An FPGA-based Run-time Reconfigurable 2-D Discrete Wavelet Transform Core

Jonathan B. Ballagh

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

June 2001
Blacksburg, Virginia

Keywords: FPGA, Reconfiguration, Wavelets, JBits

An FPGA-based Run-time Reconfigurable 2-D Discrete Wavelet Transform Core

Jonathan B. Ballagh

(ABSTRACT)

FPGAs provide an ideal template for run-time reconfigurable (RTR) designs. Only recently have RTR enabling design tools that bypass the traditional synthesis and bitstream generation process for FPGAs become available. The JBits tool suite is an environment that provides support for RTR designs on Xilinx Virtex and 4K devices. This research provides a comprehensive design process description of a two-dimensional discrete wavelet transform (DWT) core using the JBits run-time reconfigurable FPGA design tool suite. Several aspects of the design process are discussed, including implementation, simulation, debugging, and hardware interfacing to a reconfigurable computing platform. The DWT lends itself to a straightforward implementation in hardware, requiring relatively simple logic for control and address generation circuitry. Through the application of RTR techniques to the DWT, this research attempts to exploit certain advantages that are unobtainable with static implementations. Performance results of the DWT core are presented, including speed of operation, resource consumption, and reconfiguration overhead times.

Acknowledgments

I would first like to thank my advisor Dr. Peter Athanas for the support and guidance he has given me during my two years as a graduate student. His enduring patience, motivation, and creativity have made it a pleasure to work for him in the CCM lab at Virginia Tech. I would also like to thank Dr. Mark Jones for providing me with advice and an endless supply of new ideas, for both this research and other projects I have been a part of in the lab. Both Dr. Athanas and Dr. Jones have inspired and challenged me to push the boundaries of what is possible in the field of Configurable Computing.

I extend a sincere thanks to Dr. Amy Bell and her students who were patient enough to explain many of the difficult wavelet concepts I would not have grasped otherwise. I also thank them for the help with Matlab. Their assistance and help on this research is greatly appreciated.

To everyone on the Loki team at Virginia Tech and Xilinx, I am grateful for the support and help you have provided me with during this research. It has been a great pleasure working on the team for the last two years. I am particularly thankful to Cameron Patterson and Steve Guccione for the unique internship opportunity and guidance they provided me with last summer. I would especially like to thank Eric Keller for the ideas, motivation, and assistance he has contributed (and for keeping me off the streets during my stay in Boulder). Thanks to Scott McMillan for his effective and efficient work in helping me bring the RTR I/O JBits classes to fruition. A special thanks to Phil James-Roxby and Brandon Blodget for supporting the EDIF and XDL tools I used to obtain performance results. Thanks to Dan Downs for keeping me up to date with JBits release information.

To all my friends in the lab who helped with this research, I sincerely thank you. Thanks to Jon Scalera, who is always willing to take time out to lend a hand. I would like to thank Santiago Leon, Jim Hendry, and Scott Harper for their help with the Slaac1V board. Thanks to Jae Hong Park for all the wavelet discussions. To my fellow colleagues on the Loki team here at VT: Jing Ma, Neil Steiner, and Dennis Goetz; it has been a pleasure working with you and I wish you the best of luck in your future careers. The CCM lab is full of extremely talented individuals who will find success at every corner.

Lastly, I dedicate this thesis to my family and my fiancée, Lisa.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 FPGAs and Run-Time Reconfiguration

Hardware applications that utilize run-time reconfiguration (RTR) and parameterization have been the topic of considerable research in the field of configurable computing [29-32]. Run-time reconfigurable designs are capable of dynamically tailoring circuit routing and logic to better suit the current application task. Run-time parameterizable (RTP) systems, on the other hand, define logic and routing just prior-to run-time, based on a set of parameters that define circuit behavior. Research into field programmable gate array (FPGA) RTR application design, however, has been hindered by the lack of software tools that enable true RTR design for these devices.

FPGAs provide a hardware environment in which physical logic and routing resources can be reprogrammed by a configuration bitstream in order to perform a specific function. As a result, they provide an ideal template for dynamic circuit specialization and logic reconfiguration. With the exception of research on the Xilinx XC6200 device [14,15], access to configuration bitstreams of commodity devices has been restricted. Designers, therefore, have been required to generate configuration bitstreams using traditional vendor tool flows.

The vendor tools attempt to provide optimum speed and efficiency for static circuits by using complex routing and placement algorithms. With the contemporary tool flow, one must typically cope with lengthy synthesis and bitstream generation times. For static designs, this time penalty is usually acceptable. These times are impractical for RTR designs that require fast bitstream reconfigurations. As a result, traditional FPGA tools and design methodologies have prohibited true RTR designs, simply because they could not meet the reconfiguration time constraints required by the design.

The development of the JBits API has opened access into the configuration bitstream of Xilinx 4K and Virtex FPGA devices [2]. Using the JBits API, the designer can bypass the logical synthesis and physical implementation steps, allowing rapid bitstream reconfigurations. When compared with ASICs, JBits has been used to create higher performance circuits in FPGAs using RTR [18]. The JBits API, therefore, provides the necessary tools for implementing an effective FPGA-based run-time reconfigurable and parameterizable design.

1.2 Wavelets

Wavelets have been receiving increased attention, mainly due to their wide appeal for a variety of applications, ranging from image editing and compression to electrocardiogram analysis. As today's applications become more graphically intensive, the need for efficient image compression techniques has grown extensively. This need is most apparent in the case of network applications, where bandwidth limitations are present. Before an image can be compressed, it must first be transformed into a domain where higher compression ratios can be achieved, rather than applying the compression algorithms directly to the original image itself. JPEG, the predominant image compression format for the World Wide Web, uses the discrete cosine transform as its transform technique [33].

The wavelet transform improves on the discrete cosine transform (DCT), however. The DCT divides a signal's frequency content into fixed, equal bandwidth partitions. By providing only the frequency content of the signal, the DCT is unable to represent non-stationary signal properties in the transform domain. Images are often non-stationary. To circumvent this problem, the JPEG algorithm uses a block-based transform in which the DCT is applied to image block partitions separately. Because each block is transformed on an individual basis, there are often inefficiencies between blocks. The wavelet transform rectifies this problem by providing a representation of a given signal in both time and scale domains [8]. The discrete wavelet transform (DWT) provides coarser resolution at low frequencies, while providing finer resolution at higher frequencies (Figure 1.1). The DWT can transform the entire image and preserve non-stationary information, rather than using block partitions. Because of this, the wavelet transform has been shown to significantly outperform the DCT in image compression applications, leading to their inclusion in the JPEG 2000 standard [5]. The DWT is a reversible transform and can be either a “lossly” or “lossless” process depending on the selection of wavelet. Besides image compression, wavelets are also aptly suited for image editing and progressive transmission applications since they provide a multiresolution decomposition of a signal.

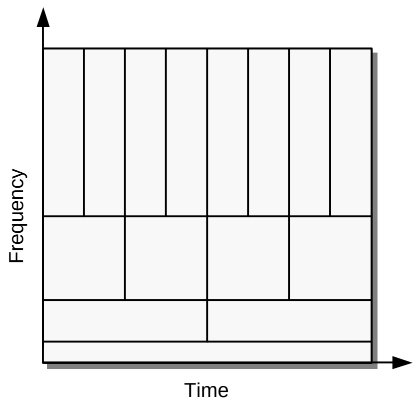


Figure 1.1: DWT frequency partitioning based on time.

To date, the majority of wavelet transform implementations have existed in software. Software provides greater flexibility of operation, however, performance is often too sluggish for high-end multimedia applications such as video and sound compression. It is often desirable to offload repetitive computations, such as compression coding and signal filtering, to a hardware accelerator in order to free the processor for other tasks.

1.3 Justifying Run-Time Reconfiguration

The purpose of this research is to explore the advantages of RTR and RTP when applied to the DWT algorithm. For the design of an RTR application to be justifiable, it should exhibit clear advantages over a similar, static circuit. Although a number of architectures for static ASIC-based wavelet transform architectures have been explored [6,7], they offer little in the way of operation customization. Advantages that can be exploited through RTR include circuit speed increases through decreased latency or increased clock frequency, and decreased resource consumption when compared to the static implementation counterpart. These advantages were considered in relation to the discrete wavelet transform core implementation.

The DWT relies on digital filtering to perform the transform operation. Although parameterization of the control logic is possible and even required to guarantee design portability between devices, the speed advantages gained by applying RTR are negligible. Therefore, focus was placed on the advantages gained from dynamic specialization of the filter circuitry.

The primary advantage of specializing filters at run-time is that the coefficients can be hard-coded into the circuit. Although a static transform implementation could also be implemented that would provide similar features, filter coefficients would have to be clocked into registers during an initialization procedure. This would not only require extra host interfacing circuitry to provide access to FPGA registers, but would also complicate the control logic required for circuit operation. Guccione notes that the elimination of system interface circuitry through RTR results in lowered system costs if a smaller device can be used on the reduced circuit [26]. Another advantage is that specialized filter circuitry can be defined that optimizes latency and resource usage based on a set of filter coefficients. It was perceived that the possible advantages gained by RTR warranted the design of a two-dimensional discrete wavelet transform core.

1.4 Shared Processing Environment

The use of JBits for RTR requires a separate "host" processor and Java Virtual Machine (JVM) to execute the Java classes that perform bitstream reconfigurations. It was desirable, therefore, to have the wavelet transform application benefit from the required PC-FPGA shared processing environment. More specifically, the DWT core could be used to accelerate computationally intensive signal processing tasks offloaded from the host processor.

Image processing applications have been shown to benefit from shared processing environments, in which the FPGA is utilized as a co-processor [16,17]. This concept can be extended to utilize RTR for the wavelet transform core, in which the "host" process defines a specialized circuit coprocessor instance to accelerate computation of the current image-processing task.

Designing specialized circuitry is useful in wavelet-based image coding applications, where the performance of the coding algorithm is dependent on the image itself [19]. The optimizations occur through the selection of wavelet family, directly affecting the signal-to-noise ratio for reconstructed images in the case of image compression. In this situation, the host process can select a wavelet that provides optimum performance for an image, or video stream, and tailor an FPGA circuitry accordingly. It is feasible, therefore, for the coder to store the circuit optimizations as instructions for circuit rebuilding, and transmit them with the compressed image as a header. In other words, the decompression circuit is passed along with the image data. The host on the decoder end can construct a specialized circuit based on the circuit instructions in the header, and decompress the image. For wavelets, these instructions are defined in terms of filter coefficients. These concepts can be extended to video compression, where the compression ratio may need to be periodically adjusted to provide necessary throughput requirements. Again, the selection of filter coefficients can alter the compression ratio.

1.5 JBits

JBits [2] is a Java-based API that enables direct modification of Xilinx 4K and Virtex device bitstreams by providing access to all FPGA SRAM-based configuration resources. Sample configuration resources include configurable-logic block (CLB) elements, routing switches, input/output blocks (IOB), block RAM (BRAM), and routing MUXes. A Java JBits class and set of configuration constants represent these resources. The resource classes provide methods for configuring the functionality of the corresponding FPGA resource by setting the SRAM values to a particular configuration constant.

Each resource is identified by a series of indices that identify its position in the resource array. The majority of reconfigurable resources are located within the CLB array. The resources in this array are indexed by the row and column of the CLB tile that the resources are located in. JBits locates the origin of the CLB array in the lower-left corner of the device.

A typical JBits application accepts a bitstream as input, modifies and analyzes the appropriate configuration resources as necessary, and then saves the modified bitstream to file. The JBits API also provides the ability to perform readback of selected resources from the FGPA during execution, thereby providing a powerful debug utility at the bitstream level.

Figure 1.2 provides sample JBits code in which F and G look-up tables (LUTs) are configured. The JBits *Expr()* method converts a string representation of a Boolean equation into a sixteen-bit integer vector that can be loaded into a LUT. In this case, the four F-LUT input pins are being *AND'd* together, while the four G-LUT input pins are *OR'd* together. The "~" character negates the expression. The negation is required since LUT values are inverted when written to the configuration bitstream.

```
/* define row and column values */
int row = 5; int col = 4;

/* define logic function for F LUT */
int[] F_LUT_Vals = Expr.F_LUT("~(F1&F2&F3&F4)");

/* define logic function for G LUT */
int[] G_LUT_Vals = Expr.G_LUT("~(F1|F2|F3|F4)");

/* set the F LUT value for slice 0 */
jbits.set(row, col, LUT.SLICE0_F, F_LUT_Vals);

/* set the G LUT value for slice 1 */
jbits.set(row, col, LUT.SLICE1_G, G_LUT_Vals);
```

Figure 1.2: Configuring F and G look-up tables with JBits

Using this API, bitstreams can be modified in relatively short amounts of time when compared to the time required for traditional FPGA vendor tools to produce bitstreams. Whereas synthesis tools can take on the order of a few hours to complete, a JBits stand-alone application can produce bitstreams in a matter of seconds. These times can be further reduced if only a portion of the bitstream requires modification, as is the case with partial reconfiguration.

Although direct JBits calls occur at a very low level, higher-level tools have been built on the foundation of configuration calls using the object-oriented model afforded by the Java. These tools have transformed JBits into a powerful FPGA design and debug environment. Chapter 2 discusses the higher level tools built on the JBits foundation that are pertinent to the implementation of the wavelet transform system.

1.6 Thesis Contributions

The most significant contribution of this thesis is the presentation of run-time reconfigurable and parameterizable two-dimensional discrete wavelet transform core. Although the design was tested and run on the Slaac1V PCI-accelerator platform, the design can be easily ported to other platforms. The implementation of the wavelet transform core required the development of a core library. Among the most pertinent of these cores is the RTP *FIRFilter* core, which offers full parameterization in terms of coefficient precision and number of taps. Several of the cores in the library have uses outside of the wavelet transform implementation and are included in the JBits release distribution.

