codewords. This is known as the inclusion information. Tier II must also represent the zero bit plane information which specifies the first none-all-zero bit plane in the code block. Finally, Tier II must also represent the number of coding passes contributed to each bitstream layer, known as truncation points.

The inclusion and zero bit plane information for a code block is encoded using two Tag Trees. A Tag Tree is a specific type of quad-tree data structure and provides a way to represent a two-dimensional of non-negative integers. An example of the Tag Tree data structure for a 6x3 array is shown in Figure 2.6. The 6x3 array is reduced to smaller resolution levels, from 3 down to 0. The elements in resolution n are formed from the minimum of each 2x2 subarray of resolution $n+1$. The 2x2 subarrays are denoted by dotted lines. The structure shown in Figure 2.6 can be thought of as a tree data structure, with resolution 0 treated as the root of the tree. The leaf nodes of the tree contain the original two-dimensional array elements and the internal nodes represent the intermediary resolution levels. Tag Tree coding follows simply from the structure shown in Figure 2.6. Each node in the tree is encoded by d 0's follow by a 1, where d is the difference between a node and its parent. The root node is assumed to have a zero valued parent node. For a more in-depth example of Tag Tree coding, refer to [1].

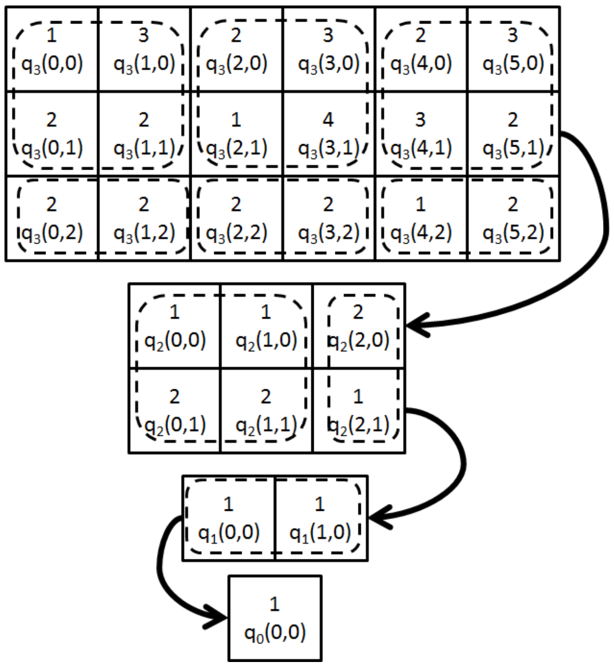


Figure 2.6: Example of Tag Tree data structure (taken from [1])

The Tag Tree structure defined is used to encode the packet header information. A packet is compressed data containing a specific component, tile, layer, resolution and precinct. A layer contains a number of consecutive bit plane coding passes from each code block in a tile. The bitstream of each code block may be spread across multiple layers. A precinct is a partition in each DWT resolution. The contiguous segment containing the compressed bitstreams for a specific tile, layer, resolution, component and precinct is called a packet. The header information for a packet is contained in the bitstream just prior to the packet itself. The packet header contains: a bit indicating whether the packet is empty, inclusion information (Tag Tree), number of leading zero bit-planes (another Tag Tree), the number of coding passes for each code block in the packet (encoded with lookup table) and the length of the bitstream for each code block.

The JPEG2000 standard allows for five different progression orders, or ways to order the information within a packet [11]. The progression order defines the order in which data is encoded in the bitstream, and therefore the order in which the data is decoded as well. Different progression orders are useful for different applications, as some progress in quality while others progress in quality. The default progression order specified in the standard is layer-resolution-component-position progressive [11]. Other progressions are: resolution-layer-component-position, resolution-position-component-layer, position-component-resolution-layer, and component-position-resolution-layer. Much of the detail of Tier II has been omitted from this thesis, as the Tier II code is borrow for this implementation. For more detai regarding the bitstream format, bitstream markers, or progression orders, refer to [1, 11, 15].

CHAPTER 3

ALTERA'S NIOS II PROCESSING CORE

Due to the serial nature of JPEG2000 Tier II processing, an embedded processing core is selected to implement this algorithm. Since the existing embedded DWT and Tier I implementation is targeted for an Altera FPGA, an Altera embedded processing core is selected to implement the Tier II algorithm. The NIOS II soft-core processing core from Altera is selected as the embedded processor of choice since it can be implemented on an Altera FPGA. In addition, Altera offers a fully IDE for developing software applications to run on the NIOS II core. The NIOS supports native C/C++. A generic block diagram of a system leveraging a NIOS II core can be seen in Figure 3.1.

The NIOS II soft-core processing core from Altera is a 32-bit processor which utilizes the reduced instruction set computing (RISC) design strategy [6]. The NIOS II processor is termed 'soft' because it is entirely implemented using logic synthesis on an FPGA, unlike traditional 'hard' processors which are composed of dedicated logic units. Both implementations have advantages and disadvantages. Since a hard processor is composed of dedicated hardware devices, these cores tend to be faster and highly optimized since the hardware is designed specifically for that processor. This level of optimization allows the core to support higher clock rates and ultimately achieve higher throughput. However, since these cores are built on

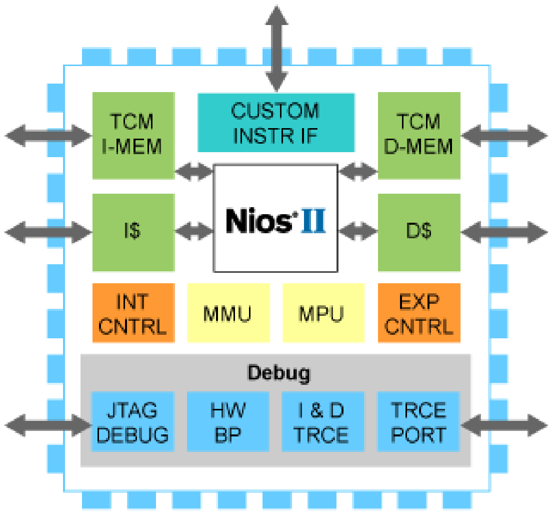


Figure 3.1: Block diagram of a generic NIOS II system

dedicated hardware, they are unable to be changed to meet the needs of the specific application.

On the other hand, a 'soft' processing core is entirely implemented using logic synthesis, most comonly on an FPGA. Therefore, soft processing cores are capable of being reconfigured easily since they are not built on dedicated hardware. Therefore the designer is free to add or remove peripherals to and from the system depending on the application. This results in processing systems which are highly optimized for their application, as all unnecessary peripherals are removed from the system in the intreset of saving resources. While the soft nature of these processing cores results in slightly reduced throughput compared to hard processing cores, the added flexibility of the soft processing core makes it an ideal choice for a wide array of embedded applications.

3.1 NIOS II Processing Core Specifications

As previously mentioned, Altera's NIOS II processing core is a 32-bit RISC processor offering a full 32-bit instruction set, data path and address space. The NIOS II core offers 32 general-purpose registers as well as optional shadow register sets. It can support up to 32 interrupt sources and features an external interrupt controller interface to support additional interrupt sources. The core features single-instruction 32x32 multiply and divide as well as dedicated instructions for computation of 64-bit and 128-bit products of multiplication. Custom instructions can be added to the core to provide additional features and Altera offers floating-point instructions for single-precision floating-point operations. Additional options include: single-instruction barrel shifter, memory management and memory protection units and hardware assisted debug modules [6].

Altera offers three different NIOS II processing cores: economy (/e), standard (/s) and fast (/f). The differences between the three cores can be seen in Table 3.1 (max clock frequencies are for Stratix III devices).

Table 3.1: Available NIOS II Processing Cores

$IPCore$	$LogicUsage$	$MemoryUsage$	$MaxClockFrequency$
$NIOSII/e$	$600-700LEs$	2 M9Ks	340 MHz
$NIOSII/s$	$1200-1400LEs$	2 M9Ks + Cache	230 MHz
$NIOSII/f$	$1400-1800LEs$	3 M9Ks + Cache	290 MHz

The economy core is basic NIOS II core and is highly optimized to provided the lowest power and cost of the three cores. The standard core is the same as the economy core, but supports additional features such as: a dedicated instruction cache, branch prediction and dedicated hardware for multiply and divide instructions. The standard core is a balance between cost and performance. The fast core offers more features over the standard core and is designed to provide the highest performance of the three cores. The fast core supports additional features such as optional memory management and memory protection units, an external interrupt controller, dedicated instruction and data caches and dynamic branch prediction to name a few. For more information regarding the NIOS II processing core please refer to the NIOS II Reference Handbook [6].

3.2 SOPC Builder: NIOS II Processing Systems

The advantage of using soft processing cores such as the NIOS II core are their ability to be easily reconfigured to support a wide variety of peripherals, both on and off chip. When refering to systems leveraging NIOS II processing cores, generally one refers to a NIOS II processing system. Not only does this include the NIOS II core, but any other peripherals attached to the NIOS II core. These peripherals include: on-chip RAM modules, memory controllers for off-chip memory, direct memory access (DMA) controllers, serial interfaces and many more. The NIOS II core is also capable of interfacing with custom made peripherals as well. To design, create and maintain complex systems comprised of processing cores and peripherals, Altera offers the System-on-a-Programmable-Chip (SOPC) builder. A generic SOPC system can be seen in Figure 3.2.

Altera's SOPC Builder is a tool which enables users to create and maintain complex processing systems without dealing with the complexity involved in interfacing all of the various processing cores and peripherals. SOPC Builder allows users to select from a wide array of IP cores, including NIOS II cores, to tailor the system

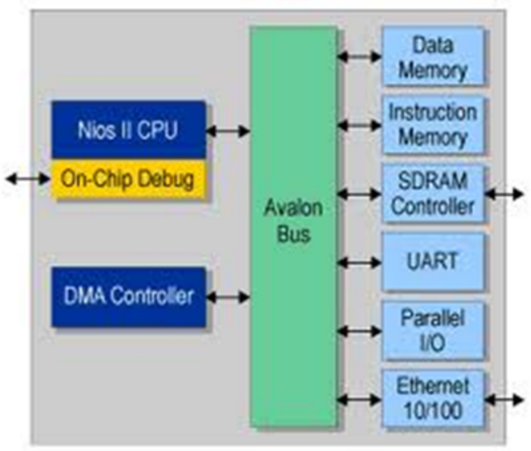


Figure 3.2: Block diagram of a generic SOPC system

to meet the needs of the specific application [3]. Communication between multiple components is handled by the Avalon Bus, a master/slave architecture allowing simultaneous communication between different sets of masters and slaves [4]. SOPC Builder stream-lines the process of integrating multiple IP cores by handling bus arbitration, width matching and clock domain crossing automatically. This results in a heterogenous system composed of multiple IP cores, each of which may be operating in its own clock domain [3].

3.3 Avalon Interface Concepts

Interconnection between multiple different IP cores in SOPC systems is facilitated by the Avalon bus. The Avalon bus is a master/slave architecture which allows multiple master/slave pairs to communicate simmultaneously across the bus. All bus arbitration, width matching and clock domain crossing is hidden from the user and handled by SOPC Builder. Altera has defined an Avalon interface specification which defines interfaces appropriate for multiple different applications. All of the components available in Altera's SOPC Builder leverage Avalon interfaces to communicate over the Avalon bus.

Three different Avalon interfaces are addressed here: Memory Mapped, Streaming, and Interrupt interfaces. The Avalon Memory Mapped (MM) interface is an address-based read/write interface. These interfaces are typical of master/slave connections and are used by many IP cores such as memory controllers and parallel input/output (PIO) devices. The Avalon Streaming (ST) interface is designed for unidirectional dataflow. Data flows from an Avalon-ST source to a Avalon-ST sink and are idea for data steaming, packets and DSP data. The Avalon Interrupt interface defines an interface which allows components to signal events in other components. This interface is essential in event-driven systems and is ideal for implementing non-blocking functionality [4].

CHAPTER 4

JPEG2000 - EBCOT TIER II IMPLEMENTATION

In order to realize a fully embedded JPEG2000 encoder, existing DWT and EBCOT Tier I modules are coupled with a newly designed Tier II processing module. As previously mentioned, the existing encoding modules (DWT and Tier I) are the result of an ongoing collaboration between AFRL and UDRI. The Tier II processing module leverages a NIOS II processing system which features an embedded NIOS II core. The NIOS II system interfaces directly with the existing encoder modules to receive the necessary data to perform Tier II. The software code used to perform the Tier II processing is written in C/C++ [12, 14] and is adapted for embedded use for this application.

4.1 Target Platform

The existing DWT and EBCOT Tier I modules are designed and implemented on FPGA cards from GiDEL, an Israeli company specializing in high-performance FPGA solutions [13]. The existing modules have been implemented on a number of GiDEL FPGA cards which leverage multiple Altera Straix III or Stratix IV FPGAs . For this application, GiDEL's PROCeIV 530 card is selected as the target platform. A picture of a PROCeIV is given in Figure 4.1. The PROCeIV 530 features a single

Stratix IV E 530 FPGA an communicates with the host processor via 4-lane PCIe (x4). The Stratix IV E 530 FPGA features over 530,000 logic elements.

Figure 4.1: PROCeIV FPGA Card from GiDEL

As mentioned, the PROCeIV communicates with the host processor via PCIe x4. The PROCeIV supports user designs running at clock rates up to 325 MHz. This card features 512 MB of on-board DDR2 memory as well as 2 DDR2 SODIMMs for additional off-chip memory. Each SODIMM supports an additional 4 GB of memory each. A block diagram of the PROCeIV architecture is given in Figure 4.2. The block diagram shows that each of the three memory modules are designated as seperate banks: Bank A refers to the 512 MB on-board DDR2 RAM module while Banks B and C refer to the two DDR2 SODIMMs [9]. These names are used throughout the rest of this document to refer to the memory modules.

In addition to providing high performance hardware in the form of the PROCeIV, GiDEL also provides software development tools for implementing designs on their boards. Additionally, GiDEL offers software APIs for sending and receiving data to and from the board in addition to communicating with the board. Host processors may send and receive data to and from the board using DMA over PCIe. Details on

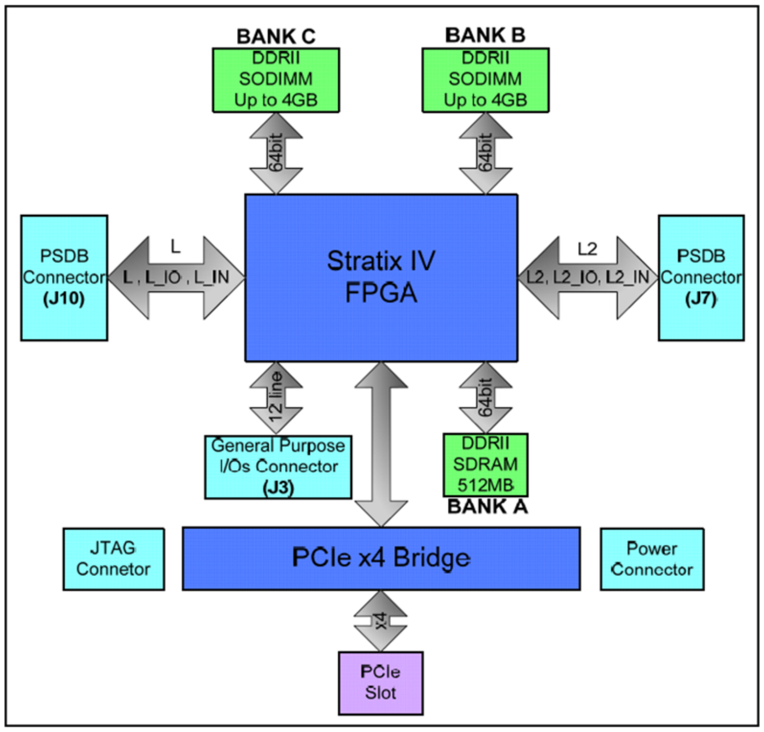


Figure 4.2: Architectural block diagram of PROCeIV

this interface are provided in 4.1.1. In addition, the host processor may communicate with the FPGA board by reading and writing to registers on the board through GiDEL API calls.