In addition, this research provides a means for automatically interfacing a JBits design to IOBs. A designer was previously required to determine the JBits *Pin* resources that attached to each signal end-point of the design. Because a large number of IOBs were often required, this was a tedious and error-prone process. Hard coding IOB routes also restricted an IOB dependent core to a particular FPGA device. For this reason, a utility was created that automated the process of interfacing a top-level RTP core to IOBs based on the specifications given in a standard user constraint's files.

1.7 Organization

Chapter 2 provides a brief background on the tools and subject material of this research. More specifically, JBits, the discrete wavelet transform theory, and the Slaac-1V board are introduced. The chapter also provides a brief survey of existing DWT implementations. Chapter 3 describes the design and implementation of the core library that provided the design building blocks. Understanding the operation of the smaller cores is critical in understanding the composition and operation of the larger wavelet transform system. After a discussion of the DWT core library, the overall two-dimensional discrete wavelet transform core is presented in Chapter 4, including the filter bank design and address generator logic. Chapter 5 describes how the core was simulated using a behavioral Java representation of the hardware. The chapter continues by describing the bitstream level debugging process used during core design. Chapter 6 introduces a set of JBits classes that automate the interfacing of cores to FPGA I/O resources. The chapter also describes how the discrete wavelet transform core was interfaced to the Slaac1V board. Chapter 7 describes the operation of the "host" PC application that controls operation and reconfiguration of the target FPGA platform. Chapter 8 presents the performance results of the core. The research is summarized along with a postulation of future work in Chapter 9.

Chapter 2

Background

This chapter introduces the software, hardware, and theory used throughout this research. The JBits design environment is presented as a series of tools that aid RTR design. The run-time parameterizable core specification is introduced separately, as it defines a structural hardware description language that is at a higher level of abstraction than the original low-level JBits functions. A brief introduction is provided on IOB routing, which is necessary in understanding how the DWT implementation was interfaced to FPGA I/O resources. The DWT theory is explained in relation to the implementation in FPGA hardware. The specifications of the Slaac1V PCI accelerator board are also discussed. The chapter concludes with the introduction of several existing DWT implementations on different hardware devices.

2.1 JBits Environment

The JBits environment provides users with the tools needed to implement RTR and RTP designs for Xilinx Virtex and 4K series FPGAs. These tools aid in the coding, debugging, simulation, and validation aspects of the design process. Each component offers different levels of abstraction, allowing the design process to take place at the level of detail required by the designer. Figure 2.1 shows a component view of the JBits environment. The following sections discuss each component in greater detail.

JRoute – The JRoute API is a run-time reconfigurable router for use with JBits designs [22]. Choosing individual routing resources, selecting a route template, or using the auto-router are the supported techniques for specifying routes. These three routing methodologies allow the designer the level of routing abstraction he or she desires. At the lowest level, the user can specify a list of all individual routing resources that define a point-to-point connection. At a higher level of abstraction, the user can select between templates that dictate which routing resources should be considered by JRoute when connecting endpoints. An auto-router provides the highest level of abstraction; however, it offers the user little control over how the connection is defined.

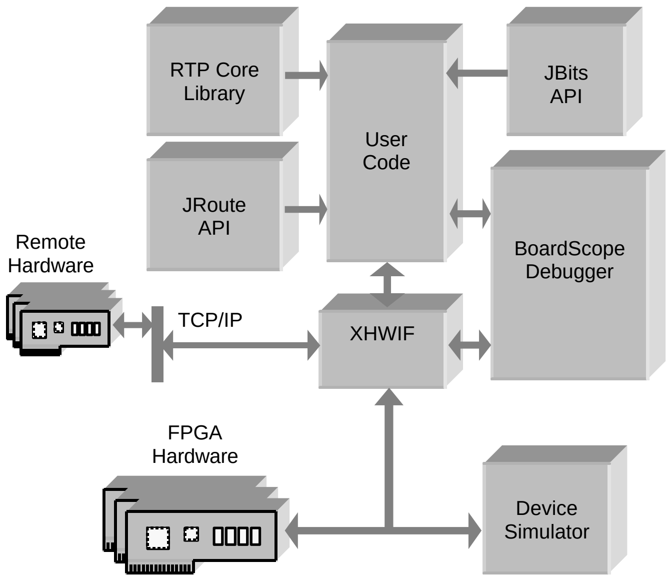


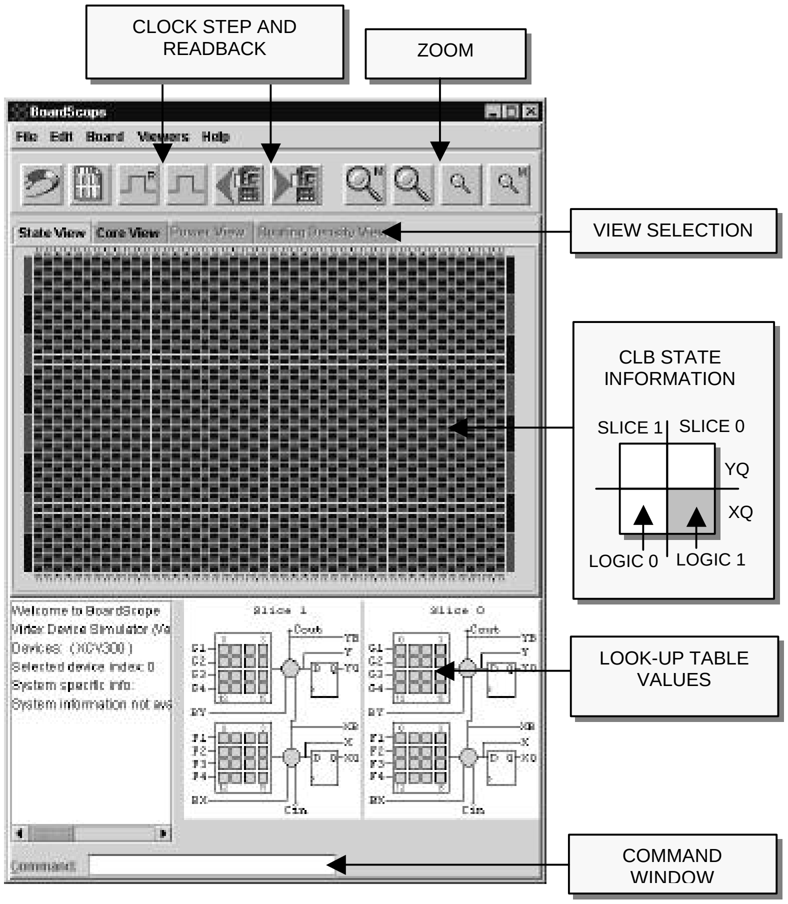
Figure 2.1: JBits environment

XHWIF – The **X**ilinx **H**ard**w**are **I**nter**f**ace API provides generic; non-device specific calls for communicating with FPGA based hardware platforms. XHWIF methods let the user step the clock, reset the device, read and write to memory, load bitstream configurations, and perform device readback. Communication with the FPGA is done through the XVPI interface [21]. Using the Java Native Interface (JNI) to perform non-JVM functions, an API specific to any Xilinx Virtex/4K FPGA hardware platform can be ported to XHWIF, assuming the device API provides basic readback and clock stepping capabilities. XHWIF provides a single communication interface for a variety of FPGA based hardware platforms.

VirtexDS – The VirtexDS [20] is a Java based simulator for Virtex devices. The simulator models the system level hardware functionality of the Virtex family. The specific Virtex device, along with a global clock identifier parameterizes the VirtexDS. The behavior of RTR designs is accurately depicted since the hardware itself is being simulated, rather than using the fixed circuit netlist approach often used by traditional simulators. The simulator interface has been ported to XHWIF, allowing communication to take place with the simulator in the same manner as physical hardware. The interface also allows for seamless integration with the BoardScope debugging environment. As a result, designs can be safely validated in software before being tested on the physical FPGA.

BoardScope – BoardScope is a graphical FPGA debugging environment that operates at the bitstream level. Communication with hardware takes place using an underlining XHWIF layer. This allows BoardScope to use any hardware platform that has been ported to XHWIF. The BoardScope environment displays FPGA read-back information in a graphical context. This information includes CLB flip-flop states, LUT configurations, BRAM data, and IOB register states. The debugging process is started with a connection to a supported FPGA hardware platform, either locally, or remotely using a network connection. After connecting to desired hardware, bitstreams can be loaded on the device. Debugging in BoardScope allows the user to switch between hardware platforms, including the VirtexDS. Figure 2.2 provides a screen shot of the BoardScope debugging environment, along with captions that explain features.

The debugging environment features different graphical views in which the operation of the FPGA hardware is shown in different contexts. Possible views include State, Core, Power, and Routing Density. As an example, the State view provides a graphical display of read back state information. The main grid representing the CLB layout shows the state information of all four flip-flops within a single grid square. Clicking on a CLB grid square causes the look-up table configuration to be displayed in the graphical CLB viewer. These views create a robust debugging environment for RTR applications.



```
Welcome to BoardScope
Virtex Device Simulator (Ve
Devices: (XCV300 )
Selected device index: 0
System specific info:
System information not ava
```

Figure 2.2: BoardScope graphical debugging environment

2.1.1 The Run-Time Parameterizable Core Specification

The JBits run-time parameterizable (RTP) core specification [3] provides a means for abstracting away the low level JBits configuration calls, thereby creating an environment similar to traditional hardware design languages (HDL)s. By taking advantage of the Java object-oriented paradigm, bitstream configuration calls are encapsulated by low-level primitive cores, such as MUXes, LUTs, and gate-level logic. These primitive cores can then be used together to create higher abstraction cores.

The distinction between JBits RTP cores and cores used in traditional structural HDLs is that each core must be physically placed on the FPGA device during implementation. The RTP core template specification provides two methods for placing cores. The designer can choose between placing cores relative to other cores and explicitly defining the core location within the CLB grid.

The *Place* class allows cores to be placed in relation to previous child core placements and parents' core boundaries. The class defines a series of placement directives such as ABOVE_PREV_ALIGN_LEFT, which determines the placement of the current child core relative to the placement of other child cores. Other included placement directives, such as LOWER_LEFT and LOWER_RIGHT allow the working child core to be aligned with the parent core's boundaries. The core is placed when a child core is added to the parent using the *addChild()* method.

As an alternative to using *Place* directives, core offsets can be calculated and defined explicitly by the designer. This is accomplished through the use of an *offset* class. Every RTP core has an *offset* class as a member field. The *offset* class allows the designer to anchor an RTP core to the desired location within the FPGA CLB array. A core offset is defined in terms of both a horizontal and vertical offsets. These offset values are defined relative to the offset of the core's parent. The relative offset is measured from the origin (0,0) located at the lower-left corner of the parent core (Figure 2.3).

The sizing and placement of an RTP core depends on the core's horizontal and vertical granularity. The granularity of a core specifies the coordinate grid in which the core is aligned. The three granularities include CLB, slice, and LE granularities. LE provides logic element alignment resolution, where a logic element is comprised of a LUT and flip-flop. Providing different granularities allows the designer to make more efficient use of FPGA resources.

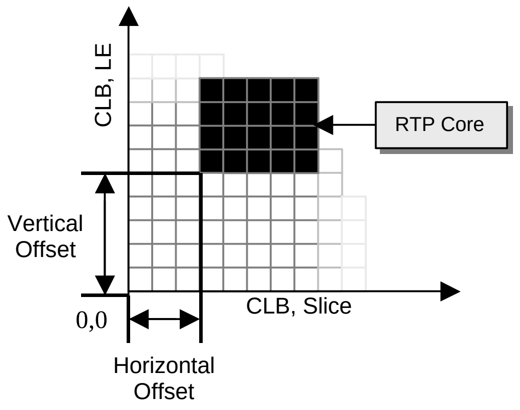


Figure 2.3: FPGA CLB array with offset origin located in lower-left corner

Net and bus signals provide the interconnections between core interfaces. Both *Net* and *Bus* classes extend from the base class *Signal*. The *Bus* class is a collection of *Net* signals. A core's interface is defined by a series of I/O ports. A port is realized in JBits by the *Port* class. A *Port* allows for both internal and external signal connections. External signals provide connections between cores. Child cores within a parent core, on the other hand, are connected together by internal signals. Figure 2.4 provides a graphical view of this relationship.

Because ports are only an abstraction, they do not explicitly bind signal routing to physical resources. At the primitive core level, signal sources and sinks must be bound to physical pins within the FPGA. The *Pin* class allows physical signal endpoints to be defined and instanced. A *Pin* is defined by four parameters: a tile type, two location coordinates, and the JBits resource to use within a particular tile. The tile types provide routing to all FPGA resources, including CLB, IOB, BRAM, and clock DLLs. Pins are attached to ports using the *Port.setPin()* method.

By making the appropriate JRoute API calls, the bitstream is modified so that the core interconnections are implemented in physical hardware. The RTP core specification essentially transforms JBits from a low-level language to a high-level language using a structural design approach requiring physical placement of cores.