4.1.1 GiDEL MegaFIFO

As mentioned in this section, GiDEL provides software APIs for sending and receiving data to and from their FPGA boards of PCIe. One way to send and receive large amounts of data is to use DMA via PCIe. This facility is provided by GiDEL's MegaFIFO IP core which simplifies the process of transfering data to and from the board. The MegaFIFO IP core provides high-speed access to the various DDR2 memory banks on the board from both the host processor and user subdesigned on the FPGA. MegaFIFOs, as the name suggests, partition the memory banks into large FIFOs which can have multiple read/write ports depending on their configuration. MegaFIFOs can be configured to transfer data between the host processor and user subdesigns or between user subdesigns as large FIFOs. Due to the multiple configuration possibilties, the MegaFIFO is an integral piece to this design. MegaFIFOs are used to transfer raw data to the board and compressed data to the host as well as transfering data between stages in the encoding pipeline. All MegaFIFOs used in this design are configured to have a single read/write port.

4.2 Top Level Modules and Data Flow

At the top level, the initial embedded Tier II implementation features four distinct modules. A block diagram showing the four modules and their interconnections is given in Figure 4.3. Two of the modules at the top level are the JPEG2000 encoding modules: the existing DWT and Tier I modules (combined into a single module at the top level) and the Tier II processing module (created for this design). The other two modules in Figure 4.3 are GiDEL interface modules. The module entitled "Interface Module" handles the configuration of the available clocks as well as communication with host processor via reading/writing of registers on the board. This is why feedback and control signals connect the GiDEL Interface Module to both the DWT & Tier I module as well as the Tier II module. The other GiDEL module labeled "Bank A Control Module" handles control of the MegaFIFOs located in Bank A on-board memory, the details of which will be discussed when covering the data flow through the top level modules.

Data flow through the system follows a linear path, given in Figure 4.4. This figure shows a general data flow through the system and does not specify where the data is stored in the system, which will be covered next. Since the existing

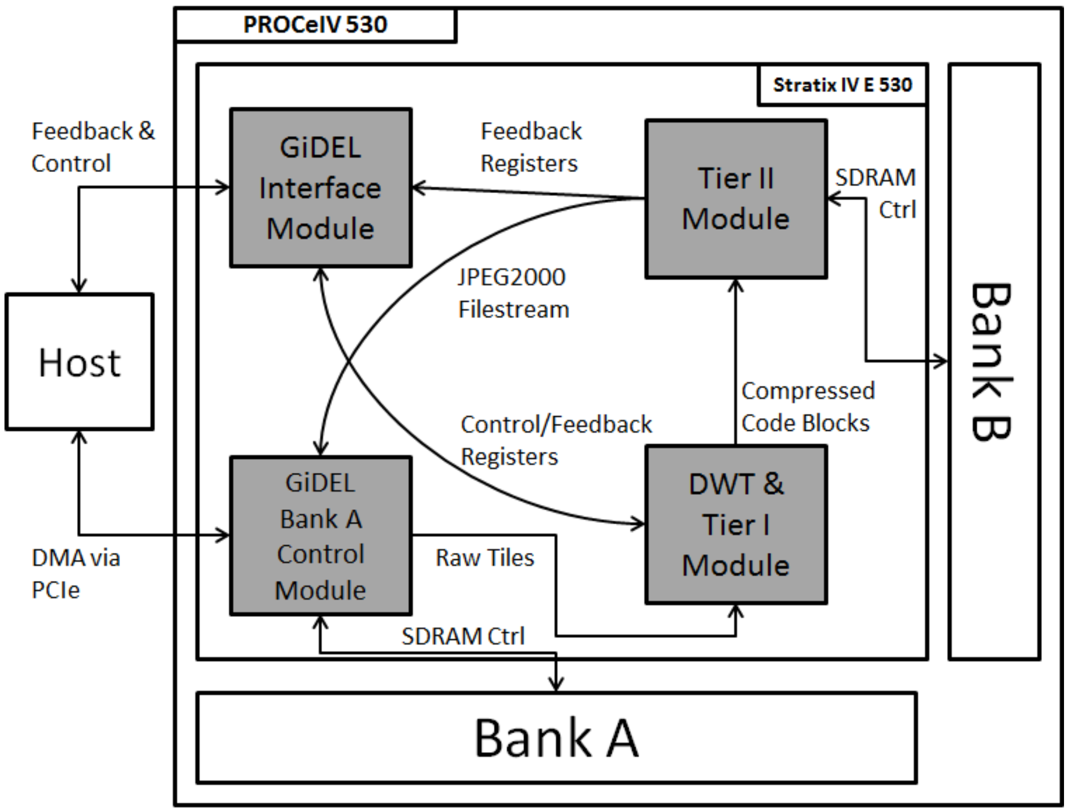


Figure 4.3: Top level block diagram

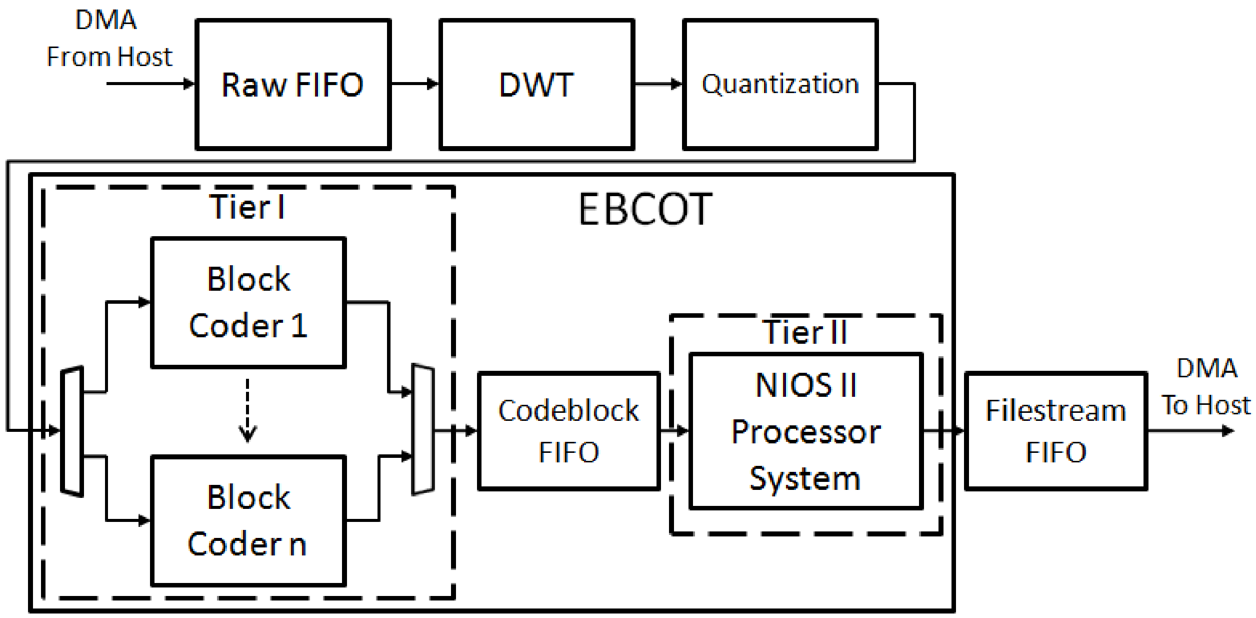


Figure 4.4: Top level data flow of design

DWT and Tier I modules operate on a tile basis, as in they only process a single tile at a time, the data flow through the system is described for a single tile. The DWT module receives tiles from the host, stored in a raw data FIFO and produces the coefficients which make up each of the subbands for each resolution level of the DWT. Each of these subbands are then quantized and partitioned into 32x32 code blocks prior to being sent along to Tier I. Note that Tier I is comprised of multiple block coders which encode code blocks in parallel. The compressed code blocks produced by the block coders are then muxed and placed into a FIFO. The Tier II module then reads code blocks from Tier I and, once 1024 code blocks have been received, performs the Tier II encoding and places the final bitstream for that tile into an output FIFO. These tiles are then received by the host and aggregated to form a full JPEG2000 filestream.

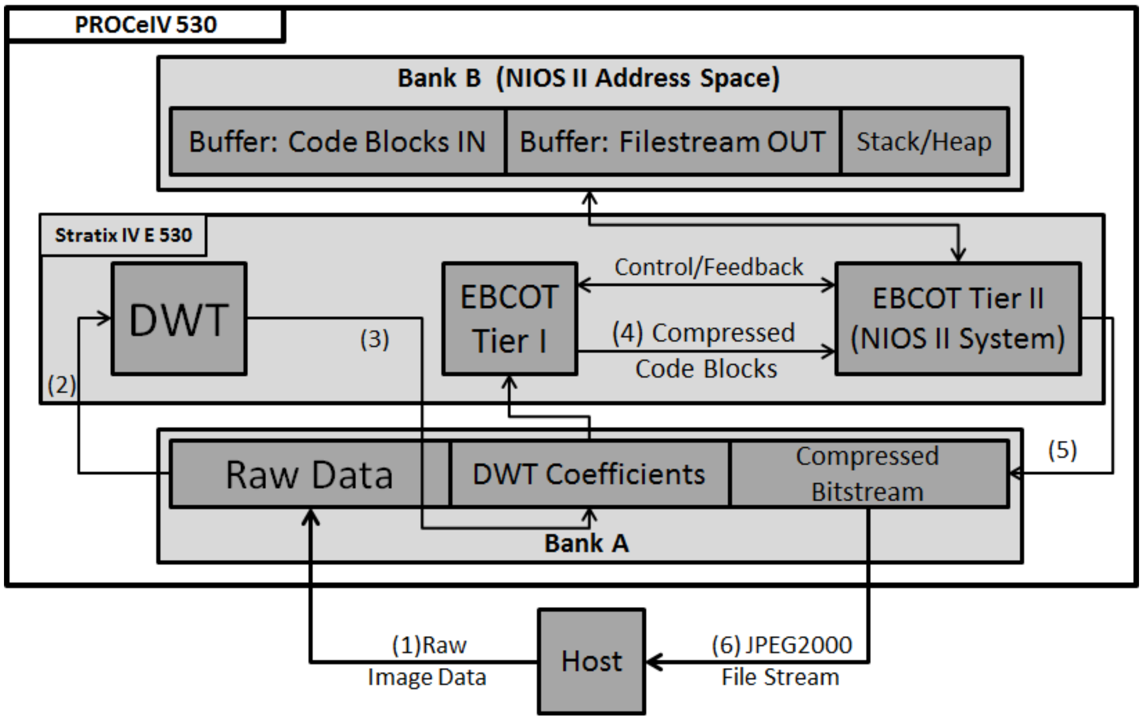


Figure 4.5: Architectural block diagram of PROCeIV

Figure 4.5 shows a more detailed description of the data flow through the system. This figure shows that Bank A is partitioned into three MegaFIFOs: one to receive raw data from the host, one to store DWT coefficients produced by the DWT and one to store the compressed bitstream for DMA back to the host. Note that the Tier II module communicates directly with the output of the Tier I module. This is important to note when discussing the performance of the system. Bank B is dedicated to the NIOS II system and stores two large buffers: one for buffering compressed code blocks received from Tier I, and one for buffering the compressed filestream for the current tile prior to output to the MegaFIFO.

4.3 Existing DWT and Tier I Modules

At the top level (Figure 4.3), the DWT and Tier I are coupled together into a single module. Since this thesis is concerned with the implementation of a NIOS II system for Tier II processing, the details of these existing modules will be spared, focusing instead on the input and output specifications of the modules. However, it should be noted that the DWT module runs at a clock rate of 100 MHz while the Tier I block coders are clocked at 200 MHz.

The DWT receives a single tile at a time from the Raw Data MegaFIFO. Once a full tile has been read, the DWT is performed and the resulting subband coefficients are placed into a MegaFIFO. A MegaFIFO is used between the DWT and Tier I to provide a high speed buffer between the two modules running at different clocks. It also enables the DWT to continue processing without worrying whether Tier I has begun processing the subbands or not.

The Tier I module receives the subband coefficients from the MegaFIFO as they become available. Prior to partitioning each subband into code blocks, each subband is quantized seperately based on quantization step sizes received from the host. Once quantized, each subband is partitioned into 32x32 code blocks. Since Tier I features a variable number of block coders, the code blocks produced by the DWT are processed in parallel by Tier I. Each block coder has a FIFO at the back end to store a single compressed code block. The control signals for these output FIFOs are fed through a mux which cycles through the block coders to expose their output data. Note that a block coder will not begin processing the next code block until its output FIFO is empty.

4.4 Tier II Module

The Tier II module is basically a wrapper for the NIOS II system which performs Tier II processing. The Tier II module is responsible for registration of various control signals and also converts the endianness of data sent from the NIOS II system back to the host via MegaFIFO. In addition, this module has a state machine to control the various signals which couple the NIOS II system with the output FIFO from Tier I. This state machine is necessary for two reasons, both of which are related to unexpected behavior of the DMA controllers used in this NIOS II system. This behavior necessitating this state machine and the operation of the state machine are explained in later subsections.

4.4.1 SOPC System

All of the Tier II processing performed in this module is done by a NIOS II processing core. This processing core is coupled with mutliple IP cores using SOPC Builder to form a full NIOS II system. A block diagram of the SOPC system designed for Tier II processing can be seen in Figure 4.6. The SOPC Builder system implemented makes use of Avalon-MM, Avalon-ST and Avalon Interrupt interfaces. There are three clocks available in the system: 290 MHz, 200 MHz and 100 MHz. The 290 MHz clock is generated from a PLL which takes the standard 125 MHz clock available on the PROCeIV. The 100 MHz and 200 MHz clocks are passed in from another PLL which generates these clocks for use by the DWT and Tier I modules.

The NIOS II fast core is selected for this system as it provided the highest throughput available of the three processing cores. No advanced features are added to the processing core except for the optional hardware divide. The instruction and

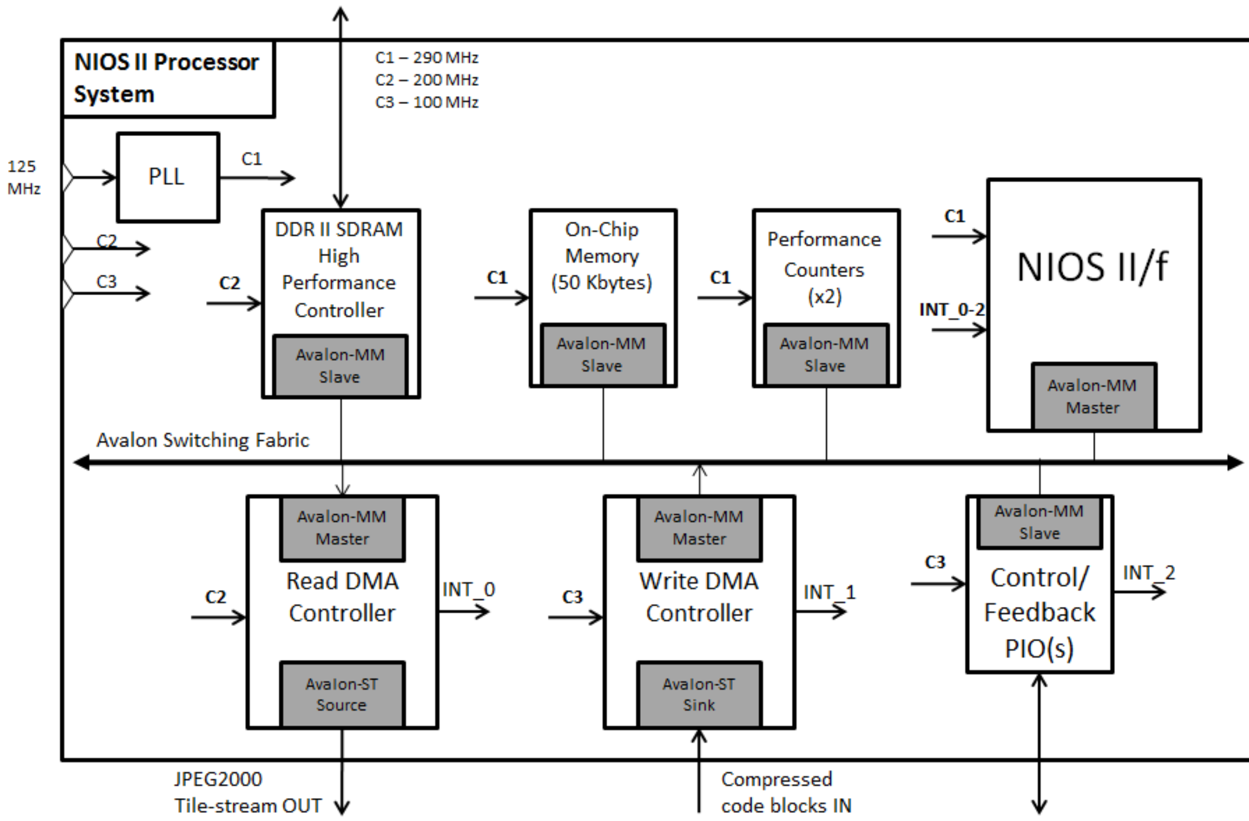


Figure 4.6: SOPC System design to perform Tier II processing

data caches are configured as 4 kBytes and 8 kBytes, respectively. The NIOS II/f core is clocked using the 290 MHz clock.