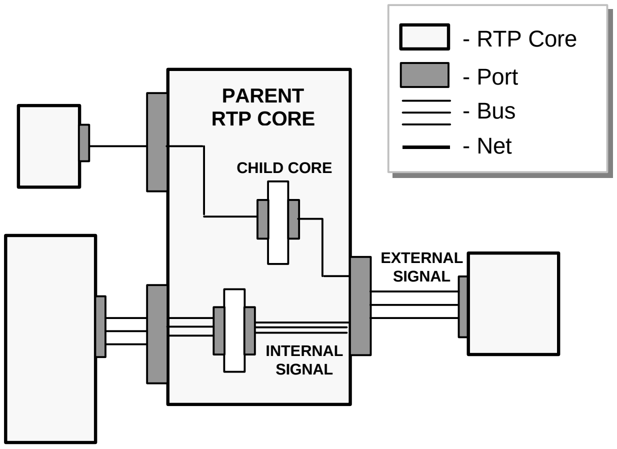
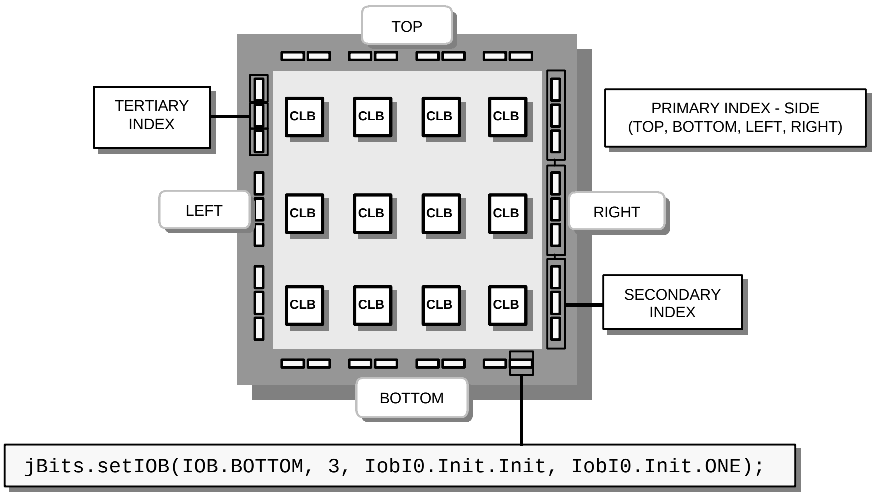


Figure 2.4: Diagram showing relation between internal and external signals

2.1.2 Routing to JBits IOB Resources

In order to route and configure an IOB using JBits, the specific JBits IOB resource class must first be determined. Three indices are required to distinguish a JBits IOB resource. An FPGA diagram showing the three indices is shown in Figure 2.5. The figure also provides an example of a JBits *setIOB()* call that sets the initial IOB register state to one. The creation of these indices was dictated by the arrangement of the configuration columns within the configuration bitstream [23]. The side on which the IOB is located provides the first index of the resource. The secondary index determines which IOB group is being accessed from the particular side. IOBs are grouped together in pairs for the top and bottom sides, and are grouped as threes for the left and right sides. The third index corresponds to the individual IOB that should be configured from a given group. After determining the three indices, IOB pins and configuration resources can be accessed accordingly.



```
jBits.setIOB(IOB.BOTTOM, 3, IobI0.Init.Init, IobI0.Init.ONE);
```

Figure 2.5: Example of JBits *setIOB()* method illustrated by an FPGA diagram showing side, secondary, and tertiary IOB resource indices.

2.2 The Discrete Wavelet Transform

The two-dimensional discrete wavelet transform (DWT) was implemented using JBits because it lent itself to a straightforward implementation, requiring relatively simple logic for control and address generation circuitry. The focus of DWT operation, the FIR filter, provided a core that could be implemented with a regular structure and layout. The DWT core also provided a good example of how a hierarchy of JBits RTP child cores can be used together to implement a larger system.

The DWT converts a signal from the time domain into the time-scale domain. Although the Fourier transform provides information about the frequency content of a signal, it does not preserve the time information that indicates when those frequencies occur. Based on a multiresolution analysis framework, the DWT captures both time and frequency information [24]. This section focuses on the DWT theory as it pertains to the implementation of the DWT in hardware.

The DWT of a signal can be computed by passing a signal through a two-channel filter bank (Figure 2.7). For orthogonal wavelets, these filters are responsible for dividing the signal bandwidth and are referred to as a quadrature mirror filter (QMF) pair, where the frequency responses are reflections of one another [9]. A QMF pair is comprised of a high-pass and low-pass filter. For example, the frequency response of the low pass synthesis filter derived from the Daubechies's N=3 orthogonal, compactly supported wavelet is shown in Figure 2.6. The low-pass filter coefficients [10] that correspond to this particular wavelet are given below:

$$\overline{h_0} = \ \ 0.035226$$
$$\overline{h_1} = -0.085441$$
$$\overline{h_2} = -0.135011$$
$$\overline{h_3} = \ \ 0.459878$$
$$\overline{h_4} = \ \ 0.806892$$
$$\overline{h_5} = \ \ 0.332671$$

The low-pass filter branch generates the average DWT coefficients of the signal, while the high-pass branch generates the detail DWT coefficients.

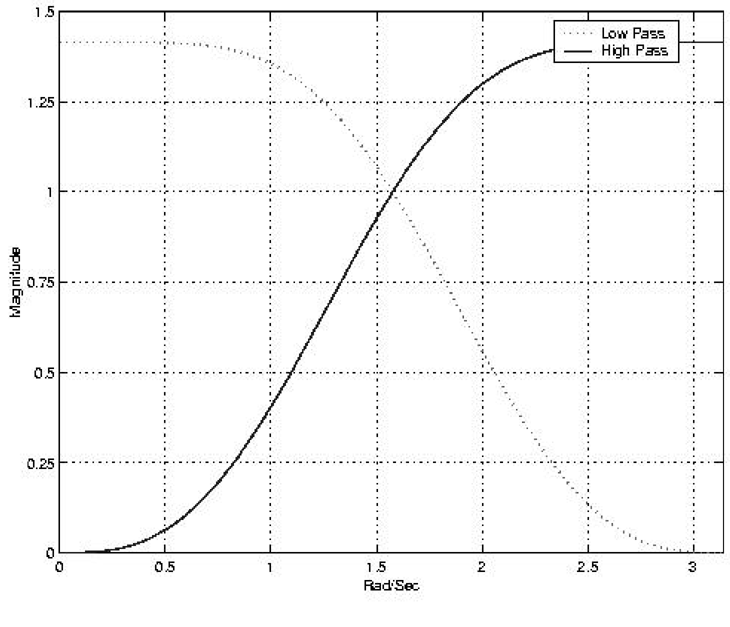


Figure 2.6: Frequency response of Daubechies's N=3 wavelet filter coefficients

The down sampled output of the high-pass filter constitutes the first octave output. As the filter pair processes the signal, the output is decimated by a factor of two. Filtering the signal controls the resolution of the signal, while the subsampling process controls the scale. Scale and frequency are inversely proportional such that higher frequencies correspond to lower (i.e. finer) scales, while lower frequencies correspond to higher (i.e. coarser) scales. Because the filters separate the frequency bandwidth, the filter pairs produce different resolutions, or levels, of detail.

Down sampling the filter output allows the output to be stored in the original signal space. The average coefficients are stored in the first half of the space, and the detail coefficients are stored in the latter half. The average coefficients are then processed again through the same set of filters producing a second set of average and detail coefficients. This DWT decomposition of the signal continues until the desired scale is achieved. Mallat illustrates this process in the Pyramid Algorithm [25].

Two-dimensional signals, such as images, are transformed using the two-dimensional DWT. The two-dimensional DWT operates in a similar manner, with only slight variations from the one-dimensional transform. Given a two-dimensional array of samples, the rows of the array are processed first with only one level of decomposition. This essentially divides the array into two vertical halves, with the first half storing the average coefficients, while the second vertical half stores the detail coefficients. This process is repeated again with the columns, resulting in four subbands within the array defined by filter output. Figure 2.7 shows a one level decomposition using the two-dimensional DWT. The filter output that results from two low-pass filters, labeled LL in Figure 2.7, is then processed again in the same manner. The process is repeated for as many levels of decomposition as are desired. The JPEG2000 standard specifies five levels of decomposition, although three is usually considered acceptable in hardware.

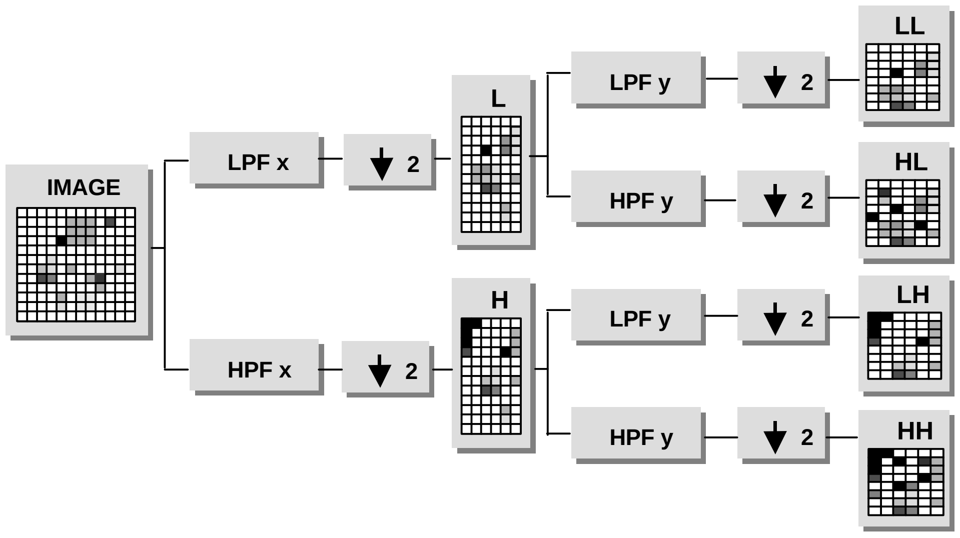


Figure 2.7: One-level decomposition using the two-dimensional DWT, where LPF x represents low-pass filtering of the image rows, HPF x represents high-pass filtering of image rows, LPF y represents low-pass filtering of image columns, and HPF y represents high-pass filtering of image columns.

2.3 Slaac1V Board

The Slaac1V PCI-board [1] was designed by the SLAAC group at the Information Sciences Institute-East. A block diagram of the Slaac1V board components and interconnects are provided in Figure 2.8. The board provides an ideal platform for reconfigurable computing applications with its usage of high-density FPGA devices that provide a template for reprogramable logic circuits. It also provides a unique test platform since a host computer can interact with the PEs through memory and FIFO accesses using the Slaac1V API. Since the API provides access to each PE's XVPI registers, partial reconfiguration and read back commands can be performed on the FPGA. The advantages afforded by the ability of the host to interact with the hardware made the Slaac1V the hardware platform of choice for this research.

Computation on the board takes place in one of three separate processing elements, or PEs. All PEs are implemented in hardware using Xilinx XCV1000 devices. The three PEs are designated as *X0*, *X1*, and *X2*.

Interconnects are provided between the PEs to enable communication between the devices. Communication between PEs can take place over the crossbar and across the direct connects to adjacent PEs. As shown in Figure 2.8, each PE has two 72-bit data ports that allow direct connections with the PE on either its left or right side. In this manner, the interconnect buses connect the PEs in a systolic "ring" arrangement. In addition to the direct connects, a 72-bit cross bar allows for unidirectional broadcast between PEs. As a result, there are three 72-bit data paths provided to connect a PE with other PEs.

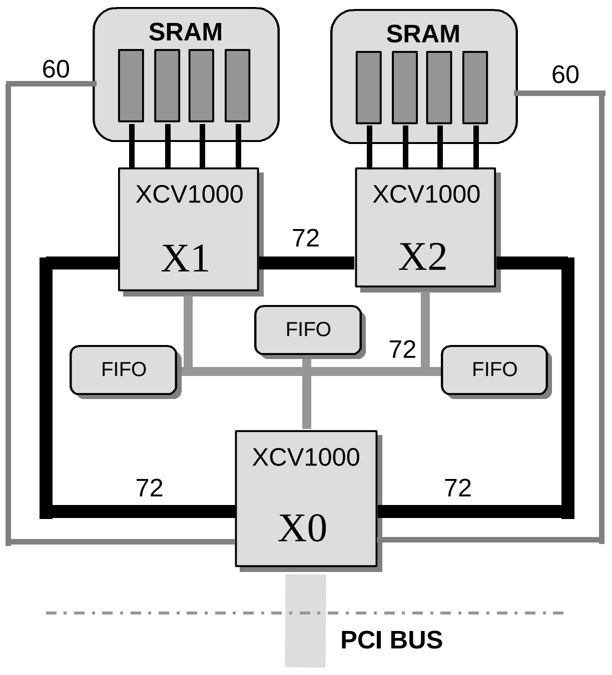


Figure 2.8: Slaac1V diagram

Designated SRAM and FIFO interfaces are provided to each PE as well, allowing data flow through the board. The Slaac1V board has a total of ten $36 \times 256K$ SRAM banks. PEs *X1* and *X2* have access to four $36 \times 256K$ SRAM banks. *X0* on the other hand, can access the remaining two SRAM banks, and can also access any of the other eight SRAMs. While having access to two of its own SRAMs, *X0* can swap one of its SRAMs with any of *X1*'s SRAMs and can swap its other SRAM with any of *X2*'s SRAMs. The host retains the ability to take control of a PE's memory at any time during operation.

2.4 Existing DWT Implementations

DWT designs have been implemented on a variety of devices, including standard commercial processors, DSPs, ASICs, and FPGAs. Although the slowest, software implementations for PCs provide the greatest deal of flexibility in terms of the selection of wavelet, bits per pixel, precision, number of transform levels, and image size. DSP implementations operate faster than generic processor designs since the instruction set is tailored to signal processing applications. Their flexibility is more limited than a generic microprocessor, however. ASIC designs provide the fastest operation since they use dedicated hardware to perform the transform. The tradeoff for speed, however, is lack of support for parameterization. FPGA implementations are located somewhere between ASICs and DSPs in terms of speed and parameterization capabilities. This section provides a brief survey of existing DWT implementations for processors, DSPs, ASICs, and FPGAs.