The NIOS II system features two seperate memory controllers. One is a 50 kByte on-chip memory module, configured as RAM, with a 32-bit data width. The other memory controller is the DDR2 SDRAM High Performance Controller. The DDR2 memory controller is used to control Bank B as has access to all 4 GB made available by the SODIMM. The on-chip memory runs off of the 290 MHz clock while the DDR2 controller runs at 200 MHz.

The system also features two DMA controllers: one for reading from Tier I and one for writing to the output MegaFIFO. The names of the two DMA controllers are counterintuitive, as the "Read DMA Controller" actually writes data to the output FIFO while the "Write DMA Controller" reads data from the Tier I output FIFO. The naming convention used is relative to the memory device master by both DMA controllers (DDR2), as the read DMA reads from SDRAM and the write DMA writes to SDRAM. The names are based on the configuration of the DMA controllers, which is elaborated on in the following section. The "Read DMA Controller" is clocked at 200 MHz while the "Write DMA Controller" is clocked at 100 MHz. Both DMA controllers have interrupt interfaces which are serviced by the NIOS II core. This allows the DMA controllers to signal to the NIOS when transfers are completed.

Two performance counters are also added to the system. These allow for system profiling to benchmark the software. The performance counters are clocked at 290 MHz, as this is the same clock as the NIOS II core. Additionally, there are a number of parallel input/out (PIO) interfaces added to the system. These are used to receive and provide feedback to and from hardware on the board as well as to read and write GiDEL feedback registers to communicate with the host. Most of these are simply used for debugging purposes, except for one which enables Tier I to signal the NIOS II core when a code block is ready. This PIO, refered to as the Codeblock Ready PIO, has an interrupt interface which enables the NIOS II core to service this event in the same fashion as DMA transfers.

4.4.2 Modular SGDMA Controller

The DMA controller used in this implementation is not available from Altera's SOPC Builder and was designed to replace the default DMA controller. This DMA controller is a module scatter-gather DMA (SGDMA) controller and is different than a normal DMA controller. An SGDMA controller differs from a standard

DMA controller in that the SGDMA is capable of handling mutliple transfers in hardware without intervention from a host. This results in less overhead and is ideal for networking, audio and video processing applications. This modular SGDMA controller is chosen over the default DMA in SOPC Builder due to a more user friendly API and modularity.

The modular SGDMA core is composed of three components: a dispatcher, write master and read master. The dispatcher receives commands (called descriptors) from a host (in this case the NIOS II). The dispatcher then interprets the descriptor and sends the appropriate read/write commands to either the read or write master, or both. The read and write masters receive commands from the dispatcher and perform the specified task, sending a response back to the dispatcher upon completion. Once the response has been received, the dispatcher will raise an interrupt on the host, if requested, to signal that the transfer has completed.

The modular SGDMA controller can be configured in one of three different ways: memory-mapped to streaming, streaming to memory-mapped or memory-mapped to memory-mapped. Examples of the streaming to memory-mapped and memory-mapped to streaming configurations can be seen in Figures 4.7 and 4.8. Note that each configuration only includes the necessary moduless for the specified transfer. Both of these configurations are leveraged in the NIOS II system in Figure 4.6.

Both the read and write masters have an internal buffer which is used during memory transfers, the size of which is configurable. This facilitates faster reading and writing from various source/destination bus widths. However, the controller for write master (used in ST to MM transfers) requires that this internal buffer be filled prior to beginning any data transfers. This results in the write master attempting to read from whatever ST interface it is connected to prior to a command being

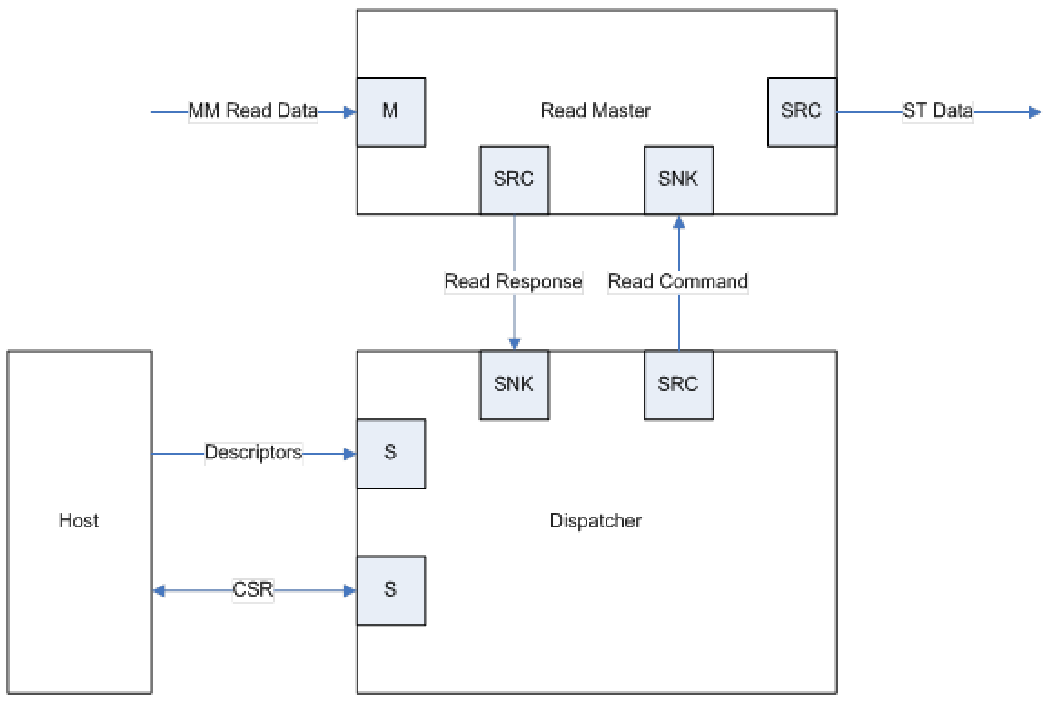


Figure 4.7: Modular SGDMA configured for MM to ST data transfer

received. This led to a number of issues on system initialization as the write master attempts to read from interfaces prior to those interfaces being ready. This issue is addressed through the addition of a state machine controlling the input and output signals of the write master based on the state.

4.4.3 Tier II State Machine

As mentioned, a state machine is added to the Tier II module to work around some unexpected behavior from the modular SGDMA write master. The state machine has five states: INIT, IDLE, READ, FLUSH and RESET. A state transition diagram for this state machine can be seen in Figure 4.9.

The INIT state is the default state. It holds the read request signal out of the Tier II module from the write master low while holding the read acknowledge signal

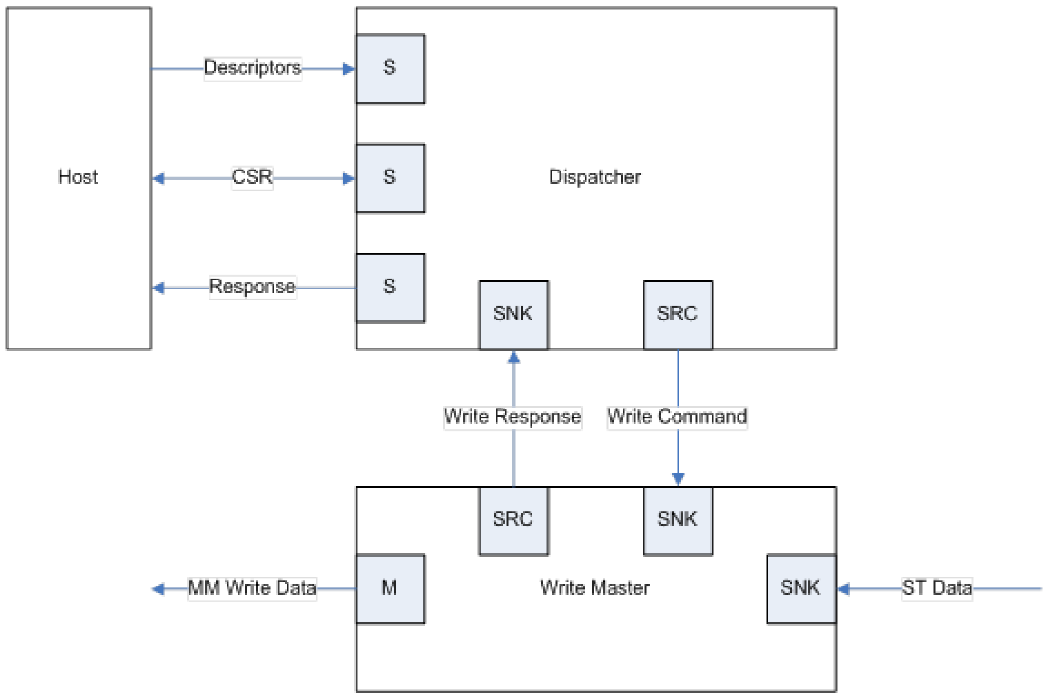
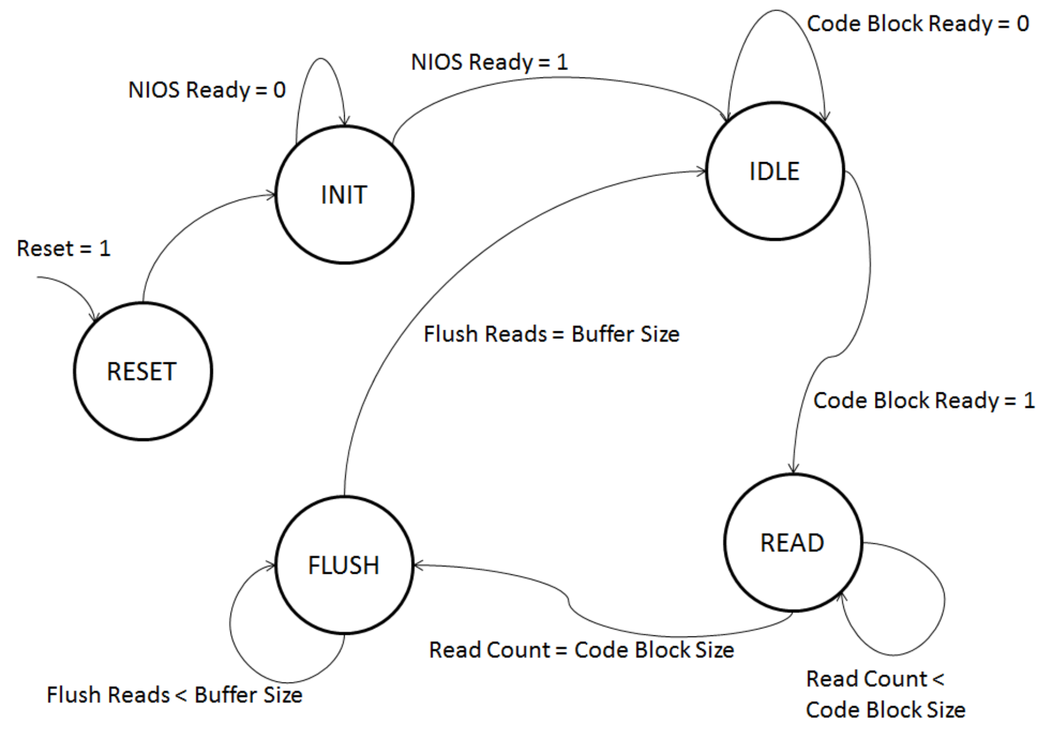


Figure 4.8: Modular SGDMA configured for ST to MM data transfer

into the write master high. This is necessary for proper initialization of the input DMA controller, as this allows the write master to fill its internal buffer without interfering with Tier I interface it is connected to prior to that interface being ready. However, by doing this, the internal buffer is filled with garbage data which must be thrown out by software. Once an initialization complete signal is received from the NIOS system, the state machine moves to the IDLE state.

The IDLE state simply waits for a signal from the Tier I output to signal that a code block is ready to be read. While IDLE, the control signals to and from the write master are considered valid and passed along to the Tier I. Once a code block is ready, the next state the READ state.

NOTE: upon receiving a code block complete signal, the NIOS II system reads in the size for the current code block and schedules a DMA transfer of that size plus



Code Block Ready = 0
NIOS Ready = 1
NIOS Ready = 0
INIT
IDLE
Reset = 1
RESET
Flush Reads = Buffer Size
Code Block Ready = 1
READ
FLUSH
Read Count = Code Block Size
Read Count < Code Block Size
Flush Reads < Buffer Size

Figure 4.9: State transition diagram for the Tier II state machine

the size of the write master's internal buffer. This is necessary to flush the interal buffer to ensure all good data is received.

The READ state allows the write master to read from the Tier I output FIFO until all of the data has been read from the FIFO. This is tracked by counting the number of read requests sent from the write master and comparing that count to the code block size. Once the read is complete the next state is the FLUSH state.

The FLUSH state allows the write master to read in data even though their is no data available (same as INIT) in order to flush the good data out of the internal data buffer. This is achieved by forcing the read request signal out of the write master low (to prevent interference with the Tier I output) while also forcing the read acknowledge into the write master high (to allow the write master to read).

This will flush the internal data buffer without disturbing the Tier I output FIFO's counts. Upon completion, the read count for the current code block is reset to zero and the next state is INIT.

The RESET state is only entered upon receiving a falling edge on the reset input signal to this module. That signals a reset of the entire module and resets the code block read count while also setting the next state to INIT, as the NIOS II system is also reset.

4.4.4 Software Design

The software design for this system uses an event-based architecture which is triggered by hardware interrupts received from both the PIOs and DMA controllers. All of the code is written in C with no C++ constructs used. The software is designed in order to reduce the code size as well as reduce overhead.

The software design follows a very similar pattern to the Tier II state machine detailed in the previous section. Upon startup, the processor enters an initialization routine. All interrupts are disabled while the processor setups up all of the interrupt service routines (ISR) to handle the various interrupts associated with the PIOs and DMA controllers. Once the interrupts have been initialized, they are reenabled. At this point, the processor waits until data has been sent to the board (this is determined by polling a GiDEL control register which holds the number of tiles to process).

Once data has been sent to the board, the processor saves the number of tiles to process. Main execution then blocks while awaiting the first tile to be fully buffered. The processor knows when a full tile is buffered by polling a global variable which is maintained by the ISR for "Write DMA Controller". This variable is refered to as TILES_BUFFERED. At this point, main execution will block indefinitely until 1024 code blocks are received from Tier I.

Upon completion of a code block, Tier I raises an interrupt in the NIOS II core on the Codeblock Ready PIO. This causes the associated ISR to execute which schedules a DMA transfer of the code block size from Tier I to main memory. Upon completion of that transfer, another interrupt is raised, this time by the "Write DMA Controller". The ISR for this interrupt checks the number of code blocks buffered for the current tile and, if 1024 code blocks have been received, updates the global TILES_BUFFERED variable. This signals the main execution thread that a tile is ready for processing.

Once a full tile has been buffered, the processor performs Tier II coding on it. The Tier II coding procedure used is adopted from [12, 14]. The code has been stripped down and modified to work on the NIOS II core. The details of this code are not explained in detail. It should be noted that while Tier II processing is being performed, interrupts are still serviced by the processor, meaning that the next tile is buffering while the current tile is being processed.

Tier II encoding builds a full JPEG2000 bitstream for the current tile, without JPEG2000 headers, in a memory buffer. Once the tile has been fully encoded, a DMA transfer is scheduled with the "Read DMA Controller", which transfers the bitstream from the NIOS II address space to the compressed bitstream MegaFIFO located in Bank A. Once scheduled, the processor is free to process the next tile, assuming there is one available.

Once all of the tiles have been processed and all DMA transfers have been completed, the processor signals the host that processing is complete by updating a feedback register through a PIO. After completion of all the tiles, the system returns to an idle state waiting for the next image to be sent down from the host.

4.4.5 Memory Layout

The NIOS II system features two memory controllers: 50 kBytes on-chip memory and 4 GB DDR2 SDRAM. By nature, the on-chip RAM is a fast, low-latency resource, but is small in size. The size of this memory is limited as the DWT and Tier I modules require a large amount of on-chip memory. The DDR2 SDRAM, on the other hand, is much slower than the on-chip memory but is also much larger. These characteristics determine how the address space of the NIOS II system is allocated.

Due to its small size and low latency, the on-chip memory is used to store the instructions for execution by the processor as well as the global variables which are used throughout the software system. The two large buffers (code block buffer, compressed bitstream buffer) both reside in SDRAM as they are far to large to reside in on-chip memory. The code block buffer must be large enough to buffer multiple tiles worth of data while the compressed bitstream buffer must be large enough to hold an entire encoded tile. In addition to these buffers, both the stack and heap used for execution by the NIOS II core are also stored in SDRAM. While this is not ideal, due to the high frequency in which these resources are accessed, their dynamic size coupled with the limited on-chip memory resources necessitates this configuration.

CHAPTER 5

OPTIMIZATIONS

Once the initial implementation is designed and implemented, it is necessary to benchmark this implementation. Not only does this determine the overall throughput of the system but also gives a baseline for comparison for the various optimizations performed. This section details the method used to test the performance of the system as well as discussing the various system optimizations made to increase the performance of the system.

5.1 Benchmarking Setup

The same test setup is used to produce all of the performance results throughout the rest of this thesis. All implementations are tested using the same two images: a single tile image and an image consisting of multiple tiles. The single tile image, refered to as “Pentagon”, is 1024x1024 aerial photo of the Pentagon building, given in Figure 5.1. This image, due to the large amount of high frequency data, tends to be difficult to compress and serves as a good worst-case scenario for a single tile. The wide area image, refered to simply as “Aerial”, is an aerial photo which consists of 120 tiles of data. This image is useful for testing the full throughput capabilities of the system. “Aerial” can be seen in Figure 5.2.

Figure 5.1: 1024x1024 “Pentagon” image used for benchmarking

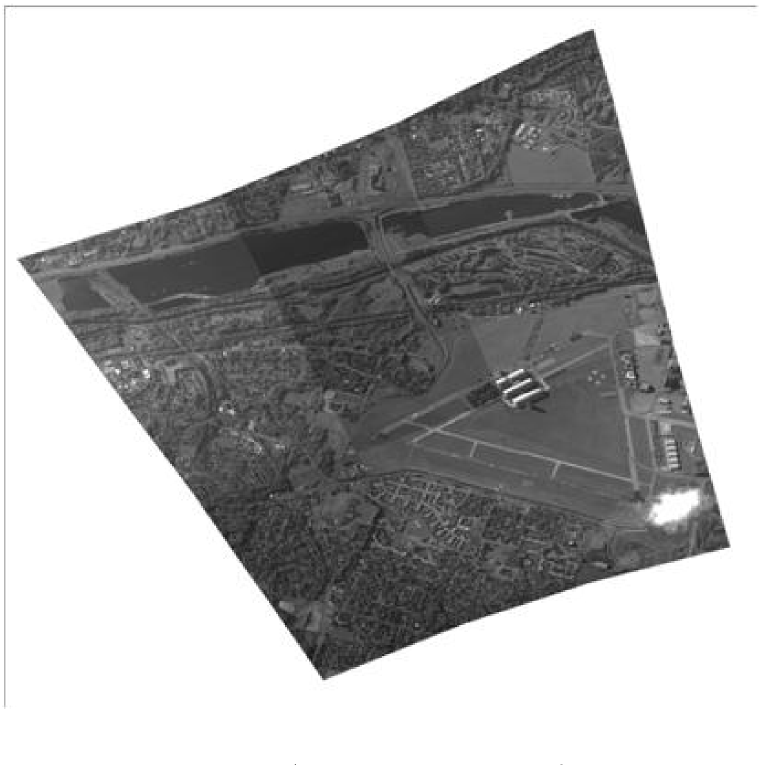


Figure 5.2: Multi-tile “Aerial” image used for benchmarking

The same quality parameters and number of Tier I block coders are used for all testing performed on the two test images. This includes testing of the existing embedded DWT/Tier I implementation, which is refered to "Existing" throughout the following sections. All test implementations are benchmarked with 25 Tier I block coders. This is found to be the minimum number of Tier I block coders necessary to achieve maximum throughput with the embedded Tier II implementation. The amount of quantization performed for benchmarking is considered visually lossless, which means that while irreversible techniques are used (DWT, quantization), these do not degrade the visual quality of the compressed image.

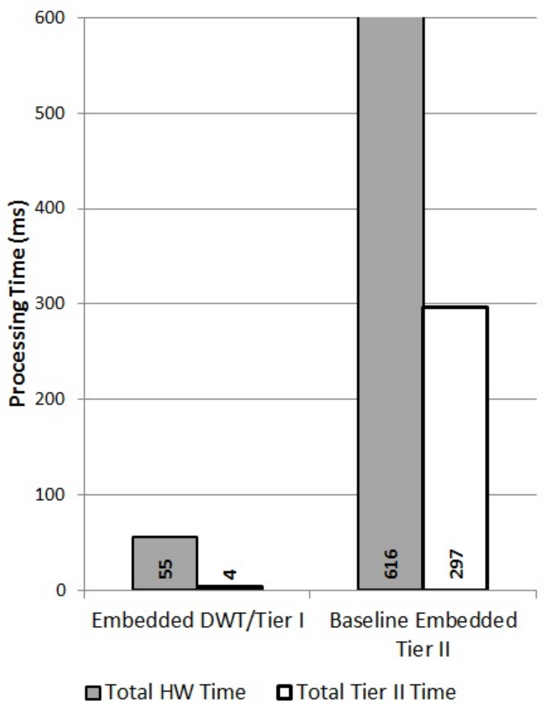
Two metrics are used to describe the performance of the system. The first metric, Total HW Time, is the total time required to compress an image. This includes the transfer times to and from the board as well as the time spent processing. The second metric, throughput, measures the throughput of the system in MBytes per second. The throughput calculation is given in Equation 5.1, where *Data* is the input data size and *Time* is the processing time.

$$Throughput = \frac{Data(Mbytes)}{Time(seconds)} \tag{5.1}$$

A third metric, Total Tier II Time, is frequently shown alongside the Total HW Time. This is a measure of time spent processing on the NIOS II core, including time spent waiting for code blocks. This metric is useful for quantifying the (assumed) slowdown that will result from the addition of the NIOS II core.

5.1.1 Baseline Embedded Tier II Performance Results

Prior to performing any system optimizations, the baseline embedded Tier II design is first benchmarked against "Existing". This provides an idea of how much optimization is needed to achieve comparable performance. Both implementations are tested with both “Pentagon” and “Aerial” test images. The results from this initial benchmarking are shown in Figures 5.3 and 5.4.



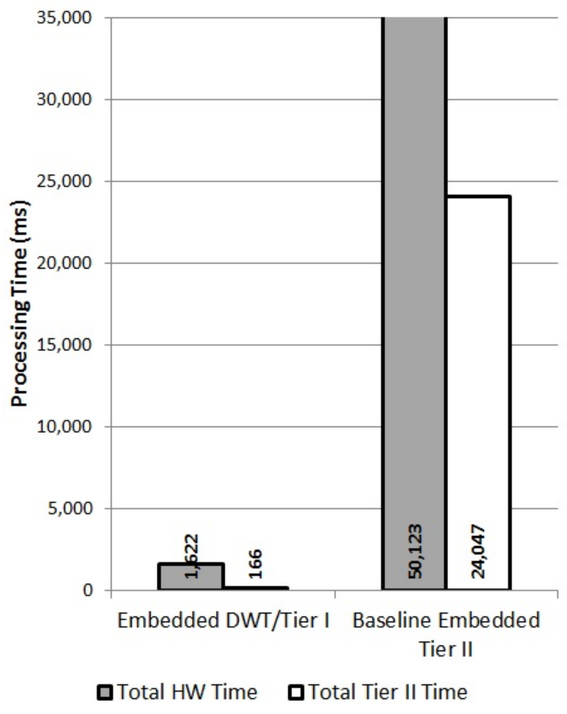
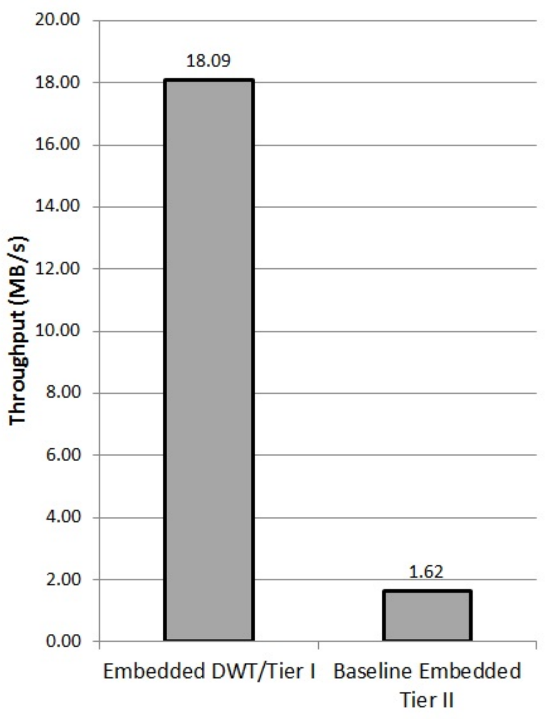


Figure 5.3: Total processing time comparison between “Existing” and the baseline embedded Tier II designs (“Pentagon” on left, “Aerial” on right)

Both Figure 5.3 and Figure 5.4 tell the same story: the baseline embedded Tier II design is far slower than “Existing”, which is nearly 12 times faster than the baseline embedded Tier II design for “Pentagon”. It is important to note that the Total HW Time for the baseline embedded Tier II design is nearly twice the Total Tier II Time. While the Total Tier II Time is slow, this also shows that the NIOS II system is slowing down the other upstream modules. Both the Total Tier II Time



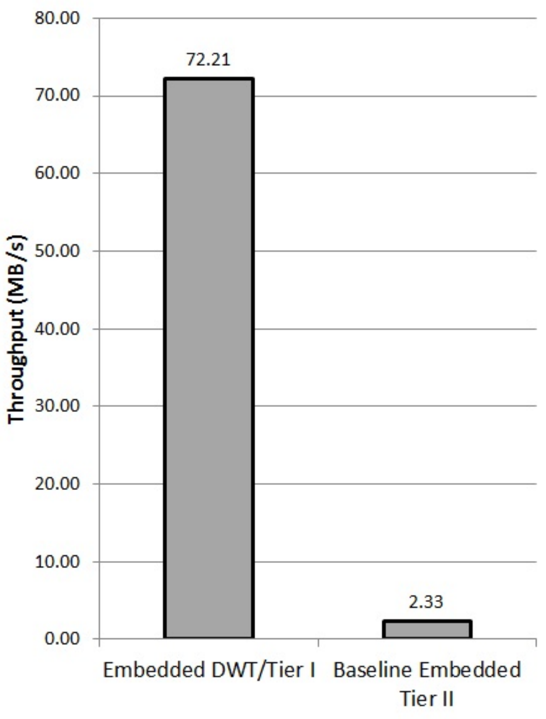


Figure 5.4: Throughput comparison between "Existing" and the baseline embedded Tier II designs ("Pentagon" on left, "Aerial" on right)

and the large disparity between the Total HW and Tier II times are addressed in later sections.

5.2 System Profiling

As Figures 5.3 and 5.4 show, optimizations are needed to enable the system to perform close to the same level as "Existing". In order to determine potential areas for optimization, the software running on the NIOS II core is profiled using the two performance counters included in the SOPC Builder system. These performance counters are used to determine the percentage of execution time taken by certain functions.

For the purposes of this thesis, the total execution time is partitioned into five segments: Buffer to CBs, Create Code Stream, Create Precinct Packet, Variable Length Encode and Calculate Indicator Bits. The software profiling results for the baseline embedded Tier II design are shown in Figure 5.5.

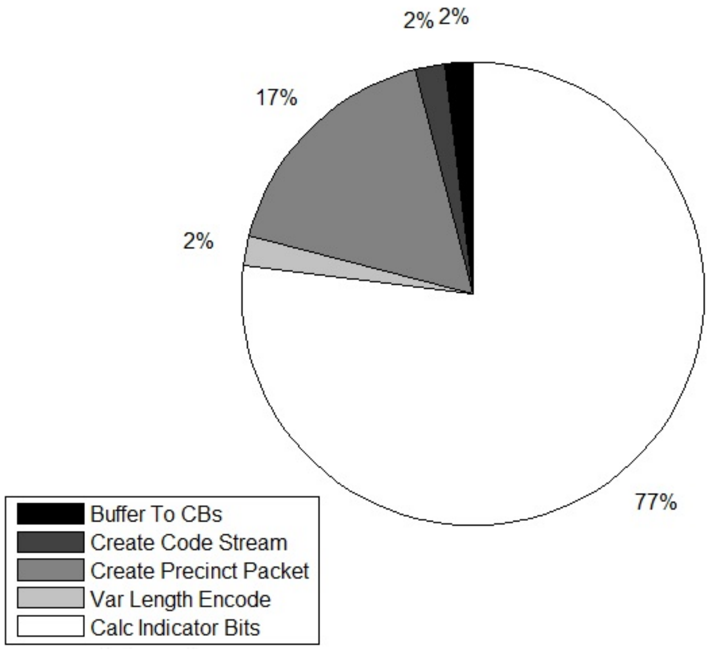


Figure 5.5: Software profile of baseline embedded Tier II implementation for "Pentagon"

Buffer to CBs is a setup function which is performed after all of the code blocks for a tile have been received. This function is responsible for parsing through the buffer containing all of the received code blocks and forming a structure to hold all of the information associated with each code block in the tile. This function does not perform any major calculations and comprises a small amount (2%) of the processing time.

Create Code Stream is the top level function call for Tier II processing. This is called after Buffer to CBs is completed. This function allocates space for the final bitstream and adds some headers to the tile prior to beginning the processing. This funciton is mostly comprised of overhead tasks and therefore is a small section (2%) of the total system profile.

Create Precinct Packet is called by Create Code Stream and performs most of the Tier II encoding work. Note that the Create Code Stream time does not include the time to perform this function call and all subsequent function calls. This function performs all of the Tier II encoding necessary for each code block and therfore takes a large chunk of the total processing time (17%).

Variable Length Encoding is responsible for encoding the number of coding passes for each code block with a variable length code word. This is a relatively quick function and only takes 2% of the baseline software profile.

Calculate Indicator Bits is responsible for calculating the number of bits required to signal the number of bytes contributed to a packet by a code block. In other words, this function calculates the number of bits needed to represent how many bytes a code block contributes to the bit stream. The equation for calculation the number of indicator bits can be seen in Equation 5.2,

$$bits = Lblock + \lfloor log_2(P) \rfloor, \tag{5.2}$$

where *Lblock* is the state variable for the current code block and P is the number of coding passes contributed for the current code block [11]. From Figure 5.5 it is clear that this is a very expensive operation, taking 77% of the total processing time.

5.3 Optimizations

Based on the performance results shown in Figures 5.3 and 5.4, it is clear that the baseline embedded Tier II system needs to be optimized in order to have any value. With the performance profile obtained in the previous section (Figure 5.5), the most obvious candidate for optimization is the Calculate Indicator Bits function. The first two optimizations presented target this function, while the other two optimizations presented are focused on more general throughput improvements.