The majority of DWT designs are software-based algorithms. The JPEG2000 [5] and MPEG4 [34] standards both incorporate the DWT in their compression algorithms. The JPEG2000 uses a 9/7 wavelet to perform lossy compression, and uses a 5/3 wavelet for lossless compression. Both JPEG2000 and MPEG4 feature a variety of software implementations based on the underlying standards. The Matlab technical computing environment, designed by MathWorks [35], features a Wavelet Toolkit add-on that provides DWT support. The toolkit offers a wide variety of wavelets that can be used for signal processing and analysis computations.

Several DSP manufactures provide DWT implementations through source code or library functions. Motorola offers an application note for implementing a JPEG2000 version of the DWT for the StarCore DSP processor [36]. Texas Instruments includes functions for computing the vertical and horizontal wavelet transforms of image data in its image/video processing library for its TMS320C62x series DSP devices [37]. These functions allow specification of the filter bank coefficients, number of filter taps, and image size. The functions accept only 16-bit input and output data.

Several DWT ASIC architectures have been investigated [6,7]. Such research has attempted to find optimal architectures and efficient DWT designs that maximize parallelism, reduce the transform period, and reduce the amount of memory storage required for intermediate output. In the commercial sector, Analog Devices [38] has recently released the first JPEG2000 image compression chip. Analog Devices also produces the ADV601 Multiformat Video Codec chip that features a wavelet kernel. The wavelet coefficients are fixed for both devices.

FPGAs have experienced explosive growth in the number of system gates available for designs. The growth in size has allowed for the exploration of DWT implementations on FPGAs [39-41]. *Benkrid et al* have designed a two-dimensional biorthogonal DWT for the Xilinx XCV600E-8 device. This design uses a non-folded architecture to perform the DWT of an NxN image in N^2 clock cycles. The *Benkrid et al* design uses a biorthogonal 9/7 wavelet in the transform. Parameterization of the wavelet and image size requires a resynthesis of the design.

Chapter 3

Wavelet Transform RTP Core Library

3.1 Library Overview

The RTP Core specification provides a mechanism for describing designs as a conglomerate of functional cores and interconnects using the JBits API. The DWT system makes advantageous use of the RTP core specification in that it is hierarchical composition of smaller child cores that each performs a unique function (Figure 3.1). Each core used in the design offers full parameterization capabilities for the designer.

The DWT system was implemented using a bottom-up approach in which the smaller RTP cores were created first before moving to higher abstraction cores. The creation of these smaller cores led to the formation of a DWT RTP core library. The library included cores written specifically for this research and already existing JBits cores. The following sections provide a detailed description of the lower abstraction cores developed for this research.

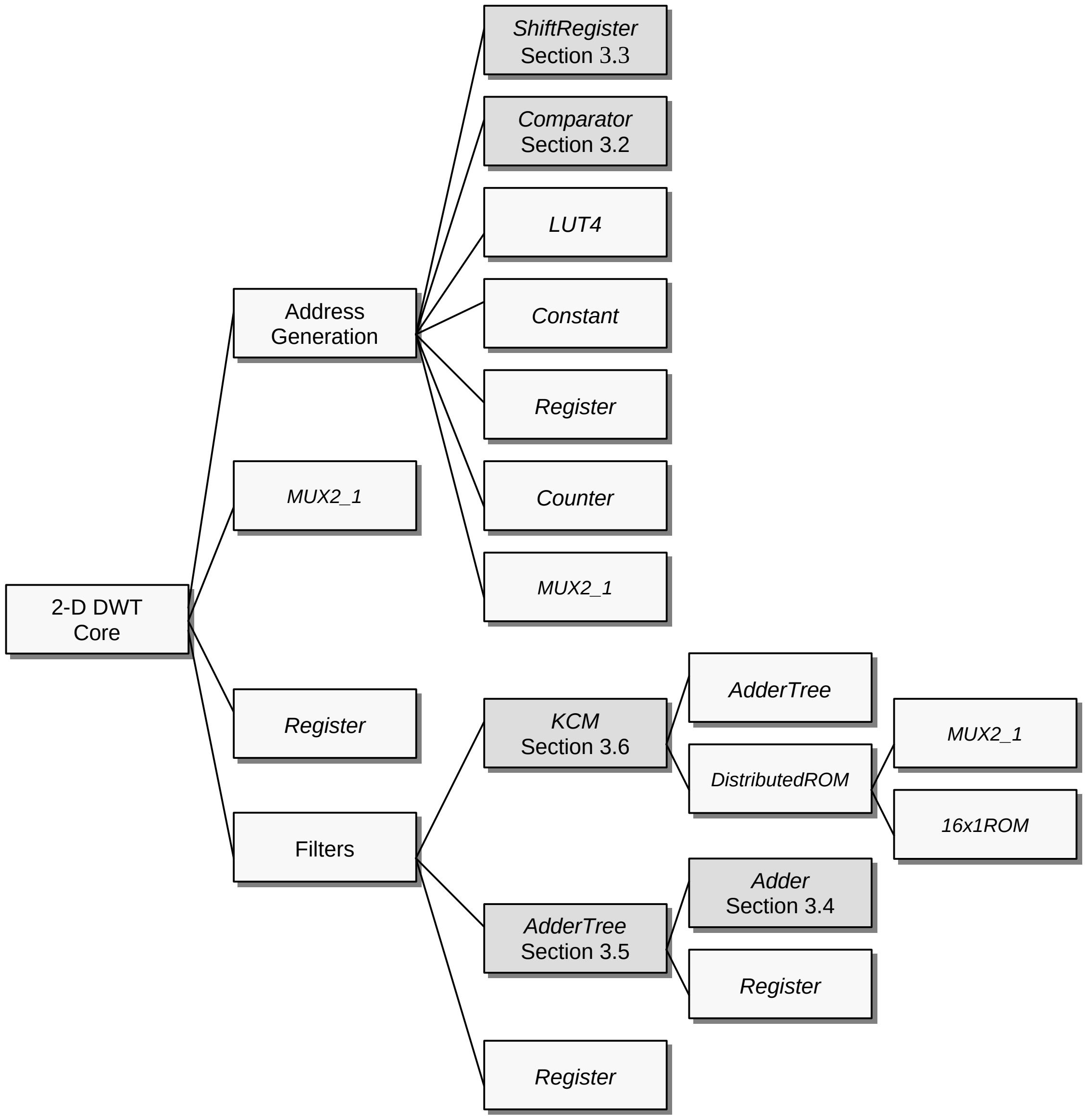


Figure 3.1: Hierarchical core decomposition of the overall DWT system. Blocks containing section numbers point to the particular section that describes the core implementation.

3.2 Comparator

The *Comparator* core determines the equivalency of two input signals. The core is parameterized by a core name, two input signals, and an output net. The output net, *dOut*, is asserted when the data on input bus *AIn* is equal to the data on input bus *BIn*. Signals *AIn* and *BIn* must have the same width. Two signals with a maximum width of two are compared together using a *ComparatorStage* core. Each comparator stage occupies a single Virtex LUT, and are stacked on top of each other to implement wide comparators. The carry chain is used to propagate an “equivalence” output signal between stages. The core can exhibit either synchronous or asynchronous behavior, depending on whether a clock signal is passed to the constructor. The core features CLB height granularity, and SLICE width granularity.

3.3 Shift-Register

The *ShiftRegister* core implements a shift register function in hardware. The core constructor is parameterized by an instance name, an input net, *dIn*, an output bus, *dOut*, and an enable signal, *CE*. The input net, *dIn*, drives the data bit that is shifted into the register on each shift operation. The output signal, *dOut*, is driven by the register outputs. The shift register size is determined by the width of the *dOut* signal. An assertion of the *CE* signal enables the shift operation. This enable signal is asynchronous. The shifting direction is specified as a parameter in the *implement()* method. The initial value of the shift register is also defined in the *implement()* method. An assertion of the global set/reset line (GSR) causes the initial values to be loaded into the registers. Individual bits of the shift register are implemented using *ShiftRegisterStage* child cores. A stage occupies a single LUT. Each consecutive stage is placed vertically on top of the previous stage. Assuming the *dOut* signal is *N* bits wide, the *ShiftRegister* is *N*/2 CLBs high, and one slice wide.

3.4 Variable-Width Adder

The *Adder* core adds data on the two input buses and drives the data out bus with the result. The core provides unique parameterization capabilities. The widths of the two inputs buses do not have to match, allowing for variable input width of both inputs. Also, the designer can choose synchronous or asynchronous operation by passing a clock net to the constructor. The constructor also provides the option to use a carry in, if a carry in net is passed as a parameter. The *Adder* core features slice width granularity, and CLB height granularity.

3.5 Adder Tree

The *AdderTree* core allows any number of tree inputs to be summed together. The core is comprised of variable width *Adder* cores that sum two inputs together, as well as *Register* cores that provide required delay in the case of unbalanced trees. An unbalanced tree results in a case where the number of tree inputs is not an even power of two. A stipulation of the current core implementation is that although the inputs can be of any width, all tree inputs widths must be equal. The output bus width of the core can be of two possible widths, depending on designer configuration. The width is either the same width as the tree inputs, or is the width of the tree input plus $\lceil \log_2 n \rceil$ in order to preserve adder carries, where *n* is the number of tree inputs.

The number of adders required for a tree with *n* inputs is simply *n-1*. The *AdderTree* core exemplifies the advantages of run-time reconfiguration, since an efficient tree is derived for any given number of inputs during core instantiation. Each addition stage is pipelined, thereby allowing for increased clock frequencies at the expense of increased output latency. The latency of the core from input to output is given by $\lceil \log_2 n \rceil$. Each *Adder* core is placed horizontally adjacent to the previous adder. If necessary, delay between addition stages is implemented using registers. Figure 3.2 illustrates the above concepts by providing an unbalanced, ten-input adder tree. As shown in the figure, the tree has a latency of four clock cycles. Two registers are required between the adder labeled eight and the adder labeled nine.

Routing between adders follows the hierarchical tree structure shown in Figure 3.2. Using this approach to placement, routing distance between adder levels is optimized. The optimization results from the fact that two adders providing input for the next stage adder are spaced an equidistant apart, except in the case where the adder core is located in an unbalanced stage. Figure 3.3 shows the horizontal placement scheme used for the adder tree.

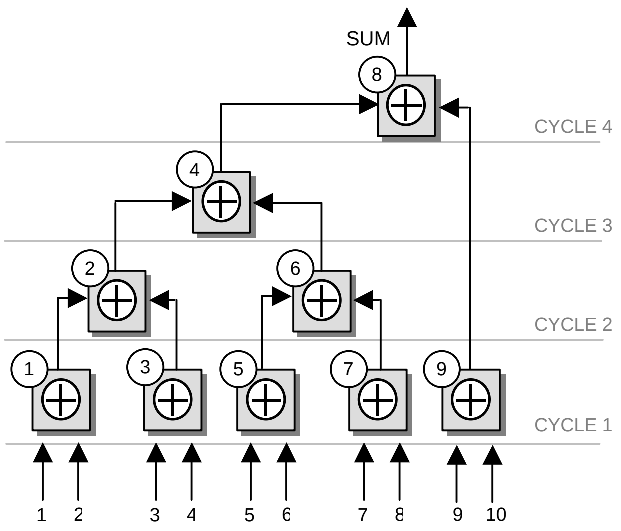


Figure 3.2: *AdderTree* core composition where the circled numbers designate the particular adder index and the horizontal lines partition the addition operations into different clock cycles.

Deriving the necessary interconnections between adders presented several challenges. First, the adder tree was designed so that adder carrys are preserved between addition stages. Secondly, the tree interconnect structure must be known before instantiating the *Adder* cores, since the *Adder* core constructor accepted both input and output buses. Lastly, some tree configurations required registers to be interspersed with adders in order to balance latency between addition stages.

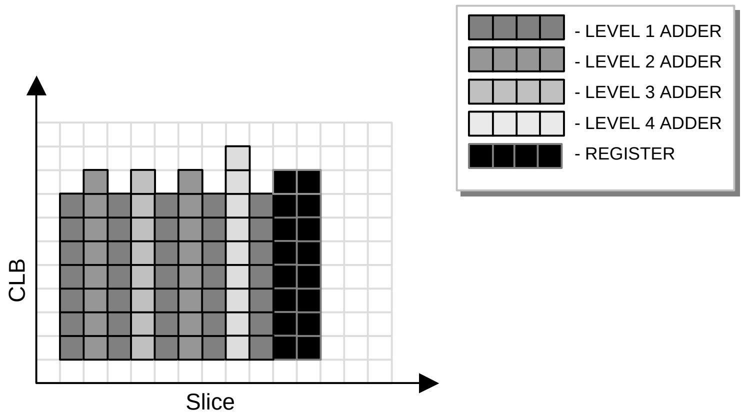


Figure 3.3: Ten-input *AdderTree* core floor plan

Two integer arrays are used to store indices of the *Adder* core outputs that provided the two inputs into that particular adder. As an example, in Figure 3.2 for the adder labeled two, the first array identifies the first input as coming from adder labeled one's output, while the second array identifies the second input as coming from adder three's output. Such values are stored for every adder, and define the adder interconnect structure. A function was written to calculate the values for both arrays, as well as the number of registers required for a particular adder instance. The function uses recursive partitioning of the tree in order to determine adder interconnects, based on an adder range defined by *high* and *low* parameter values. Partitions are created using the highest and lowest powers of two for a particular range of adders. Figure 3.4 provides a Java code excerpt from the *AdderTree* core illustrating this process.

Method *deriveAdderTree()* accepts two adder tree indices, *low* and *high*, that define the current partition. Input *A* of the parent adder is stored in *Aindex[parentAdder – 1]*, while input *B* is stored in *BIndex[parentAdder –1]*. After computing the adder input indices, left and right hand partitions are defined using a recursive call to *deriveAdderTree()*. The function continues until the range defined by the difference of *high* and *low* is less than three, meaning a leaf node has been reached.

```
/* calculate the “parent” adder index */
int log = (int) Math.ceil(Math.log((double)range)/Math.log(2.0));
int parentAdder = low + (int) Math.pow(2.0,log - 1);

/* calculate the right sided adder input index */
range = high - parentAdder;
log = (int) Math.ceil(Math.log((double)range)/Math.log(2.0));
int RHSIndex = ((int) Math.pow(2.0, log - 1)) + parentAdder;

/* calculate the left sided adder input index */
range = parentAdder - low;
log = (int) Math.ceil(Math.log((double)range)/Math.log(2.0));
int LHSIndex = ((int) Math.pow(2.0, log - 1)) + low;

/* partition left hand side of parent adder recursively */
AIndex[parentAdder - 1] = LHSIndex - 1;
deriveAdderTree(low, parentAdder);

/* partition right hand side of parent adder recursively */
if (RHSIndex != parentAdder)
{
   BIndex[parentAdder - 1] = RHSIndex - 1;
   deriveAdderTree(parentAdder, high);
}
else /* required if there is an odd number of tree inputs */
{
   BIndex[parentAdder - 1] = treeInPort.length - 1;
}
```

Figure 3.4: *AdderTree* input index computations and partitioning process

3.6 Constant Coefficient Multiplier

Multipliers are frequently used in digital signal processing systems. In the discrete wavelet transform filter bank, multipliers are required for the filter component implementation. Because filter coefficients remain constant during the entire duration of a transform, constant coefficient multipliers (CCMs) were considered for the design. Dinechin and Lefevre justify the use of constant-coefficient multipliers if the lifetime of the constant is substantially larger than the reconfiguration overhead time [27]. It is desirable to have a fast CCM that not only makes efficient use of FPGA resources, but one that lends itself to a straightforward implementation in FPGA hardware. The Ken Chapman Multiplier (KCM) fulfills the above criteria [11].

The KCM design provides an implementation of a constant coefficient multiplier that is ideally suited for RTR and the Virtex family architecture. A KCM requires fewer resources and results in less latency than variable coefficient multipliers. The design stores precomputed partial product values in Virtex LUTs. The partial products are computed by multiplying all values within a LUT address range by the desired constant value. The width of the multiplier operand input is partitioned into fixed, even width sections. The values corresponding to a particular section of the operand provide an index into a series of LUTs, which in turn produces the partial product. The partial products of all sections then are summed to produce the product output.

The JBits *KCM* core implementation is comprised of a series of *DistributedROM* cores with outputs summed together by an *AdderTree* core (Figure 3.5). Several aspects of the JBits *KCM* core are reconfigurable. In addition to being parameterized by the coefficient, the bit precision in which that coefficient is stored is also configurable. This allows the fixed-point precision of coefficients to vary depending on the designer's needs. The bit-width of the operand input bus is parameterizable as well. The input width, however, is required to be an even multiple of four.

DistributedROM cores of size $16 \times N$ are used for the KCM implementation, where N is equal to the coefficient resolution plus the input bus width. During reconfiguration, each of the sixteen ROM locations is loaded with the value of coefficient k multiplied by the ROM location index. The product is left shifted by a multiple of four, depending on input lines addressing the ROM.

The floor plan of an example 12×8 JBits KCM core is shown in Figure 3.6. A 12×8 *KCM* requires three 16×20 ROMs. A 16×20 ROM uses 20 LUT4 primitives stacked together in a vertical column. A single 16×20 ROM, therefore, has single slice width, and a height of ten CLBs. The three ROM cores are placed horizontally adjacent to one another, requiring three Virtex slices. The three four-bit bus partitions of the twelve-bit operand address their corresponding ROM and produce three partial products. In this instance, the *AdderTree* core is comprised of two *Adder* cores and a *Register*. In total, the 12×8 *KCM* is six slices wide and eleven CLBs high.

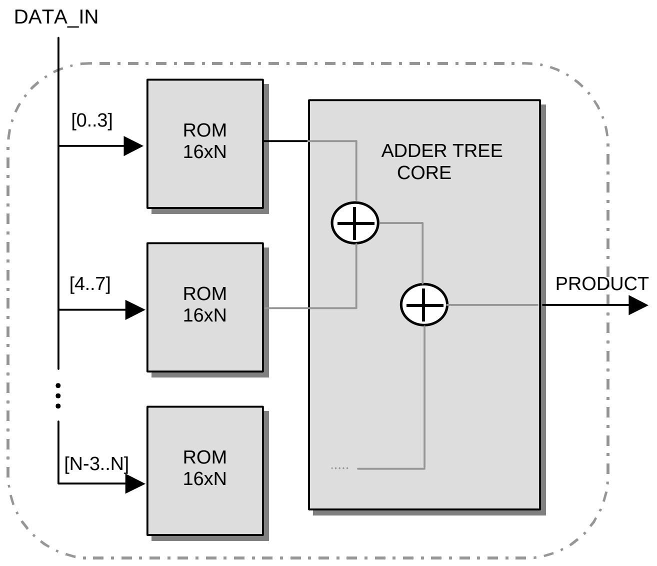


Figure 3.5: An *Nx4* input *KCM* core design

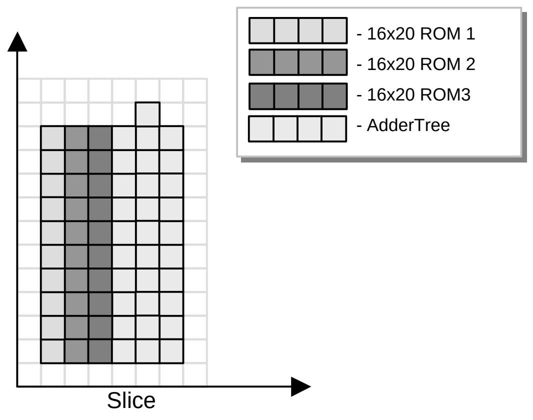


Figure 3.6: 12×8 *KCM* core floor plan

3.7 Summary

The design of a large system is more easily approached if it is broken down into smaller subtasks. With this idea in mind, the DWT system design was decomposed into smaller functional cores using the RTP core specification. The DWT core library provides the components necessary for constructing the larger DWT system. This chapter explored the implementation of the smaller cores designed for this research. The next chapter discusses how these cores are used together to create the filter bank, the address generator logic, and the overall two-dimensional discrete wavelet transform system.

Chapter 4

Discrete Wavelet Transform Core Design

This chapter describes the implementation of a two-dimensional discrete wavelet transform RTP core using the RTP core library. A design overview is presented to familiarize the reader with the major components, interconnects, and functionality of the system. The FIR filter core design and address generation logic used in the DWT system are also discussed.

4.1 Design Overview

The *DWT2D* core performs the two-dimensional discrete wavelet transform on a specified image. The core is parameterized with a core name, coefficient precision, and two *SRAMProperties* classes that define the external signal connections to memory. The *SRAMProperties* class is elaborated on in Chapter 6. The image height and width, along with the high-pass and low-pass synthesis filter coefficients are passed as parameters to the *implement()* method of the core.

Figure 4.1 shows that the *DWT2D* core is comprised of two filters, delay registers, multiplexers for switching between filter and address generator outputs, and two memory address generators that interact with the memory banks and control data flow direction. Two *FIRFilter* cores provide the focus of the core. Depending on filter length, delay registers are added at run-time to balance latency between filter outputs for a given input.

The two filters are loaded at run-time with the high-pass and low-pass coefficients that defined the particular wavelet being used for the transform.

It is desirable to have the *DWT2D* core function as an autonomous design entity within an FPGA. For this to be a realizable goal, the core implementation requires internal control logic and address generation circuitry. The wavelet transform core control logic is responsible for performing data multiplexing. The address generator supplies the necessary memory address input and output buses to both memory banks. It also generates the signals necessary for controlling the read/write operations of the external memory.

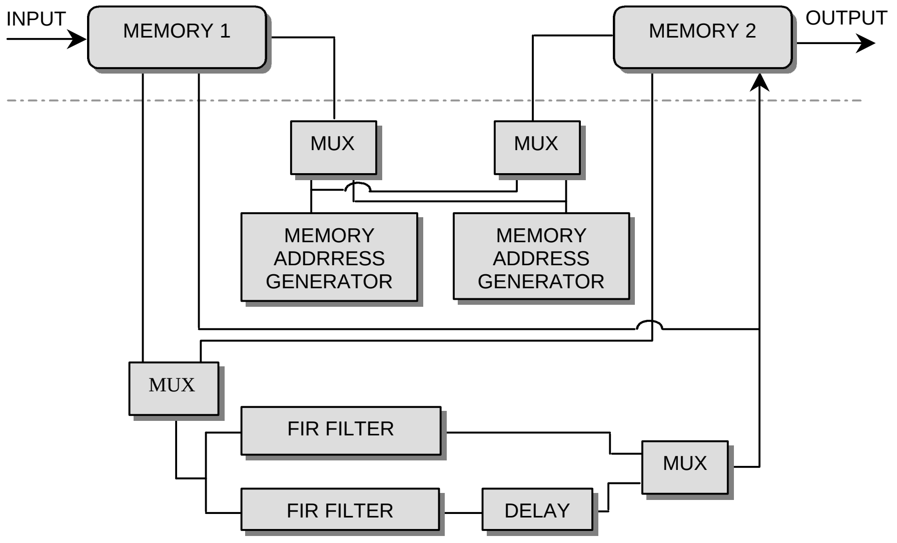


Figure 4.1: Block diagram of *DWT2D* core

4.2 Sequential FIR-Filter

Two sequential *FIRFilter* cores act as the computation engine of the wavelet transform core. The filter structure defines a sum-of-products hierarchy consisting of registers, constant coefficient multipliers, and adders. Each core makes advantageous use of the low-level bitstream access afforded by JBits in order to provide the capabilities needed for efficient placement during reconfiguration of the filter.

The output of a linear filter can be described by the convolution equation:

$$y(n) = \sum_{i=0}^{n} x(i)h(n-i) \quad [4] \qquad \text{(EQ. 4-1)}$$

The direct form implementation of the convolution equation is shown in Figure 4.2. The *FIRFilter* core is based on the direct form structure [4]. The delays are implemented using a chain of register cores. The coefficient multipliers are implemented with *KCM* cores. The multiplier outputs are summed together using the *AdderTree* core.

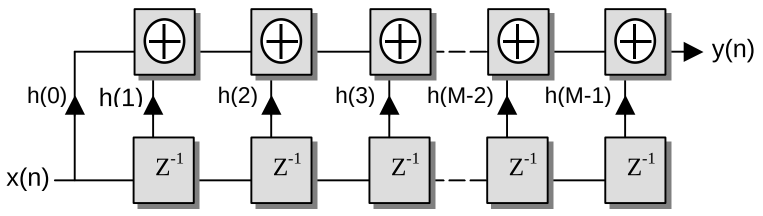


Figure 4.2: FIR filter direct-form structure

The *FIRFilter* core is created in a three-step process. Filter input and output buses are passed to the filter constructor during instantiation. The widths of these buses define data path bit widths and filter resolution. The filter core must then be anchored to physical coordinates on the FPGA. This is accomplished by modifying the filter core's *offset* field, in which CLB row and column coordinates are defined. After placement, an array of *doubles* representing filter coefficients is passed to the core's implement method, in which the parent *FIRFilter* core, along with all child cores, are instantiated, and the bitstream is configured accordingly.

Assuming a given filter is of order *n*, then *n* registers and multipliers are required, as well as *n-1* adders, to compute the sum of products. The latency of the filter of *l* taps, and input bit width *n*, is computed using the following equation:

$$latency = 1 + \left\lceil \log_2 \frac{n}{4} \right\rceil + \lceil \log_2 l \rceil \qquad \text{(EQ. 4-2)}$$

A constant multiplier is instantiated for each coefficient in the double array and loaded with coefficient value. An adder tree is responsible for summing the multiplier outputs. The resulting filter structure is shown in Figure 4.3. The higher abstraction RTP cores required for filter design include *KCM*, *Register*, and *AdderTree* cores.

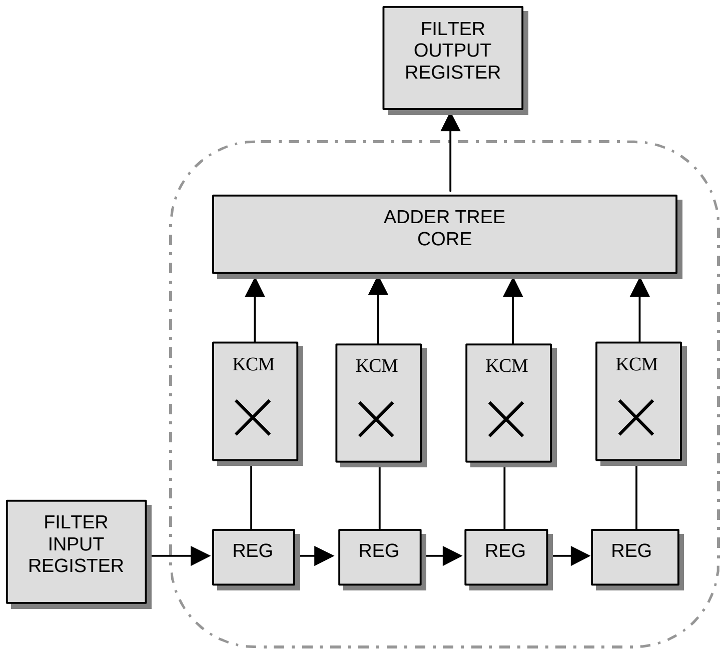


Figure 4.3: RTP core composition of a four-tap FIR filter

4.3 Address Generation

The *DWT2D* core interfaces to two SRAMs. During operation, one SRAM provides the necessary data input to both filters, while the other memory stores the transitory filter coefficient data. As each level of wavelet transformation is completed, the roles of both memory banks are swapped. The input and output address generators for the *DWT2D* core are implemented as separate RTP cores.