5.3.1 Custom Floating-Point Instructions

While the calculation performed in the Calculate Indicator Bits (see Equation 5.2) does not appear to be a computationally expensive operation, the profiling results in Figure 5.5 show otherwise. The reason this function is so expensive is due to the binary logarithm calculation for each code block. This is implemented as a floating-point (FP) operation and is by nature much more computationally expensive than integer arithmetic. The cost of FP operations are compounded by the RISC architecture used by the NIOS II core. Almost all of the Tier II algorithm used in this design is implemented using integer math, with this one binary logarithm being the only exception.

Fortunately, the NIOS II/f core supports the use of up to 256 additional custom instructions. Altera offers a custom IP core to perform FP arithmetic, which adds custom FP instructions to perform addition, subtraction, multiplication and division. The IP core adds custom FP implementations of many common math library functions as well, including logarithms [5]. While the FP instructions are simply added through a menu dialogue option on the NIOS II core, the impact on processing time is dramatic. The addition of the FP instructions results in a

58% decrease in processing time for Tier II. Note this performance increase was for "Pentagon", a single tile. However, this calculation still takes up 35% of the total Tier II time, as shown in Figure 5.6.

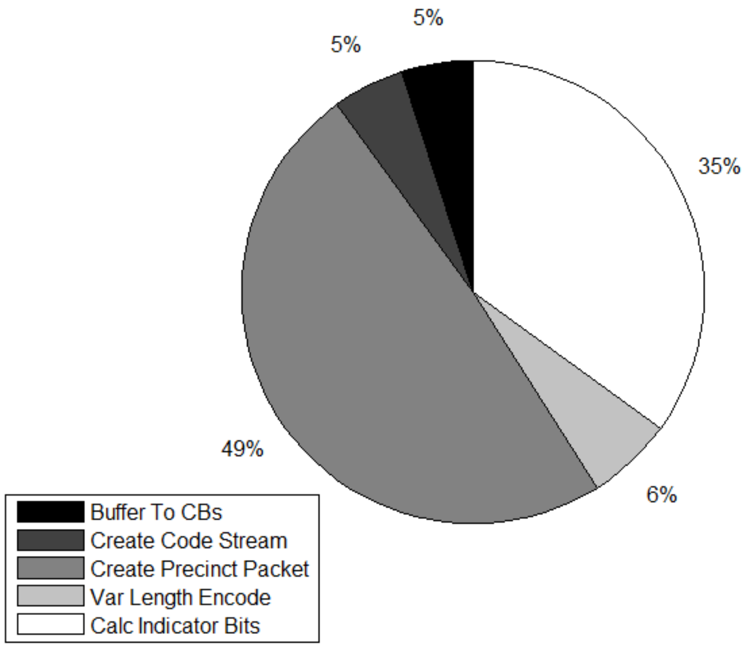


Figure 5.6: Software profile of embedded Tier II implementation with custom FP instructions for "Pentagon"

5.3.2 Lookup Table Binary Logarithm Implementation

Although the addition of custom FP instructions to the NIOS II core yield a 66% decrease in processing time, the overall Total HW Time is still much slower than "Existing". Additionally, the Calculate Indicator Bits function still takes 35% of the overall Tier II processing time.

In order to further optimize this function, a custom implementation of the binary logarithm library call is designed. Equation 5.2 shows that only the floored result of the binary logarithm is used. This means that FP arithmetic is not necessary, since the fractional portion of the result is truncated. Instead, the calculation can instead be implemented as a lookup table (LUT) which takes a 32-bit input and produces the correct result. This eliminates the need for custom FP instructions entirely and reduces the operation to only a few jump instructions.

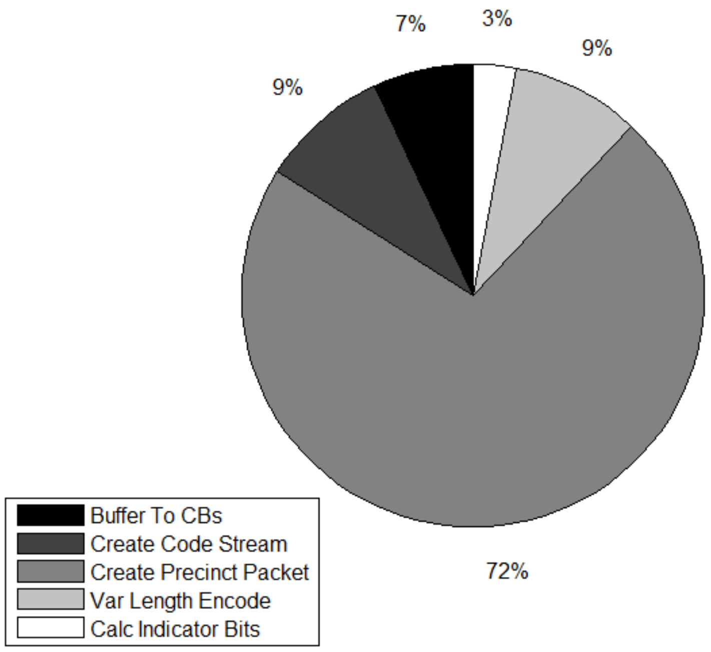


Figure 5.7: Software profile of embedded Tier II implementation with LUT binary logarithm for “Pentagon”

Therefore, the custom FP instructions added in the previous section are removed from the system and the binary logarithm library call in Calculate Indicator Bits is replaced with the LUT binary logarithm implementation. When the impact of the LUT implementation is compared to the performance using custom FP instructions, we find that the LUT implementation is 29% faster than the custom FP instruction implementation. The updated performance profile, which can be seen in Figure

5.7, shows that the Calculate Indicator Bits function is now only 3% of the total Tier II time, as opposed to 35% with the custom FP instructions and 77% prior to optimization.

5.3.3 Tightly-Coupled Memories

Once the Calculate Indicator Bits function is fully optimized, now only comprising 3% of the total Tier II processing time (Figure 5.7), general optimizations are performed to the system in order to increase overall system throughput. The first of the general system optimizations is targeted at decreasing the latency when accessing certain data structures which are commonly accessed. This is done by adding a tightly-coupled memory module to the NIOS II system.

A typical NIOS II core has two Avalon memory-mapped master ports, the instruction and data masters. These ports master memory devices to retreiving intructions and data from memory. However, these standard master ports must deal with bus arbitration on the Avalon Bus and are also separated from the processor by the instruction and data caches. These layers separating the memories from the NIOS II master ports adds latency to memory access in these devices [7].

One option for reducing the memory access latency for data and instructions is to add additional data and instruction master ports which are designated as tightly-coupled ports. These tightly-coupled data and instruction ports master individual dedicated tightly-coupled memories. A block diagram of a NIOS II system with tightly-coupled memories can be seen in Figure 5.8. These tightly-coupled memories must be on-chip memory and are not connected to the Avalon Bus. Instead, these memories are directly coupled to the NIOS II core, providing

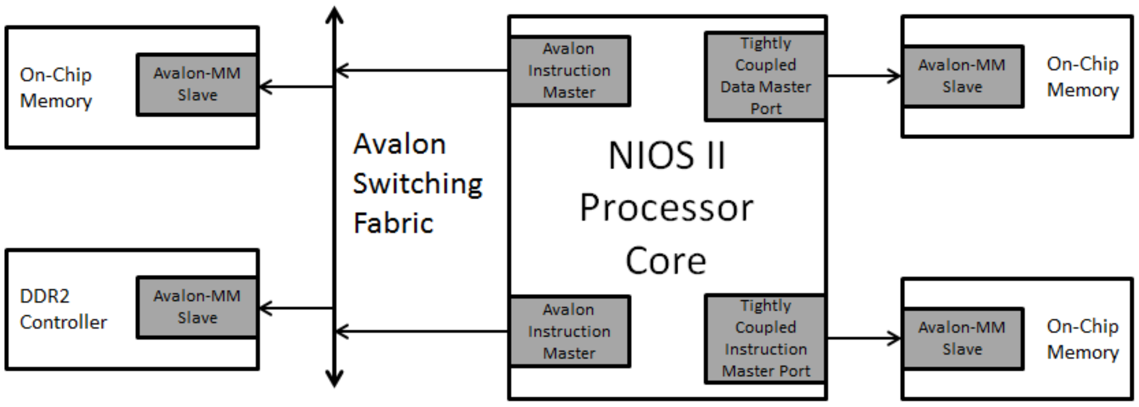


Figure 5.8: Block diagram of a NIOS II system with tightly-coupled memory modules

guaranteed low-latency memory access without having to deal with maintaining cache coherency (see Chapter 9 of [7] for more details on NIOS II caching).

For this implementation, a single tightly-coupled data memory is added to the system. The on-chip memory used is 30 kBytes and is clocked at the same 290 MHz clock as the NIOS II core. Certain commonly accessed functions, such as the structure holding all of the relevant data for each code block in a tile, are designated as tightly-coupled data using special preprocessor definitions. This data is placed into the tighly-coupled data memory address space by the linker. The addition of a tightly-coupled data module yields a 16% reduction in total Tier II processing time for “Pentagon”. While the performance increase is minimal when compared to the other optimizations, the tightly-coupled memory also has the benefit of bypassing the data cache. This was a very useful feature when eliminating some bugs which stemmed from cache corruption.

5.3.4 Non-Blocking Memory Transfers

The final optimization performed is targeted at decreasing overall system throughput, not just optimization of a single function. However, the motivation for this optimization is the result of the performance profile in Figure 5.7 which shows that while the Calculate Indicator Bits function has been heavily optimized, the Create Precinct Packet function accounts for 72% of the total processing time. The reason this function takes such a long is due to large memory copies that are performed in this function.

The Create Precinct Packet function is responsible for copying each code block from the input buffer to the bitstream buffer prior to output. Since both of these buffers are located in the SDRAM address space, these memory copies are relatively slow compared to the other functions performed by the system. These memory copies are implemented using the standard C “memcpy” function, a blocking function call. Blocking function calls prevent the host processor from executing other instructions while the call completes. Since the memory copy in the SDRAM address space is slow, this results in wasted time spent waiting for the transfers to complete.

To eliminate the idle time spent waiting on blocking memory copies, non-blocking memory copies are used to perform these memory transfers in the SDRAM address space. To implement non-blocking memory transfers, a third DMA controller is added to the NIOS II system. This DMA controller is configured to perform transfers between Avalon memory-mapped interfaces, and both the input and output Avalon MM ports master the DDR2 SDRAM controller. An updated block diagram of the SOPC Builder system with the third DMA controller as well as the tightly-coupled data memory added can be seen in Figure 5.9.

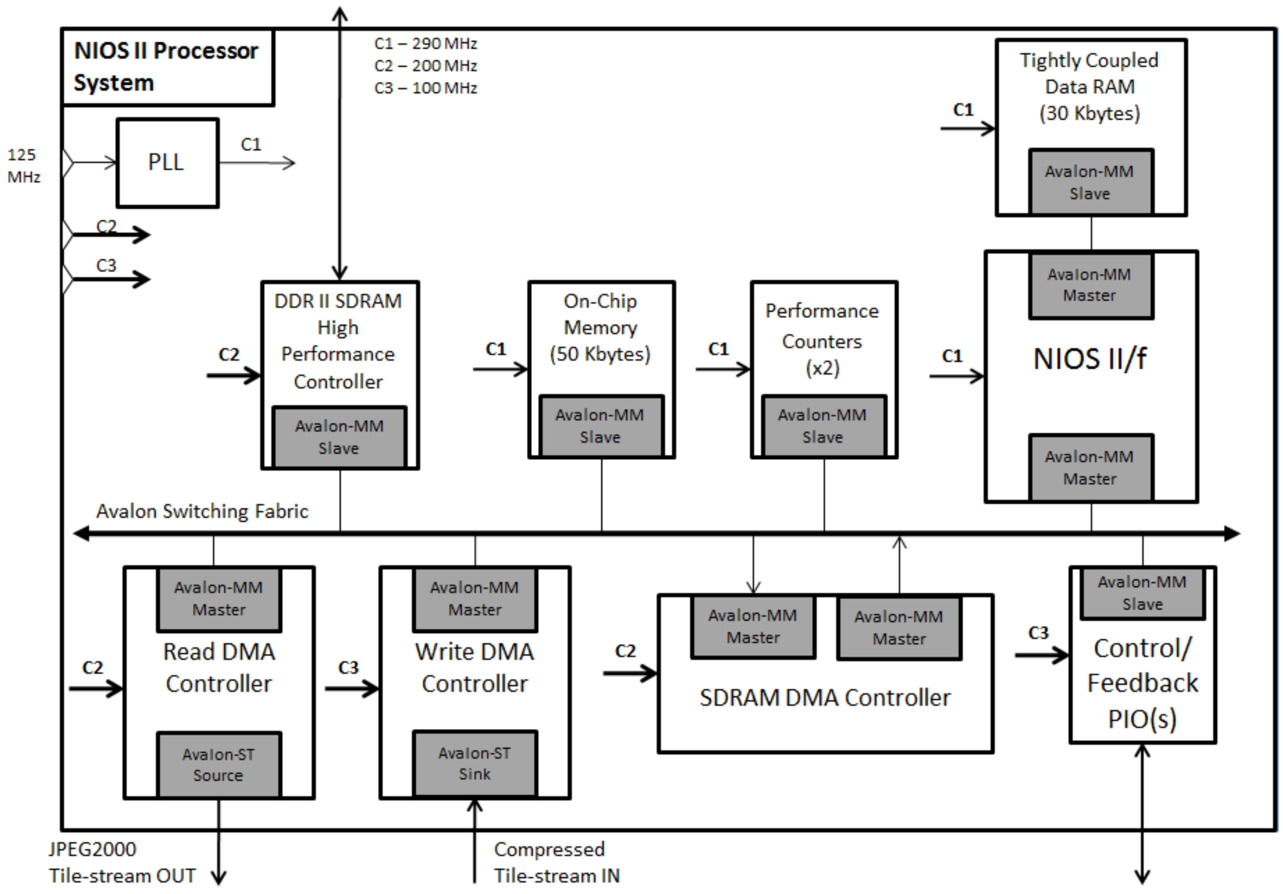


Figure 5.9: SOPC System design to perform Tier II processing with addition DMA controller and tightly-coupled memory

With the additional DMA controller, all the processor must do is send a data transfer request to the DMA controller instead of calling 'memcpy'. The processor is then free to continue executing instructions. There is no fear of data corruption from this implementation, since the contents of the input buffer are considered read-only and the contents of the output bitstream buffer are not read until the entire bitstream has been written. The performance impact of non-blocking memory copies is dramatic, resulting in a 67% reduction in Tier II processing time for a single tile ("Pentagon"). The overall performance of the optimized system is detailed in the following section. A final software profile can be seen in Figure 5.10. While Create Precinct Packet is still the largest chunk, the processing time is more evenly distributed between the other functions. Create Precinct Packet will always be the longest function, as it performs almost all of the Tier II processing for the tile.

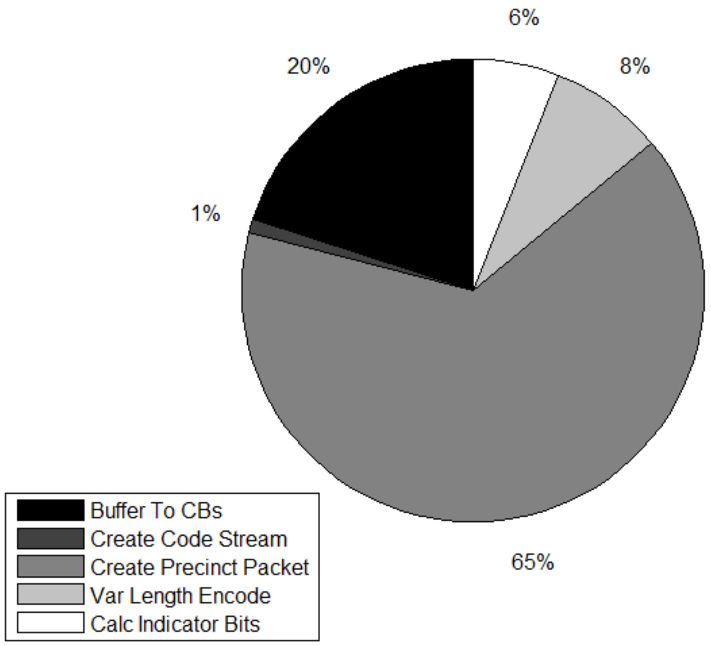


Figure 5.10: Final software profile of embedded Tier II implementation with all optimizations applied

5.4 Optimization Results

As detailed in the previous section, four sets of optimizations are applied to the system to increase the overall performance of the system. First, custom FP instructions are added to the NIOS II instruction set to reduce the computation time of a binary logarithm calculation necessary for code block encoding. This yielded a 58% reduction in processing time. Next, the custom FP instructions were removed from the system. Instead, the binary logarithm library call was replaced with a custom LUT floored binary logarithm. The LUT binary logarithm yielded an addition 30% reduction in processing time. In addition, this reduced the Calculate Indidcator Bits function from 77% of the total Tier II time down to 3%.