Two address generator cores provide the necessary addresses for both SRAM memories. The address outputs were interfaced directly to the physical SRAM address lines. One address generator core computes the input addresses, while the other core computes the output addresses. Both cores supply the necessary memory address input and output values to SRAMs. The implementation of the input and output address generator cores is discussed in Sections 4.3.1 and 4.3.2, respectively.

4.3.1 Input Address Generation

The input address generator core supplies the memory address values for the SRAM responsible for providing data input into the transform core. The implementation of this core is straightforward, since no row or column extension scheme is used for the transform. Because no extension is used, the border of resulting transformed array contains invalid transform coefficients. The resulting transformed image, therefore, requires cropping before applying the inverse transform. The amount of cropping required is dependent on the filter length.

The core *implement()* method is parameterized by an image height/width dimension value. Two restrictions are placed on images used in the transform process.

- The height of the image was equal to the width of the image.
- The height/width parameter was an even power of two.

Figure 4.4 provides a graphical depiction of the row and column address scanning process used for three levels of decomposition. The two-dimensional transform requires that image rows are processed first, followed by a scan of the image columns. The position of the row and column values are reversed in the output address register, thereby automatically transposing the matrix of coefficients as they are written to memory. Using this methodology, the need for additional logic to scan the columns is eliminated, since a second row scan of the transposed matrix accomplishes the same function. All image rows are scanned in the first level of transform. After each level of transform, the available image width and height address ranges are divided by two. This division is needed so that the following transform level operation processes only the detail coefficient suband located in the upper left quadrant of the transformed coefficient matrix. The input address generator repeates the row scan twice using the same image dimension value for each level of transform. A second transposition during the second row scan restores the position of the transformed coefficients to their proper order.

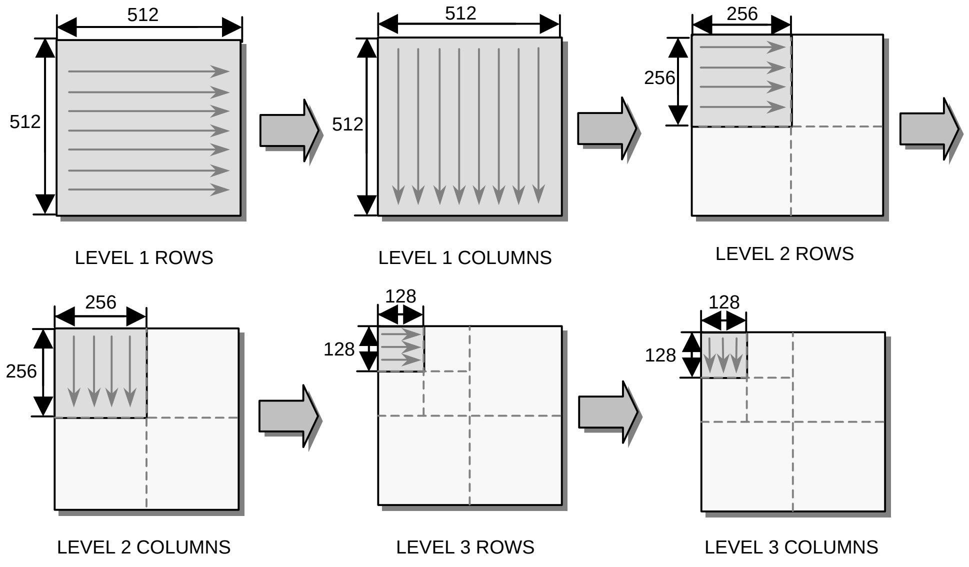


Figure 4.4: Input address generator scanning process

Two *Counter* cores provide the foundation for the row and column output address value computation. Assuming *n* represents the image width and height, the width of both counters is computed by taking the $log_2\ n$. The input address generator core constructor accepts an address bus on which the input address values were driven to the SRAM. Where the SRAM memory is larger than the image, the width of the memory address bus is larger than the combined widths of the row and column counters. In this case, a *Constant* core is used to drive the remaining bits of the address bus with zero values. Figure 4.5 illustrates the bit positioning of the constant, row and column output values within the input address value. The image width and height in this example is 256 pixels. As a result, eight bits are required to store the row and column address values. The remaining four bits are driven low by a constant core.

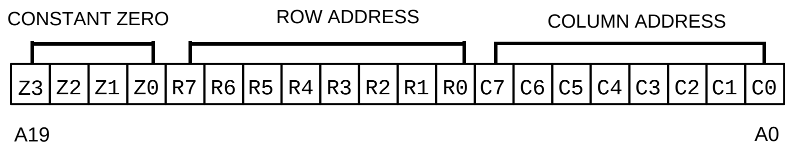


Figure 4.5: Input address composition using an address bus width of 20

After each level of transform, the working image height and width are halved, thereby decreasing the output memory space by a factor of four. Implementing this in hardware requires manipulation of the column counter output bits. Figure 4.6 shows the bit-wise generation of the row address output values. A *MUX2_1* core selects between counter output bit *n* and a constant zero value. On reset, the shift register with the same width as the column counter is initialized to all ones. During transitions between levels, the shift register is enabled and shifted, with a zero loaded into the MSB register value. When a zero is driven on the MUX select line, the MUX outputs a zero, instead of the corresponding counter bit. This scheme effectively divides the range of row output addresses by a factor of two.

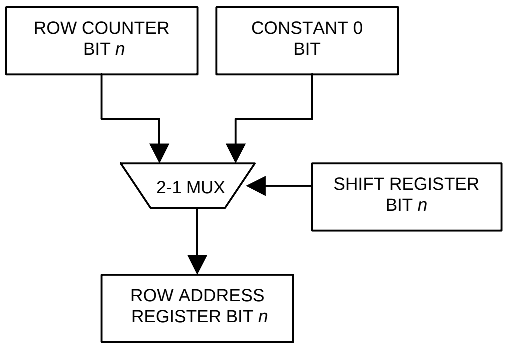


Figure 4.6: Generation of row output address bit *n*

Generation of the column input address uses the same multiplexing scheme as the row address generation to divide the address range by two between levels. It should be noted that both the row and column address logic require their own shift register cores.

Additional logic is used in conjunction with the column address generation to generate the row counter enable signal. The combined logic for the column address and row counter enable signal is depicted in Figure 4.7. The row counter enable signal requires asseration whenever the maximum column count is reached. The maximum count value is already available in the shift register output. A comparator is used to determine if the multiplexed counter output is equivalent to the shift register output. The comparator output, along with the comparator outputs from all other counter bits are AND'd together to produce the row counter enable signal.

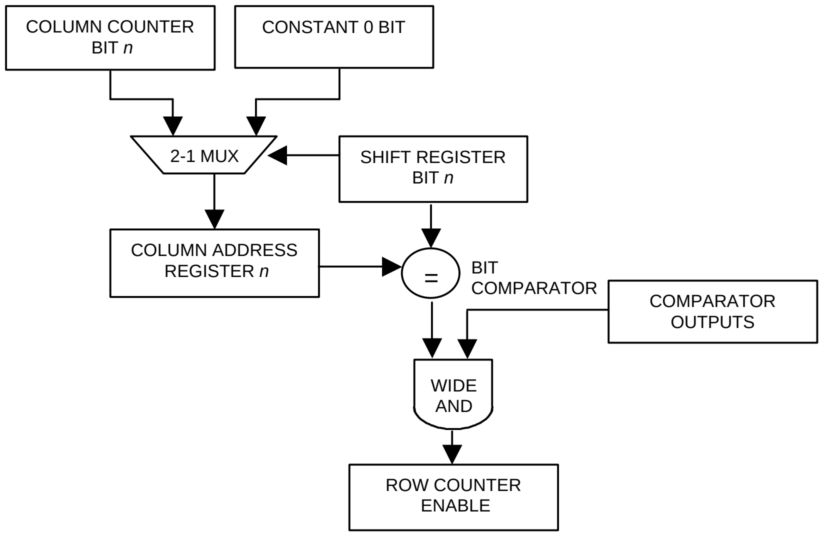


Figure 4.7: Generation of column output address bit *n*

4.3.2 Output Address Generation

The output address generator core supplies the memory address values for the *DWT2D* output data being written to SRAM. Generating the output addresses is less straightforward than the input addresses generation. There are two significant differences. First, the latency of the SRAM and filter output requires appropriate delay of the output address values. Secondly, output data has to be written to its appropriate subband partition in memory.   To address these issues, generation of the output memory address requires variations on the logic used in the input address generation.

Several sources, including the filter cores and SRAM, contribute latency delay to valid output availability. As a result, additional logic is required to delay the output address accordingly (Figure 4.8). A *Counter* core counts latency cycles after a reset assertion. A separate constant value stores the appropriate latency value for the current *DWT2D* core configuration. The counter and constant outputs are compared together to generate the CE signal for the output address counter core. An inverted comparator output signal enables the latency counter. In this manner, the counter is disabled after reaching the maximum latency count.

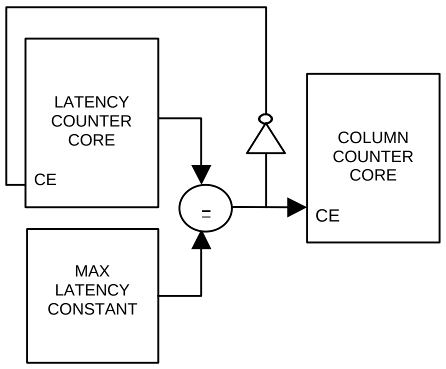


Figure 4.8: Output address latency delay logic

The positioning of the row and column address values is reversed in the output address value written to SRAM (Figure 4.9). Doing so generates a transposed coefficient matrix in the output memory. This allows the input address generator to perform a row and column transforms using two sequential row scans.

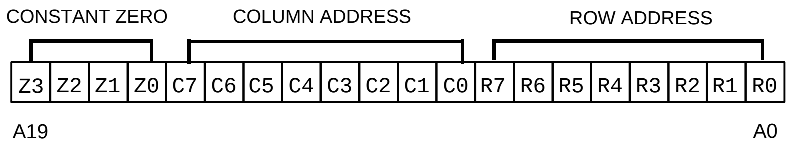


Figure 4.9: Input address composition using an address bus width of 20

The process of generating output row and column address values is similar to the technique used to generate the input row and column address values. As with the input address core, two *Counter* cores provide the logic foundation for the row and column address values.

Unlike the input address generator, however, the output address memory space for a particular level is divided into two column partitions, one for the low-pass filter coefficients, the other for coefficients generated by the high-pass filter. A *MUX2_1* core toggles between filter outputs on every clock cycle. An addressing scheme is therefore required to generate output addresses that alternate between these columns on successive clock cycles. An example column output address sequence for a 512x512 image using three levels of transform is provided in Table 1.

Level	Column Width	Column Output Address Sequence
1	512	0, 256, 1, 257, 2, 258, … 255, 511
2	256	0, 128, 1, 129, 2, 130, … 127, 255
3	128	0, 64, 1, 65, 2, 66, … 63, 127

Table 1: Example column output address sequence for three transform levels

The column switching process is realized in hardware by asserting the most significant bit of the column address on alternate clock cycles. This is accomplished by rerouting bit zero of the column counter to the MSB of the column address. The MSB position of the column address is dependent on the current level of transform. Figure 4.10 shows the generation of a column output address bit *n*.

The complication in logic results from the fact that the most significant bit position changes as the transform level changes. An additional level of *MUX2_1* cores provides a solution. Column counter bit *n+1* and bit zero provide the mux inputs. Because the counter bit zero is rerouted to the MSB for the current output address range, the position of the other counter bits are shifted down, such that bit one becomes bit zero. This bit position shift is implemented through mux routing.

A second shift register is needed to control this additional mux level. The shift register values are loaded with a zero in the most significant register, with ones in all other registers. The shift register is enabled during transitions between levels, with a zero being shifted into the most significant register. The shift register output controls the select lines of the *MUX2_1* cores. In this manner, column counter bit zero is always routed to the most significant column output address bit position.

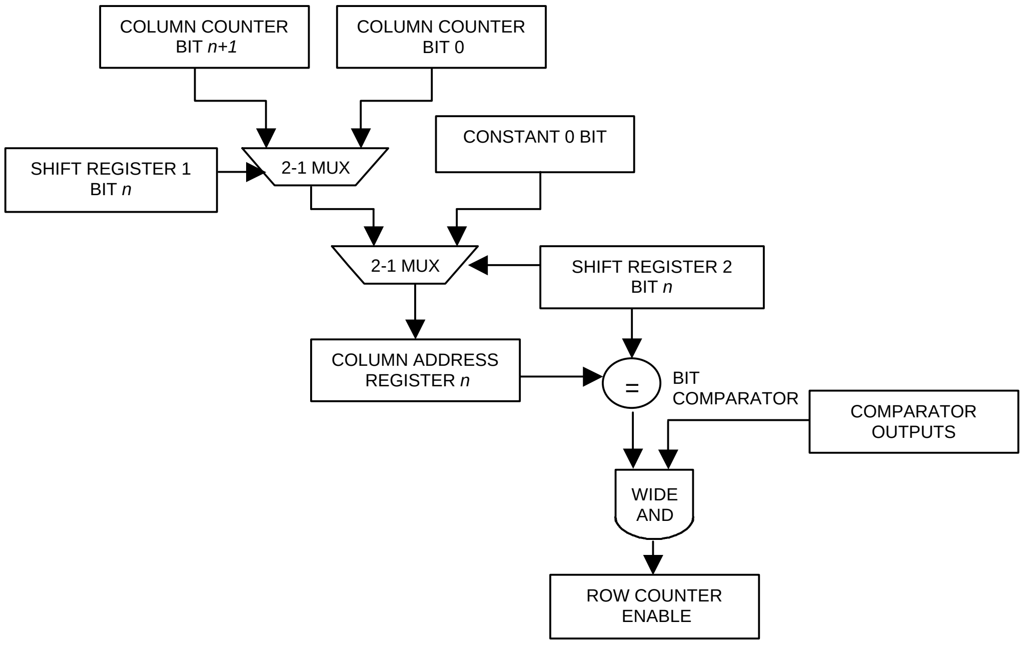


Figure 4.10: Generation of column input address bit *n*

The output address row values are computed similarly to the input address values. The row counter is enabled by the row counter enable signal generated by the column counter logic. The counter is incremented after the maximum column count is reached.

The shift registers for the input and output address generator cores are enabled using the logic scheme shown in Figure 4.11. *Comparator* cores determine when the row and column address values are equivalent to the values stored in the shift registers. The AND of these signals generate the count enable signal of the transform level counter. Bit zero of the transform level counter is then AND'd with the counter enable signal to produce the shift register enable signal. This guarantees that the enable signal is asserted after two sequential row scans. The remaining level counter bits, bit one and up, indicated the current level of transform. The same logic is used for both the input and output address generator cores.

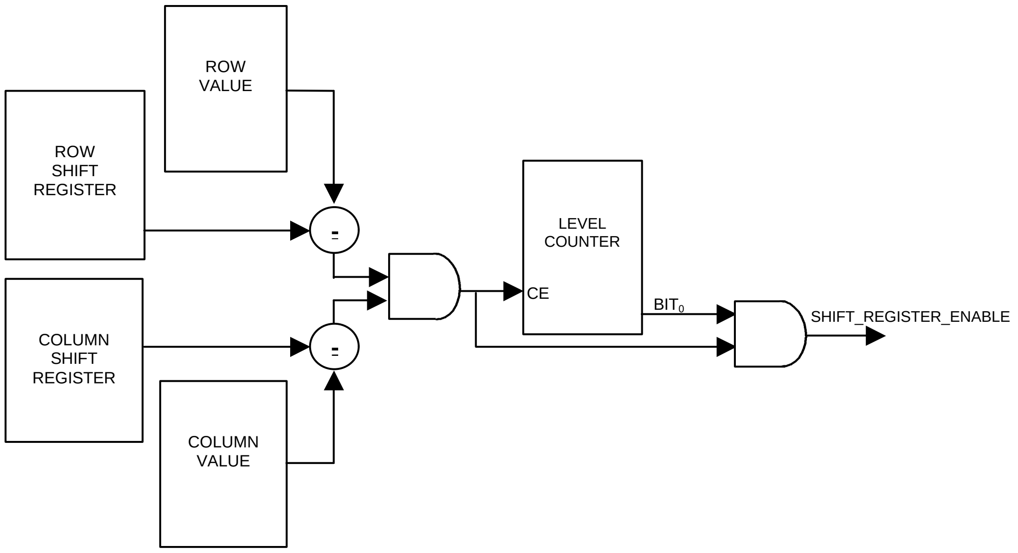


Figure 4.11: Shift register enable signal generation logic

4.4 Summary

The *DWT2D* core is defined by a hierarchical composition of RTP cores. In this chapter, the implementation of the *DWT2D* core is described, along with a discussion of the *FIRFilter* core and address generation logic. The *DWT2D* core is comprised of two FIR filters, input and output address generators, and control logic for multiplexing filter data. The *FIRFilter* RTP core is a JBits implementation of the direct-form FIR filter structure. The logic for both the input and output address generators is encapsulated within two separate RTP cores.

Chapter 5

Simulation and Debugging

This chapter explains the simulation and debugging processes used throughout the wavelet transform system design. The chapter begins with a discussion of how Java was used to simulate the operation of the discrete wavelet transform using classes that represented hardware counterparts. Having this tool aided the debugging processes by providing a means in which test data could be extracted and viewed at any step of the simulation. The chapter also introduces the concept of bitstream level debugging, and discusses how the VirtexDS was used in conjunction with BoardScope to verify correct circuit operation before the bitstream was downloaded to physical hardware.

5.1 Software Simulation

Before beginning the hardware design process, a Java-based software simulator, *DWTSimulator*, was written to model the behavior of the wavelet transform in both the forward and inverse directions. Methods *DWT* and *IDWT* performed the forward and inverse transforms for a selected image, respectively. Classes were written to emulate the functionality of the individual hardware components comprising the *DWT2D* core. The transform simulator was then constructed using these hardware model classes. By modeling the hardware behavior with Java classes, the theoretical transform output could be analyzed for correctness before implementing the system in hardware. Another advantage of using hardware model classes was that the wavelet transform simulation class provided a template for the hardware system design, in which the model classes could be removed and replaced by their respective RTP Cores.

A *FIRFilter* class was written to simulate the hardware filter implementation. The filter was parameterized by a set a coefficients that defined filter length. Method *clock()* accepted a data value, and performed the filter operation for the registered values. Software truncation and masking techniques were applied to the filter input and output data in order to achieve the same resolution as the hardware filter. The *FIRFilter* class performed the signal processing for the simulator. As with the simulator, the *FIRFilter* class acted as a model for the JBits RTP filter core implementation.

Development of both forward and inverse transforms allowed images to be transformed and then reconstructed. The Microsoft bitmap format was used for all images read and written by the simulator. A separate class, *Bitmap*, provided functions for storing and retrieving image coefficient arrays from bitmap files. The *loadBitmap()* method read a bitmap file and returned a two-dimensional integer array of pixel values. Conversely, method *saveBitmap()* accepted a two-dimensional integer array and converted the array into bitmap format. This conversion required scaling the integer array values to fit in the 0 to 255 range.

5.2 Debugging at the Bitstream Level

The design cycle of each RTP core required a debug and validation design phase. The BoardScope tool provided the environment of choice for bitstream level debugging. Using the VirtexDS class in conjunction with BoardScope allowed cores to be thoroughly debugged in software before testing them out on physical hardware. By running the core output bitstream under BoardScope, registered signal values could be monitored on a per clock cycle basis, either through the graphical flip-flop state indicators, or the BoardScope waveform viewer. Although minor discrepancies existed between simulator and physical hardware behavior, correct core operation in the simulator often translated to correct operation in hardware. This section discusses the bitstream level debugging process as it related to the Wavelet Transform core.

Individual signal behavior is determined by monitoring the flip-flop state in which that signal traverses through. As a result, knowledge of core placements and the locations of signals relative to those placements must be derived. This information includes the CLB row and column positions, the slice, and the flip-flop within the slice. With traditional simulators, this information is irrelevant since place and route algorithms have yet to be applied to the design. Another difference is that JBits signal visibility is limited to the logic states of the CLB and IOB flip-flops due to the read back information obtainable from the FPGA. Because the device simulator models the system level hardware, it can only generate synchronous flip-flop state output as well. As a result, asynchronous signal activity cannot be viewed, as is usually possible with traditional simulators. The majority of the cores used in the DWT implementation featured registered outputs however, making this simulation environment acceptable.

5.3 Testbench I/O Generation Techniques

In FPGA design, test benches are often used to provide an automated testing environment wrapper around the design. Test bench design is nonstandard when compared to traditional simulation environments due to the bitstream level output generated by JBits cores, however. The bitstream level output distinguishes JBits test benches from traditional test benches in several regards.

A significant distinction between JBits and standard HDL test benches is the methodology in which the test vector core stimuli are generated. Several techniques were used to provide test input vector stimuli into the RTP cores used in the DWT implementation. These techniques included placing hard-coded vectors in the circuit with the *TestInputVector* RTP core, using the *SimulatorClient* class to provide stimulus through the VirtexDS [43], and using BRAMs to store vectors. Each technique was utilized for the DWT core library debugging process, and warrants further discussion.

The first method involved using a *TestInputVector* RTP core. The core is parameterized with depth, clock signal, and data bus parameters. The depth parameter defines how many vectors should be written during simulation. The width of the input vectors is defined by the width of the output data bus. The file containing input vectors values is specified as an *implement()* method parameter. These vectors are then coded into LUTs operating in SRL16 mode.

Generating vectors with an RTP core is unique in the sense that the vectors themselves are coded into the bitstream. This methodology has both advantages and disadvantages. A major disadvantage results from the resource consumption requirements of the core. The depth of the core is limited by the space remaining on the device after core implementation. For signal processing cores, including the *FIRFilter* core, it is difficult to store signals of any significant length. As a result, the *TestInputVector* core was used only for debugging smaller cores, including the adder and comparator cores.

Test vectors are also obtainable through a direct interface to the device BRAMs. This scheme was used to create a streaming data interface for cores. The streaming interface required two BRAMs to provide input and store output data. An RTP *Counter* core generated the memory addresses. While this method proved effective for filter debugging, available memory was too small to store entire images. This approach suffers from the disadvantage that the test vectors must be stored in the bitstream, as with the *TestInputVector* core.

The most flexible solution involves using the *SimulatorClient* class to provide input stimulus. Upon instantiation, the *SimulatorClient* establishes a TCP/IP connection to the VirtexDS already running. The class provides methods for setting and retrieving the logic values of FPGA pins. Using these methods, vectors are injected into the core at register input pins. This methodology is much more desirable than the previous two because it eliminates the reliance on bitstream resources to store vectors. The client became available late in development, so its usage was limited to debugging higher abstraction cores developed in the later stages of design.

Chapter 6

Hardware Interfacing

This chapter describes how the *DWT2D* core is interfaced to the Slaac1V SRAM memories. The motivation for RTR I/O is presented first, followed by the introduction of a set of JBits classes that provide RTR I/O interfacing support for JBits designs. Only recently has the support to route to non-CLB tile resources, including BRAMs and IOBs, been included in the JBits distribution. These RTR I/O classes extend on the routing support, and provide automated I/O configuration and interfacing support for top-level RTP cores. Following the introduction of the RTR I/O classes, the chapter discusses how these classes were used to interface the *DWT2D* core to the Slaac1V hardware.

6.1 I/O and Run-Time Reconfiguration

As with most image processing applications, the discrete wavelet transform requires access to large quantities of data. A 512×512 grayscale image with eight bits per pixel requires 256 K bytes of memory storage. A three-level DWT, using the 512×512 image has a data throughput requirement of 672K bytes. The large data requirement forces the *DWT2D* core to use an alternate approach to the distributed RAM CLB configuration approach. Although the Block RAM (BRAM) provides a possible on-chip memory storage solution, the size of the memory is insufficient for large images. The XCV3200E is currently the largest Virtex extended memory device and offers 851,968 BRAM bits [12].

This is still inadequate storage for a 512×512 grayscale image. It is evident that an off-chip memory solution is needed to provide data to the core.

In the past, JBits designs requiring external I/O have utilized bitstreams containing I/O frameworks generated from other HDLs and synthesized using the standard tool flow. Dynamic circuit design, therefore, has been restricted to internal circuit logic, leaving I/O data paths static. There is significant motivation for providing I/O reconfiguration capabilities with JBits, however. The most apparent advantage is the elimination of the reliance on other tools. Supporting I/O reconfiguration allows complete design realization with JBits. Designs offering variable degrees of output accuracy [13] could make advantageous use of variable width I/O data paths. These designs could allot additional I/O resources or free IOBs for use by other cores during reconfiguration. Providing automated I/O interfacing capabilities eases the process of porting JBits system level designs between Virtex devices.

The *DWT2D* core is designed with the inherent memory addressing and control logic needed for it to function as an autonomous entity without the assistance of any additional FPGA circuitry derived from other HDLs. As a result, the top-level *DWT2D* core requires direct connections to SRAM data, address, and control signals. It is desired to have the *DWT2D* core and I/O data paths generated entirely from a null bitstream during instantiation. This goal requires that each port contained in the *DWT2D* core interface be routed directly to a corresponding IOB or IOBs, which in turn provides access to the appropriate SRAM signals.

Although the Slaac1V board is used as the hardware-testing platform, it is undesirable to hard code the core with routing calls specific to a particular fixed IOB configuration. Doing so effectively makes the core device package and hardware platform dependent. Porting to other platforms would require an entirely new set of IOB routing calls. Another disadvantage of hard coding the routing calls is the introduction of human error. With large designs, this process is not only tedious, but error prone as well.

As a reBsult of these disadvantages, an additional layer of abstraction is needed within JBits to automate the interfacing of an RTP core to different FPGA based hardware platforms. Introducing such a layer allows RTP cores using IOBs to become Virtex device independent. Section 6.2 presents a series of JBits classes that generate an external I/O interface wrapper around a JBits RTP core, using the specifications given in a traditional user constraints (UCF) file.

6.2 JBits RTR I/O Interfacing Classes

This section introduces a set of JBits classes that automate the process of interfacing a top level RTP core to the underlying FPGA hardware platform. These classes were developed as a coauthored research effort[1]. Using the RTP core template specification, these classes are encapsulated within a single RTP Core, the *Board* class. A system level diagram of the hardware interfacing JBits components is shown in Figure 6.1. The *Board* class provides a single core RTR solution for I/O mapping and configuration. The functionality of the *Board* class is discussed, along with the supporting I/O classes that are used by *Board* during the interfacing process.