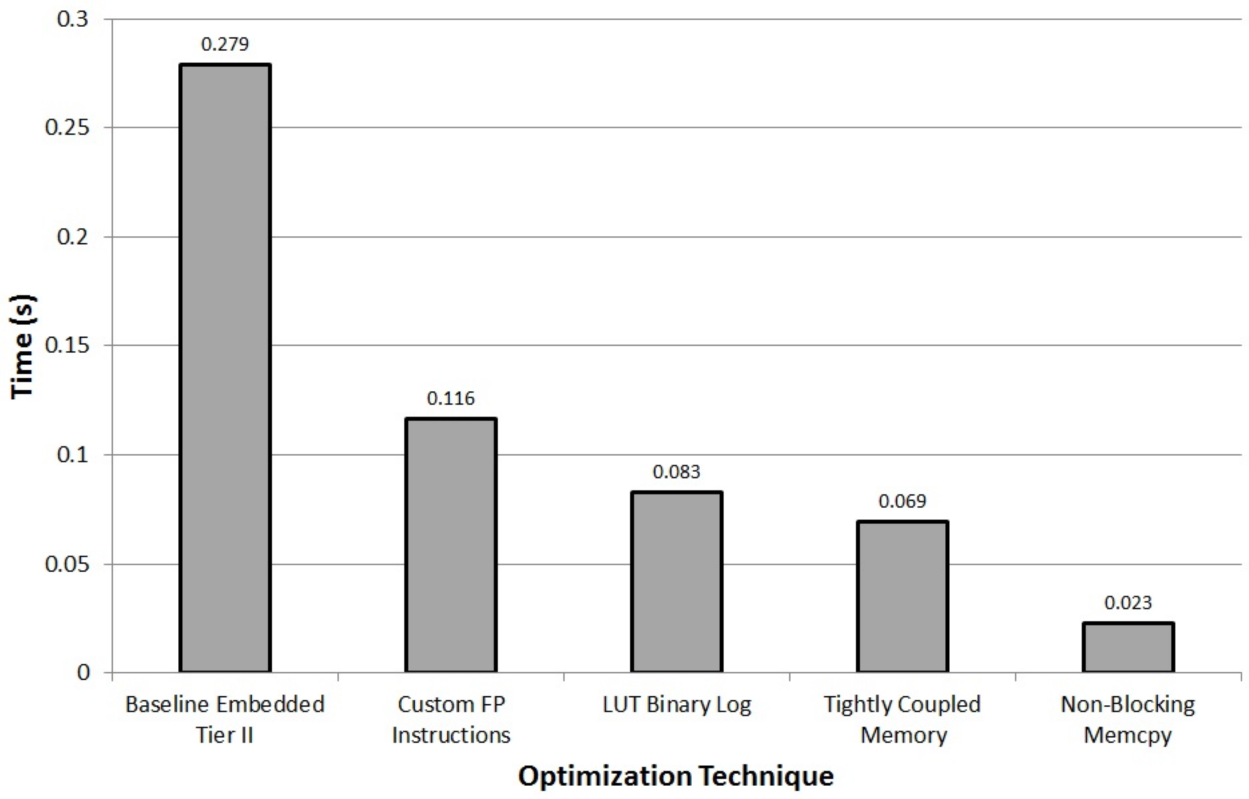


Figure 5.11: Reduction in total Tier II processing time for “Pentagon” as optimizations are applied to the system

Once the Calculate Indicator Bits function is optimized, two general optimizations are performed. First, a tightly-coupled data memory module is added to the NIOS II system. This guarantees low-latency memory access to certain commonly used data structures. This results in a 16% reduction in processing time while also eliminating some cache corruption issues. Finally, a third DMA controller is added to the NIOS II system to implement non-blocking memory transfers in the SDRAM address space. This optimization has the largest impact of all the optimizations, resulting in a 67% reduction in processing time. The impact of all four optimizations on the total Tier II processing for “Pentagon” can be seen in Figure 5.11. The overall Tier II processing time is reduced from 279 ms down to 23 ms, an overall reduction of 92%.

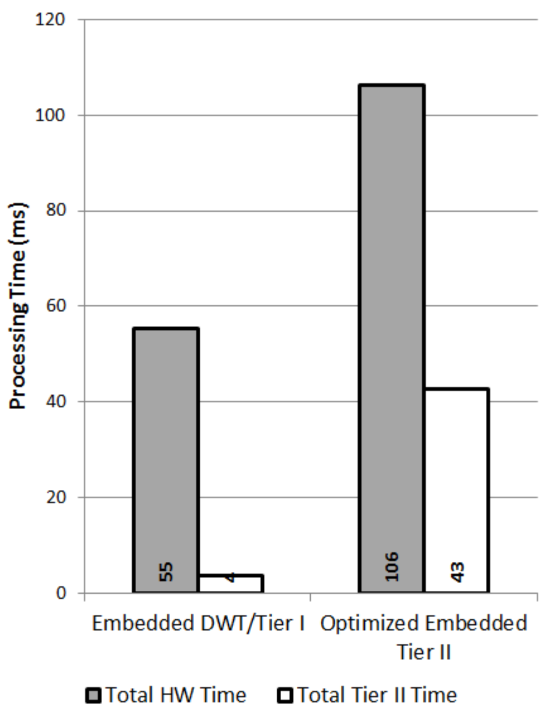
CHAPTER 6

PIPELINED DESIGN

In the previous section, optimizations performed on the NIOS II system for Tier II processing are detailed. The results are very encouraging as the total Tier II processing time is reduced by over 90%. However, the large disparity between the total HW time and the total Tier II time (see Figure 5.3) is still present. By looking at the performance of “Existing” it is clear that the DWT/Tier I modules are running slower than they are capable of. In order to realize the full performance this system, the DWT/Tier I module must be able to run at it’s full speed, with the total HW time limited only on the Tier II processing time.

6.1 Motivation

As mentioned in the introduction to this chapter, a large gap still exists between the total HW and Tier II times, suggesting that the DWT/Tier I modules are being slowed down by the Tier II module. This issue is further displayed in Figure 6.1 which shows a comparison between “Existing” and the fully optimized embedded Tier II design. Even with the fully optimized NIOS II system, “Existing” is still 50% faster than the embedded Tier II design “Pentagon”, and 60% faster for “Aerial”. However, if the total HW time is closer to the total Tier II time, the embedded Tier II design would only be approximately 25% slower.



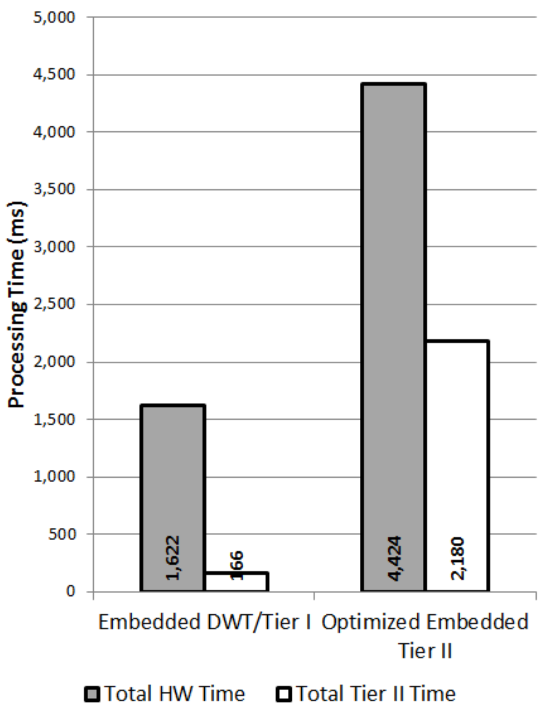


Figure 6.1: Total processing time comparison between "Existing" and the optimized embedded Tier II designs ("Pentagon" on left, "Aerial" on right)

The reason for the large disparity between the total HW and Tier II times (Figure 6.1) is the direct coupling of the Tier I output and the Tier II input in the original embedded Tier II design. Each Tier I block coder in the Tier I module has a FIFO on the back end large enough to store one compressed code block. A controller in the Tier I module cycles through the block coders, exposing the control signals for each output FIFOs to the output interface. Once the data is read from a block coder's output FIFO, the Tier I controller moves to the next block coder. Each block coder must wait for the contents of that FIFO to be read prior to processing the next code block.

The output FIFO control signals from the Tier I module are directly coupled to the DMA controller of the NIOS II system responsible for reading the code blocks.

As code blocks are completed, interrupts are raised in the NIOS II system which schedule a data transfer with the appropriate DMA controller. This architecture is acceptable if the DMA controller is able to read data as fast as the output FIFO can provide it, as this would not slow down the block coders. However, due to the limitations of the DDR2 memory controller to which the DMA controller is writing to, the transfer rate is far slower than the maximum transfer rate of the block coder's output FIFO. Since the DMA controller is not capable of reading as fast as the block coders can write, the block coders spend a lot of time idle, waiting for the transfers to complete. All of this idle time causes the Tier I to take much longer than anticipated and results in the inflated total HW times seen in Figure 6.1.

6.2 Implementation

In order to realize a fully pipelined design, the Tier II module must be decoupled from the Tier I output FIFOs. This will allow the Tier I to output data as fast as it can, resulting in no wasted time spent idle. The other two stages in the pipeline, the DWT and Tier I modules, are separated by a MegaFIFO. This same approach is used to decouple the Tier I and Tier II modules. A large MegaFIFO is inserted between the two modules. The Tier I module writes to the FIFO input while the Tier II module reads from the FIFO output. This allows both modules to read/write at their desired rate providing the perfect sychronization between the two modules. A block diagram showing the top level data flow through the pipelined system is shown in Figure 6.2 (compare to Figure 4.5). The new MegaFIFO resides in Bank A memory and is large enough to hold multiple tiles worth of data.

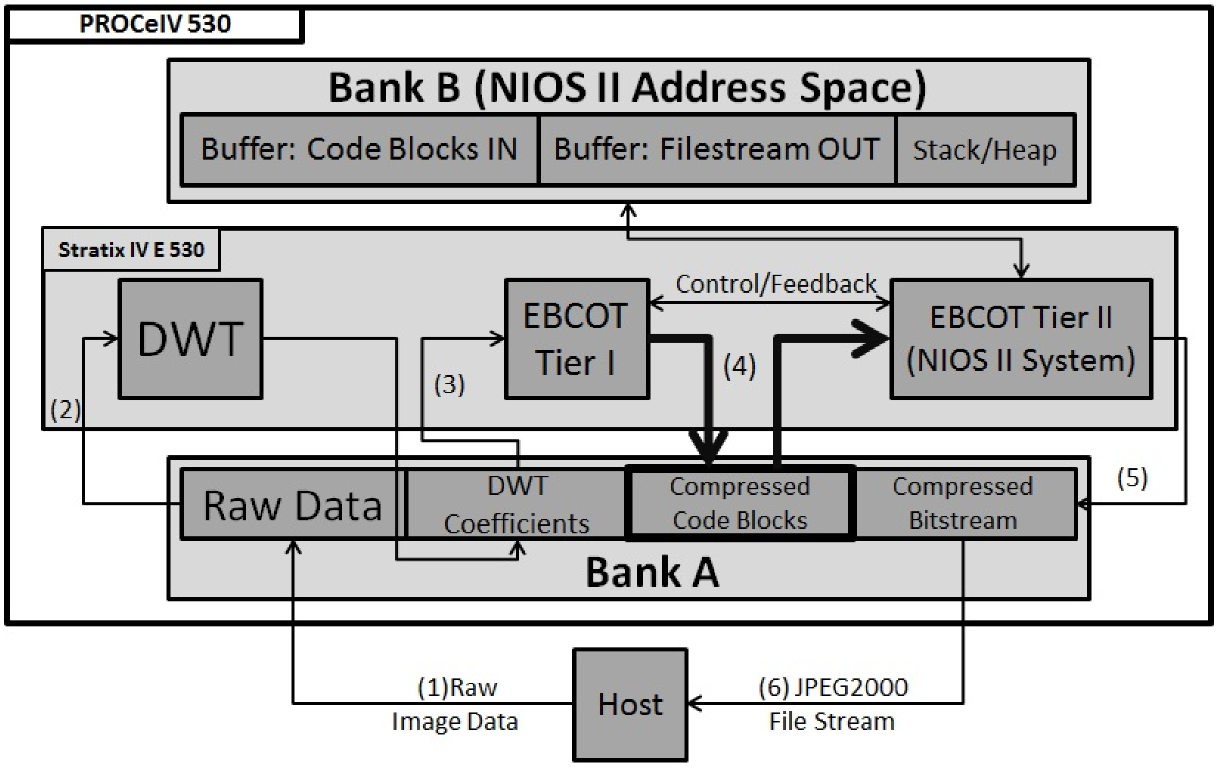


Figure 6.2: Block diagram of the pipelined architecture. New FIFO and data flow are highlighted

With the additional FIFO added, the Tier I block coders write directly to the input port of this FIFO. The Tier II module then reads from the output port of the MegaFIFO. Since both MegaFIFO ports have independent clocks, both modules are capable of reading/writing at their desired rates. This enables the Tier I module to process code blocks at its full rate, without having to wait for the Tier II module. The Tier I module would have to wait only if the MegaFIFO was full. However, due to the large FIFO size (16MBytes) and the processing speed of the Tier II module, this does not occur. While the Tier I module does get ahead of the Tier II module, the MegaFIFO size makes this a non-issue.

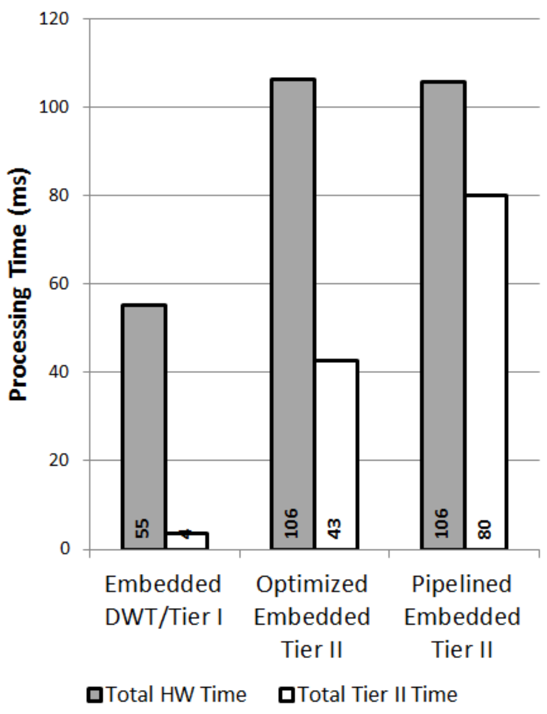
Only minor changes are needed in the Tier II module to facilitate reading from the MegaFIFO. The most noticable change is that, instead of reading code blocks one at a time, Tier II only receives an interrupt when an entire tile is ready. This reduces the overhead associated with servicing 1024 code blocks per tile as only one interrupt is received per tile instead. Minor changes are also necessary in the Tier II state machine (Figure 4.9) used to handle the flush case needed for the DMA write master. Instead of comparing the read count to a code block size, this count is instead compared to the tile size.

6.3 Performance Results

6.3.1 Pipelined Performance Results

Once the pipelined design is implemented, the design is tested using the same procedures outlined in the test setup. A comparison of the total processing times of "Existing" and the optimized and pipelined embedded Tier II designs can be seen in Figure 6.3. This figure shows the performance results from both "Pentagon" and "Aerial". A similar comparison can be seen in Figure 6.4, which shows the throughput results for all three designs.

There are a two important points to note when examining the results. The first thing one may notice is that the total Tier II times between the optimized and pipelined embedded Tier II designs actually increased (for both images). This increase can be attributed to the switch from reading individual code blocks to reading entire tiles. In both implementations, the interrupts received signaling a tile is complete are received at approximately the same time (when the last code block is processed). In the optimized case, where individual code blocks are read, this interrupt signals that the 1024th code block is complete and ready to be read.



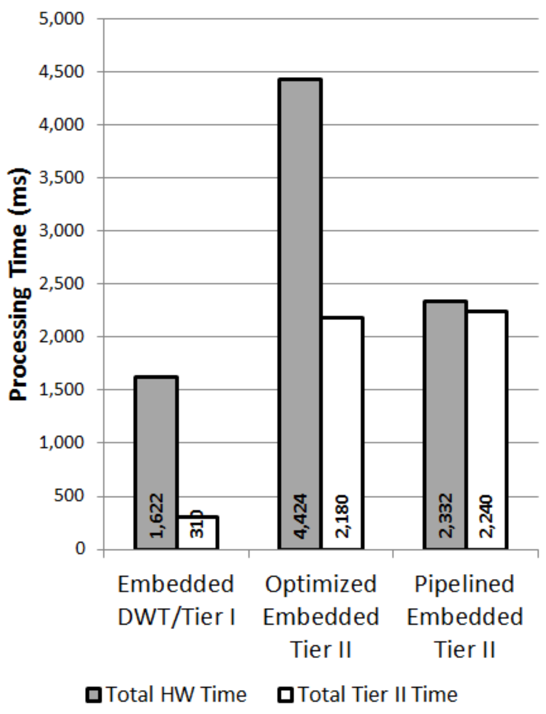
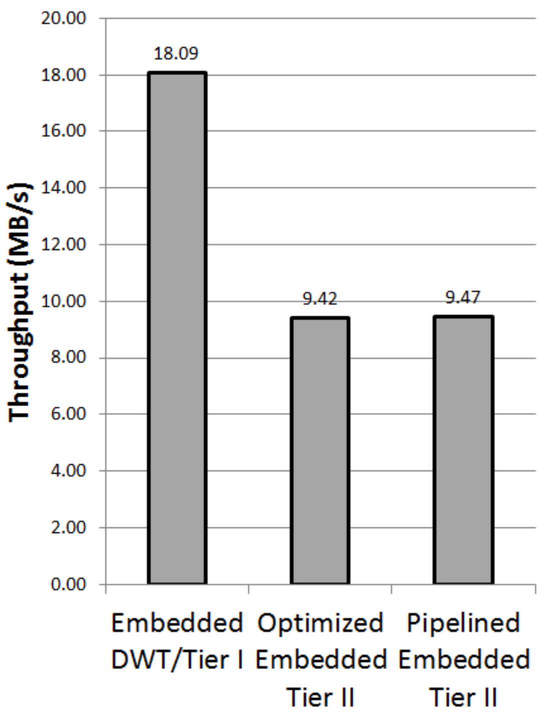


Figure 6.3: Total processing time comparison between "Existing", optimized embedded Tier II and pipelined embedded Tier II ("Pentagon" on left, "Aerial" on right)

In the pilelined case, this interrupt signals that an entire tile is complete and ready to be read. While received at approximately the same time, in the optimized case only a single code block must be read in while in the pipelined case and entire tile must be read. This causes the total Tier II time to increase slightly, but the "Pentagon" results show that this has no impact on the total HW time.

The second thing to notice is that the pipelined design has no impact on the total HW time for "Pentagon" but has a dramatic impact on the HW time for "Aerial". This is because "Pentagon" is only a single tile image while "Aerial" is composed of 120 tiles. The pipelined design does not impact the Tier II processing time. Instead, it decoupled the Tier I output from Tier II input, allowing the Tier



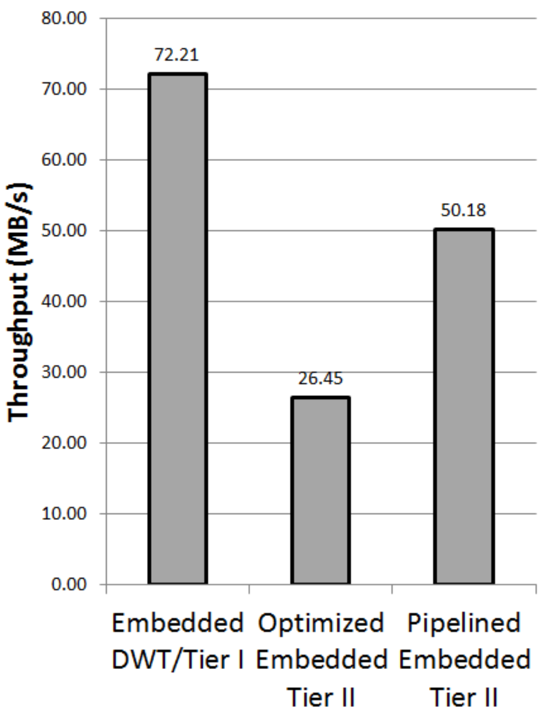


Figure 6.4: Throughput comparison between "Existing", optimized embedded Tier II and pipelined embedded Tier II ("Pentagon" on left, "Aerial" on right)

I block coders to operate as fast as possible without waiting for Tier II to read the data out of them. The impact of the decoupling is apparent when examining the results for "Aerial". While there is a slight increase in the total Tier II time, the total HW time decreased by 47%. The same percentage increase in throughput can be seen in Figure 6.4. Additionally, the total HW time now appears to be totally dependant on the total Tier II time, as expected. This is obvious when looking at the total HW time for "Existing", as this time is faster than the total Tier II time for the same image.

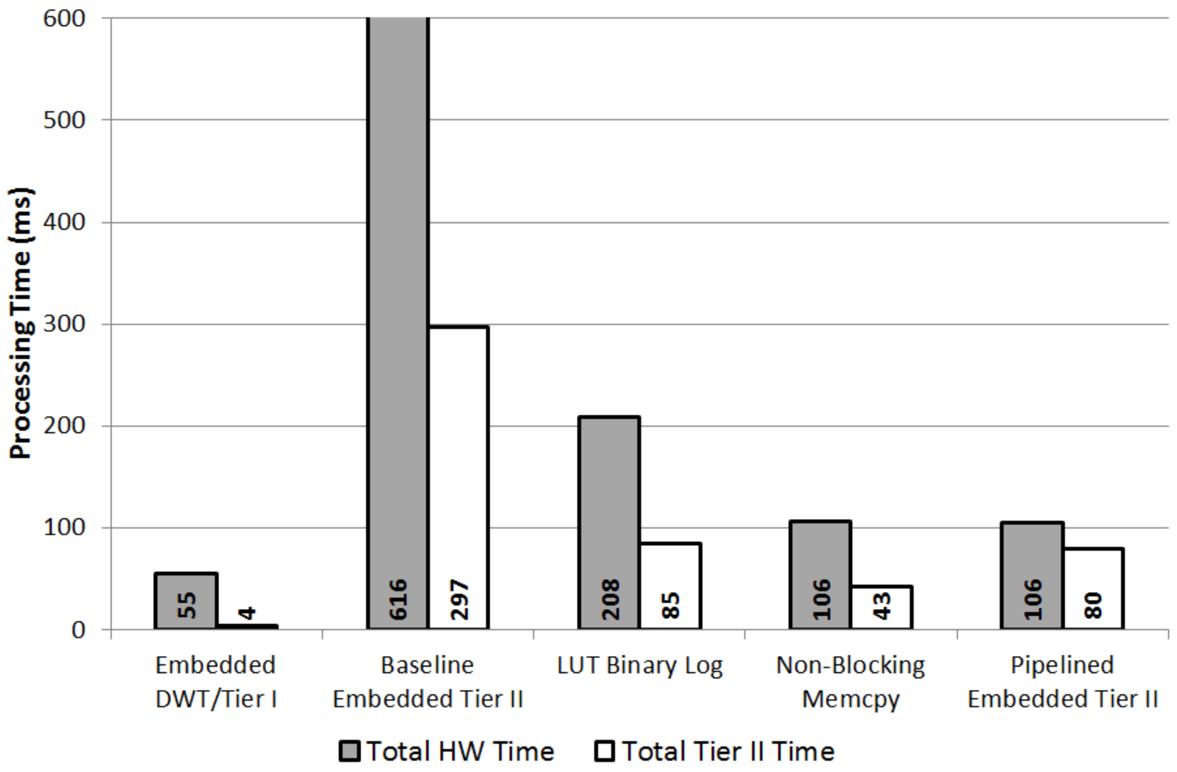


Figure 6.5: Total processing time comparison between "Existing" and the optimizations made to the embedded Tier II system (for "Pentagon")

6.3.2 Overall Performance Results

Once the pipelined design has been implemented and benchmarked, the overall performance results for all of the major system optimizations are compiled to show the progression of the system performance. The following figures include "Existing" as well as all of the major optimizations made to the system. However, these results do not include performance results for the custom FP instructions, as this optimization is replaced with the LUT binary logarithm implementation. Figures 6.5 and 6.6 show the total HW time and throughput comparisons (respectively) for the "Pentagon" image. For "Pentagon", the baseline embedded Tier II implementation was 90% slower than "Existing". The addition of the LUT binary logarithm increased the system performance by 66% from the baseline embedded Tier II implementation. By adding a third DMA controller to realize non-blocking memory copies, the system throughput was increased by an additional 50%. While the pipelined design did not yield any performance gains for the single tile “Pentagon”, the total HW time was reduced by 83% as compared to the baselined embedded Tier II design.

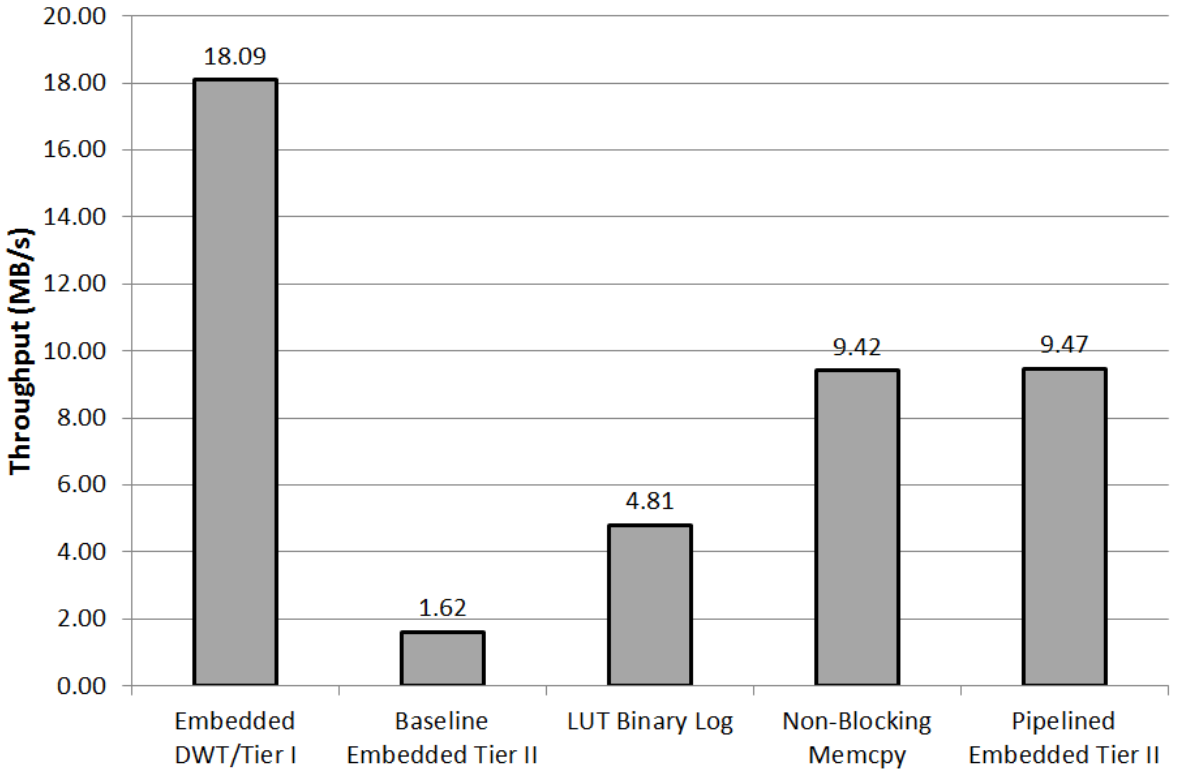


Figure 6.6: System throughput comparison between “Existing” and the optimizations made to the embedded Tier II system (for “Pentagon”)

The total HW time and throughput results for “Aerial” can be seen in Figures 6.7 and 6.8 (respectively). These figures tell a similar story to the results for “Pentagon”. The baseline embedded Tier II design was 95% slower than “Existing” for “Aerial”. With the addition of the LUT binary logarithm, the system performance was increased by 80% over the baseline implementation. Non-blocking memory copies yielded an additional 56% increase in performance. Most importantly, the pipelined design yielded a 47% increase over the fully optimized design, as the system now operates as a true tile pipeline. Overall, processing time for “Aerial” was decreased by 95% over the baselined embedded Tier II implementation.

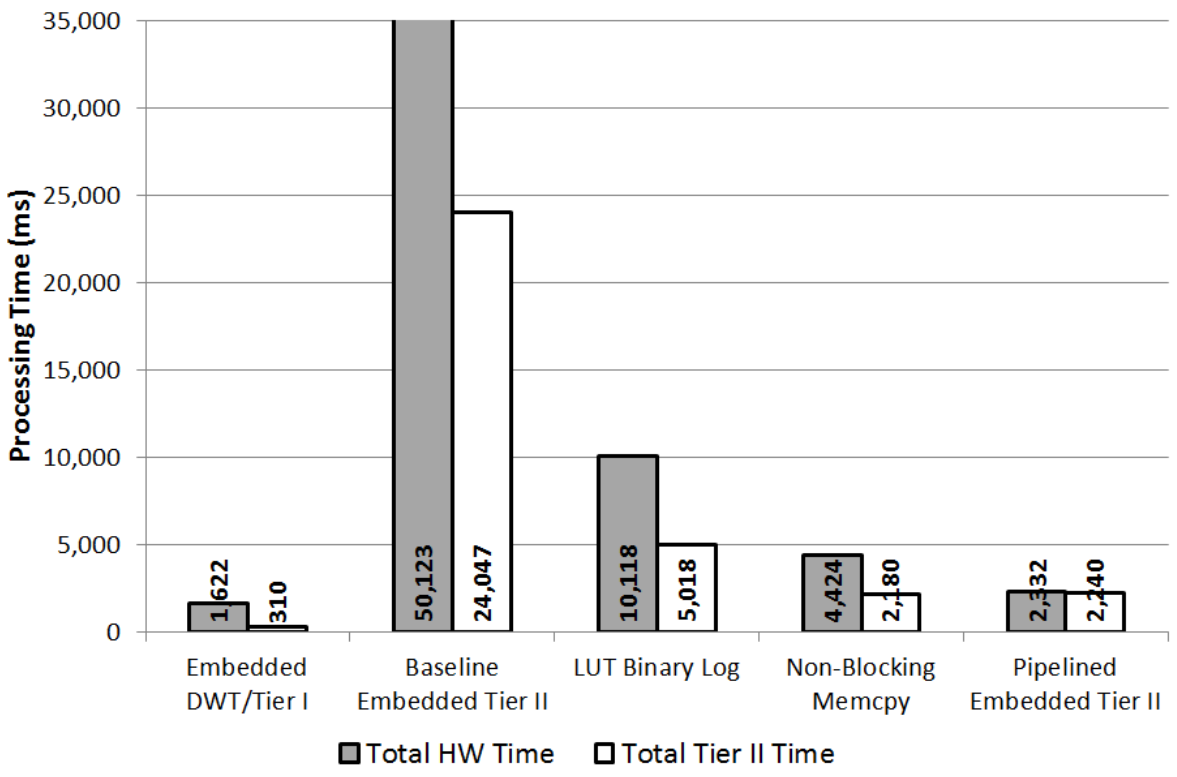


Figure 6.7: Total processing time comparison between “Existing” and the optimizations made to the embedded Tier II system (for “Aerial”)

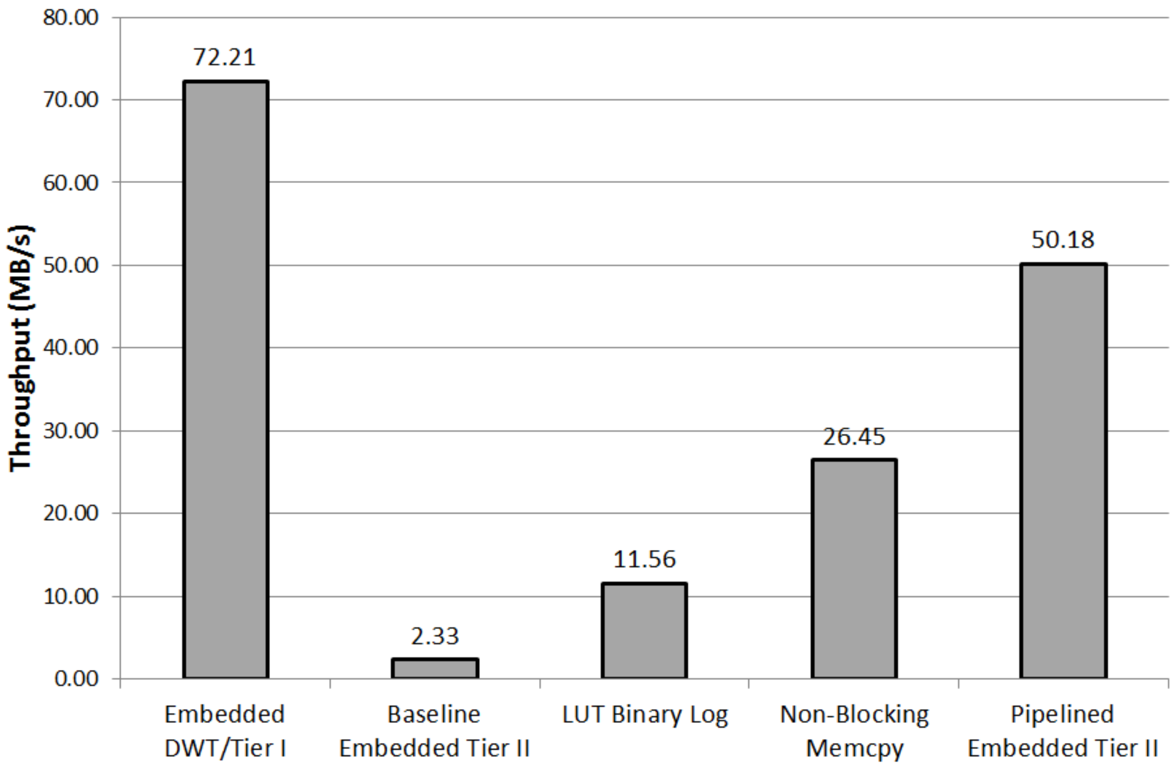


Figure 6.8: System throughput comparison between “Existing” and the optimizations made to the embedded Tier II system (for “Aerial”)

CHAPTER 7

FUTURE WORK AND CONCLUSIONS

7.1 Future Work

While the various optimizations made to the embedded Tier II design have yielded performance increases over 90% compared to the baseline embedded Tier II implementation (see Figures 6.6 and 6.8), "Existing" is still 30% faster than the pipelined embedded Tier II design (for "Aerial"). However, there is still room for increased system throughput. While optimizations could be made to the NIOS II system to improve the Tier II processing time, such as a faster DDR2 memory controller or utilizing more on-chip memory, this future work proposal is focused on an architectural improvement.

From the performance results for "Aerial" shown in Figure 6.7, it is clear that Tier II processing is the bottle-neck in the system. With 25 block coders, "Existing" can process "Aerial" in 1.6 seconds while the pipelined embedded Tier II takes 2.3 seconds, which is slightly longer than the total Tier II processing time. Also note that performance analysis of "Existing" shows that it is capable of supporting even more block coders for further increased performance. However, this does no good if Tier II is unable to keep up.

One solution to the under-utilization of the Tier I block coders is to add a second NIOS II system to the design to perform Tier II in parallel. Since the PROCeIV 530 has an additional memory bank which is unused in the design presented in this paper (Bank C), the second NIOS II system would be a duplicate of the system shown in Figure 5.9, complete with its own DDR2 memory controller. Tiles processed by Tier I would be alternately placed into two different MegaFIFOs, each feeding one of the two Tier II modules. A comparison of the block diagrams showing the data flow through the single and dual Tier II module designs can be seen in Figure 7.1

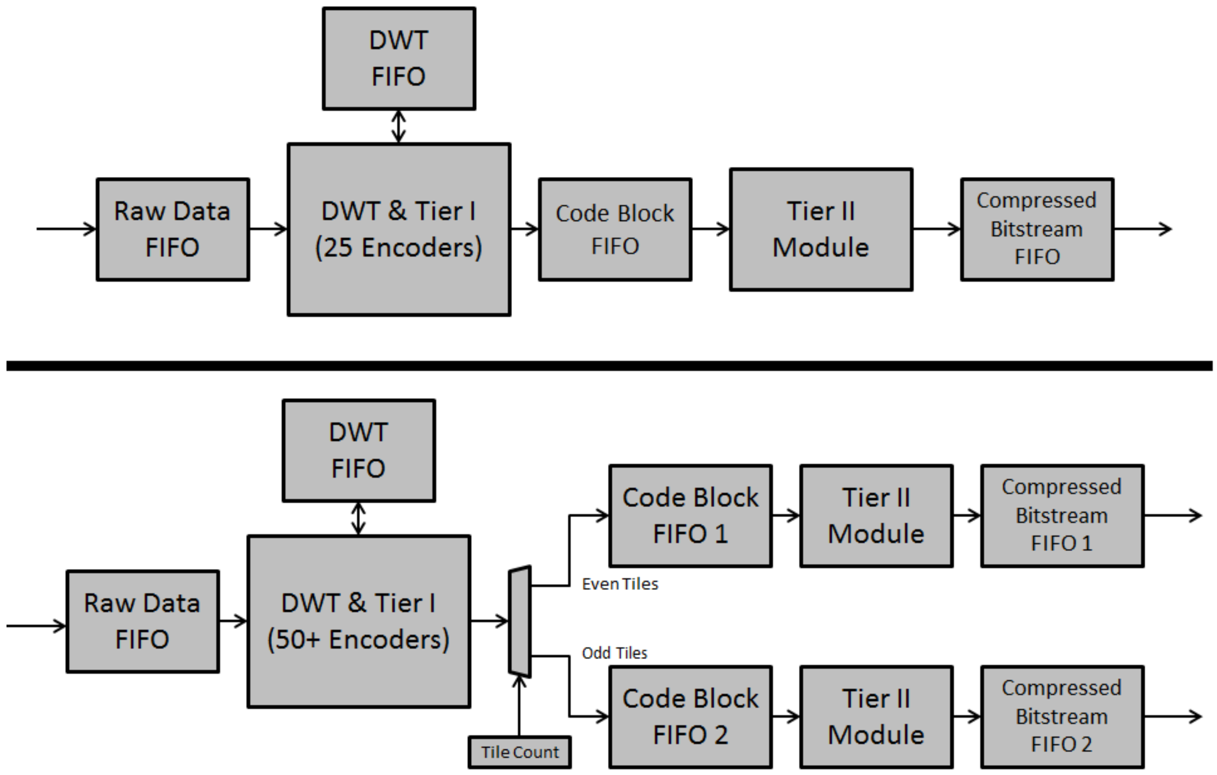


Figure 7.1: Block diagram comparison between the embedded Tier II design with a single NIOS II system (top) and the proposed embedded Tier II design with dual NIOS II systems (bottom)

The system proposed would be fairly easy to implement. A simple mux for the Tier I output control signals would be controlled by a tile counter. Even tiles would be placed into one FIFO while odd tiles would be placed in the other. The two Tier

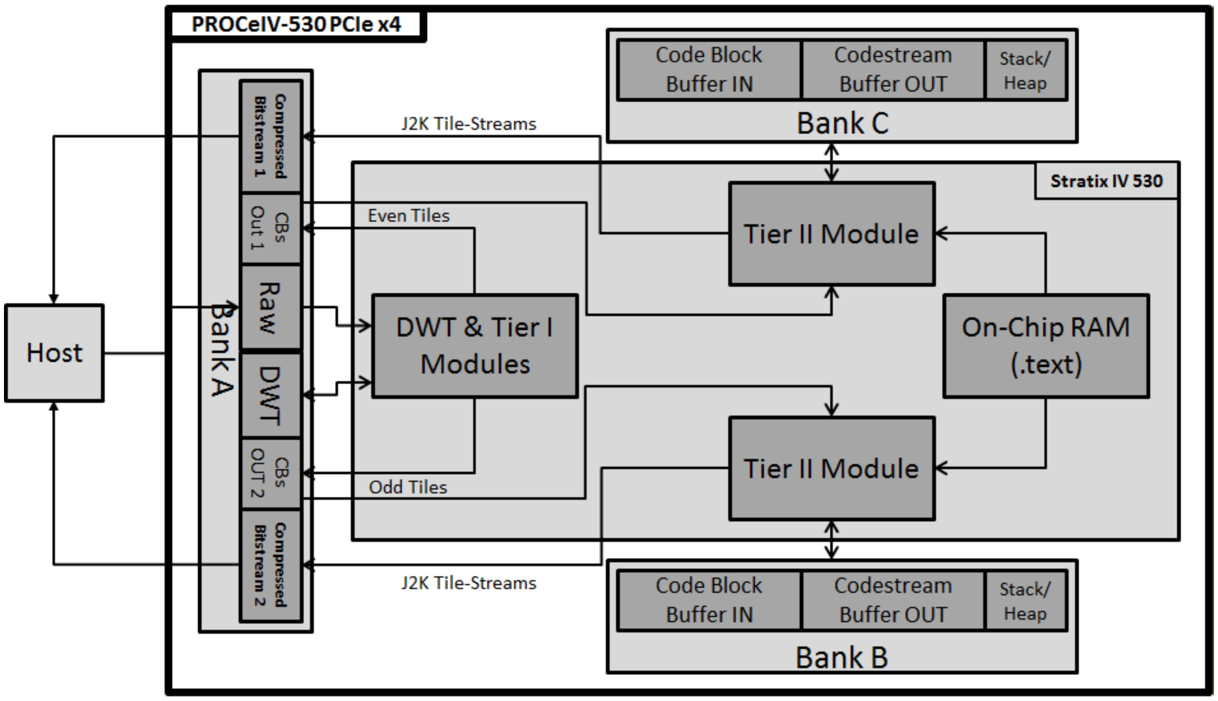


Figure 7.2: Architectural block diagram showing the proposed dual NIOS II system on the target platform

II modules would have no knowledge of each other and each would have a separate output MegaFIFO for DMA back to the host. The host would then be responsible for creating the final JPEG2000 bitstream from the two received tile buffers. An architectural block diagram showing the proposed system on the target platform can be seen in Figure 7.2. The block diagram shows the addition MegaFIFOs needed in Bank A as well as the interconnections between the modules. Most importantly, it shows that the on-chip memory used to store the instructions for execution could be shared between the two systems, as both would be running identical code. This would save valuable on-chip memory, which is a limiting factor when determining how many Tier I block coders can be used. Assuming that the DWT/Tier I module scales linearly with respect to the number of Tier I block coders, the proposed dual

NIOS II design would provide a 50% performance increase over the single embedded Tier II design.

7.2 Conclusions

The objective of this thesis is to realize a fully embedded JPEG2000 encoder on an FPGA. To accomplish this goal, existing embedded DWT and EBCOT Tier I modules are coupled with an embedded processing core to perform EBCOT Tier II coding in order to form a full JPEG2000 encoder. The design is implemented on a high performance PCIe FPGA card from GiDEL featuring a single Altera Stratix IV FPGA. The embedded processing core chosen is Altera's NIOS II soft-core processor and was leveraged to perform Tier II processing on the back end of the embedded pipeline.

The baseline embedded Tier II design yields a system throughput of only 2.33 MBytes per second, far below the throughput of "Existing". To increase the throughput of the design, optimizations are made to both the Tier II algorithm as well as the NIOS II processing system. Two different approaches to minimizing the computational cost of floating-point calculations are taken. First, custom FP instructions are added to the NIOS II core, which yields a 58% decrease in Tier II processing time as compared to the baseline embedded Tier II implementation. Then, the floating-point instructions are removed entirely in favor of a lookup table implementation of the floored binary logarithm used. This implementation yields an additional 30% decrease in Tier II processing time. Additionally, two system level optimizations are made to increase the Tier II processing time. First, tightly-coupled data memory is added to the NIOS II system to provide guaranteed low latency access to some critical data which yields a 16% decrease in Tier II processing time. Then, an additional DMA controller is added to the system to provide non-blocking memory transfers. This yields a 67% decrease in Tier II time. Overall, these optimizations yield an overall reduction in Tier II processing time by 92%.

In addition to the NIOS II system optimizations, a pipelined version of the design is implemented by decoupling the Tier I and Tier II modules' inputs and outputs. Instead, a large FIFO is inserted between the two modules. This enables the Tier I block coders to operate at full speed, unrestricted by the Tier II module's relatively slow input interface. This yields a 47% increase in total system throughput. Overall, the final pipelined embedded Tier II design is 95% faster than the initial baseline implementation and capable of processing 50 MBytes per second. However, this is still 30% slower than "Existing". Therefore, a modified architecture featuring two identical Tier II modules is proposed with the potential for a 50% increase in system throughput.

BIBLIOGRAPHY

[1] T. Acharya and P. Tsai. *"JPEG2000 Standard for Image Compression: Concepts, Algorithms and VLSI Architectures"*. John Wiley & Sons, Inc., Hoboken, New Jersey, 2005.

[2] C. Christopolous, A. Skodras, and T. Ebrahimi. "The JPEG2000 Still Image Coding System: An Overview". In *IEEE Transactions on Computer Electronics*, volume 46, pages 1103–1127, November 2000.

[3] Altera Corporation. *"SOPC Builder User Guide"*, Dec. 2010.

[4] Altera Corporation. *"Avalon Interface Specifications"*, May 2011.

[5] Altera Corporation. *"NIOS II Custom Instruction User Guide"*, Jan. 2011.

[6] Altera Corporation. *"NIOS II Processor Reference Handbook"*, May. 2011.

[7] Altera Corporation. *"NIOS II Software Developer's Handbook"*, Feb. 2011.

[8] D. Cruz and T. Ebrahimi. "An Analytical Study of JPEG2000 Functionalities". In *Proceedings of ICIP 2000*, 2000.

[9] GiDEL. *"ProceIV Data Book"*, May 2011.

[10] R. Gonzalez and R. Woods. *"Digital Image Processing: Third Edition"*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[11] "ISO/IEC 1.29.15444-1". *"JPEG 2000 Part I Final Committee Version 1.0"*, Sept. 2004.

[12] D. Janssens. "Tag Tree Implementation in C". Source Code, 2002.

[13] GiDEL Ltd. "About GiDEL", 2012.

[14] D. Mundy. "The Study and HDL Implementation of the JPEG2000 MQ Coder". Master's thesis, University of Dayton, May 2007.

[15] D. Taubman and M. Marcellin. *"JPEG2000: Image Compression Fundamentals, Standards and Practice"*. Kluwer Academic Publishers, Norwell, MA, 2002.

[16] G. Yang, N. Zheng, C. Li, and S. Gou. "Extensible JPEG2000 Image Compression Systems". In *IEEE National Conference on Industrial Technology*, pages 1376–1380, 2005.