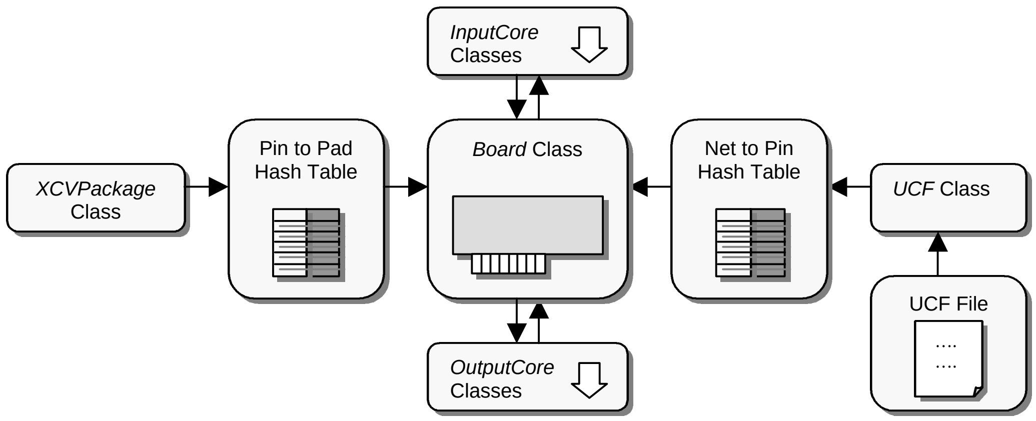


Figure 6.1: System level diagram of JBits RTR I/O interfacing classes

[1] Design of the RTR I/O classes was a collaborative effort with Scott McMillan on the Loki team. Scott was responsible for taking the existing interfacing methodology designed for this research, and encapsulating it within the *Board*, *InputCore*, and *OutputCore* RTP cores. He also extended the classes by adding automated I/O configuration support to the RTP cores and enhancing the UCF parser.

6.2.1 The *Board* Class

The *Board* RTP core class provides a physical FPGA hardware framework for accomplishing run-time mapping and reconfiguration of I/O resources. In essence, the *Board* class is an overall abstraction of the underlying board and device hardware. Fields are provided for defining hardware specific information, such as the board global clock and FPGA device package pin to pad mappings. The mappings of a particular Virtex device package are stored in an *XCVPackage* class. The *XCVPackge* class warrants further explanation and is discussed in detail in Section 6.2.3. By extending the *Board* base class, a designer can assign values to these fields accordingly, thereby creating a user defined board class. As an example, the code for the *Slaac1VBoard* class is provided in Figure 6.2. The global clock line for the Slaac1V board is defined as GCLK 2. Three xcv1000_fg680 devices constitute the *XCVPackage* FPGA device array. These device specifiers correspond to the three processing elements on the Slaac1V board.

```
public class Slaac1VBoard extends Board
{

   public Slaac1VBoard(String name) throws CoreParameterException
   {
      super(name);
      setXCVPackage(xcvPackage);
      setGCLK(GCLK);
   };

   private XCVPackage xcvPackage[] =
   {
      new xcv1000_fg680(), new xcv1000_fg680(), new xcv1000_fg680()
   };

   private static int GCLK = 2;

}; /* end of Slaac1V board class. */
```

Figure 6.2: *Slaac1VBoard.java* code

The *Board* class provides methods for manipulating FPGA IOB resources, however such resource configurations calls refer processing to the resource configuration methods in *InputCore* and *OutputCore* classes. An RTP core abstraction of IOB resources is represented using *InputCore* and *OutputCore* RTP core classes. Both cores are elaborated on in later sections. I/O cores are added to a board using the *addInput()* and *addOutput()*

methods. These methods accept a string name of the I/O core and a *Signal* instance. The board class maintains array lists of all added cores. Methods are also provided for configuring specific IOB resources. The JBits code in Figure 6.3 demonstrates the process of adding and configuring an output core to a board. In this case, a *Bus* signal *XBar* of width 20 is instantiated. Next, a *Slaac1VBoard* instance is created. A new I/O output is then added to the board, using the *slaac1V.addOutput()* method. In order to drive the physical I/O pad, the IOB tristate is inverted using the *slaac1V.setOutputInvertT()* method, since the tristate enable is active low and the unconnected tristate wire are designed to float high in the Virtex architecture.

```
/* create a signal to run to the cross bar pins on the Slaac1V X2 */
Bus XBar = new Bus("XP_XBAR", null, 20);

/* create a new board */
Slaac1VBoard slaac1V = new Slaac1VBoard("SLAAC1V");

/* add a cross bar output core to slaac1V board instance */
int XBarOutput = slaac1V.addOutput(Xbar.getName(), XBar);

/* configure the IOB resources output operation */
slaac1V.setOutputInvertT(XBarOutput, true);

/* implement the slaac1V board */
slaac1V.implement(0, "slaac1V.ucf");
```

Figure 6.3: Adding and configuring an *OutputCore*

After the desired input and outputs have been added to the board and appropriately configured, the *implement()* method must be called. The *implement()* method accepts two parameters, the targeted FPGA device number and the UCF name to use in the net name to pin mapping process. The method performs two tasks. A hash table containing net name to device pin translations is first generated using the *UCF* class. The *implement()* methods are then invoked for every *InputCore* and *OutputCore* that had been added to the board.

6.2.2 The *InputCore* and *OutputCore* Classes

The *InputCore* and *OutputCore* classes provide software core abstractions of the physical FPGA IOB resources. Because the *Board* class manages the functionality of these cores, their operation is hidden from the designer. The core constructors are parameterized by a name and an associated signal instance. Core methods are provided setting particular IOB configuration parameters. These parameters are used by method *implement()* to make the IOB bitstream modifications necessary to realize these configurations. This method accepts an *XCVPackage* instance, as well as the UCF hash table created in the *Board* class. Using these classes, the *implement()* method is able to map the signal instance to a corresponding IOB or group of IOBs. The process of mapping signals to JBits IOB resources is shown below in Figure 6.4. The depicted mapping process deserves further elaboration.

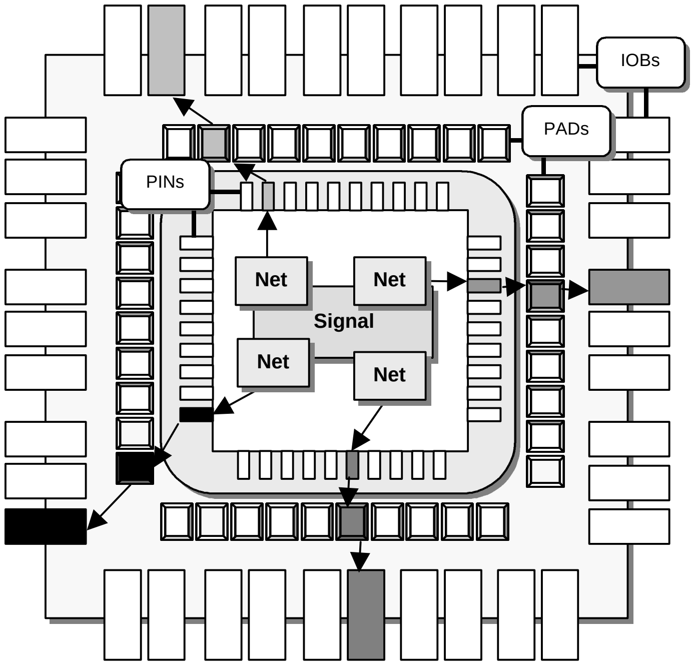


Figure 6.4: Mapping JBits *Signals* to IOB *Pins*

The *InputCore* and *OutputCore* classes have an inherent *Signal* associated with each instance. The *implement()* method considers every net within the signal on an individual basis. To begin, a net is mapped to a physical pin location using the net name to pin entries in the UCF hash table. After determining the pin location, the pin is translated to the connecting I/O pad within the FPGA. This is accomplished using the pin to pad translations contained in the *XCVPackage* hash table. The JBits IOB resource can then be computed based on the pad identifier. Finally, a JBits *Pin* can be attached to the endpoint of the net using the indices that identify the JBits IOB resource.

Once the hash table has been created for the constraints file, operation of the generate method continues by examining the port interface of the RTP core. The port examination process involves determining which net or bus that specific port has connected as an external signal. After extracting the net/bus, the string name assigned to the signal is used as an index into the constraints file hash table. The return value, a device pin identifier, is then used to index the XCVPackage hash table. The XCVPackage hash table returns the pad number corresponding to the device pin. By using the JBits device row and column, the corresponding JBits IOB resource is identified.

6.2.3 *XCVPackage* Class

The *XCVPackage* provides a base class framework for storing device pin to I/O pad mappings specific to a particular Virtex device package. By instancing an XCVPackage class, a hash table member field is defined with the chip pin *String* identifier as the hash key and the corresponding *Integer* pad number as the return value. It is the responsibility of any class extending the *XCVPackage* to define the contents of the table. Sets of such classes that extend *XCVPackage* were created in order to provide pin to pad hash tables for every Virtex package currently available. The class names are representative of the device package names, for ease of use. As an example, the *xcv800_bg560* class represents the Virtex 800 part with a BG560 package. The class also provides methods for retrieving the number of CLB rows and columns for the particular Virtex package.

The *xcvPackage* array field of the *Board* class stores *XCVPackge* classes for every FPGA device on the hardware platform.

6.2.4 *UCF* Class

The *UCF* class provides a user constraints file parser that creates a hash table in which the net string name identifier provides the key, while a corresponding *UCFValues* instance provides the return value. The *UCFValues* class holds configuration parameters, including skew, drive, pull, and the iostandard for a specific IOB. The designer never deals directly with the parser, however, as *Board* method *implement()* is responsible for a calling the parser automatically, after receiving a constraints file from the parameter list.

It was decided to use the standard user constraints files (UCF) file format to map *Signal* names to physical device pins. This format was used for two reasons. First, the format is consistent with the format used by the Xilinx tool chain. Secondly, since this is the preferred format, user constraints files already exist for the majority of hardware platforms. Although the UCF parser can extract IOB behavior commands, such as skew rate and drive strength, only the nets to pin mapping commands are considered in the route determination process. The format of the net to pin mapping is shown below in Figure 6.5. The code features an excerpt from the Slaac1V board's UCF file in which SRAM 1's control signals are mapped to pin locations. Net names are preceded by "net" and the pin location identifiers are preceded by "loc=". In this case, the first line gives a pin location of C30 for the "XP_MEM1_CE_N" net.

```
# Slaac1V memory 1 control signals

net XP_MEM1_CE_N    loc=C30;
net XP_MEM1_LD_N    loc=A31;
net XP_MEM1_WE_N    loc=B30;
```

Figure 6.5: Excerpt from Slaac1V UCF file

6.3 Interfacing to the SLAAC1V-Board

The *DWT2D* core was tested using the Slaac1V board. The bitstream containing the wavelet transform system was loaded and run on the *X2* XCV1000 processing element. The Slaac1V board provided four 256k SRAMs per processing element, providing enough storage for a 512×512 image. The core implementation required that data be read from one memory, while filter output was written to a second memory during a single clock cycle. As a result, the *DWT2D* core required access to two of the processing element's four $32 \times 256\,\mathrm{k}$ SRAMs. The core required direct interfacing to the physical memory data, address, and control signals for both memories. SRAM zero was initially configured to provide input to the core, while SRAM one was configured for output. Sections 6.3.1 and 6.3.2 detail how the core was interfaced to the physical SRAM memories, using the JBits RTR I/O classes.

6.3.1 The *SRAM* Core

An *SRAM* RTP core was created using the core template specification to hide the Slaac1V SRAM hardware interfacing details from the designer. Using the RTR JBits I/O calls provided by the *Board* class, the *SRAM* core attaches the appropriate SRAM control, address, and data signals to their respective JRoute IOB *Pins*, and configures the necessary IOB resources for either input or output depending on the signal. Because signals are mapped to IOB resources based on their name attribute, the physical targeted SRAM is selected by the choice of memory signal names. As a result, the core provides a general abstraction for defining interfaces to any SRAM available to a particular processing element. Figure 6.6 shows the declaration of SRAM address and data signals for SRAMs zero and one on the Slaac1V board. Note that the two SRAMs are distinguished by the choice of names; in this case "XP_MEM0_" designated SRAM memory one, while "XP_MEM1_" designated SRAM memory two. These signal names correspond to the SRAM net names provided in the Slaac1V constraints file.

```
/* define memory signals for Slaac1V X2 memories 0 and 1 */

Bus addr[] = new Bus[2];
Bus data[] = new Bus[2];

/* memory address */

addr[0] = new Bus("XP_MEM0_ADDR", null, 18); /* SRAM 0 */
addr[1] = new Bus("XP_MEM1_ADDR", null, 18); /* SRAM 1 */

/* memory data */

data[0] = new Bus("XP_MEM0_DATA", null, 12); /* SRAM 0 */
data[1] = new Bus("XP_MEM1_DATA", null, 12); /* SRAM 1 */
```

Figure 6.6: Code showing how Slaac1V SRAMs are distinguished through signal names.

The external signals that connect to the port interface of a *SRAM* core are defined in a separate *SRAMProperties* class. An *SRAMProperties* object is passed to the *SRAM* core constructor, thereby allowing external signals to be associated with ports. Figure 6.7 shows the signals and port interface of the *SRAM* core. This class includes methods for setting and retrieving standard memory signals, including chip enable, write enable, address, and data I/O signals. Because the data direction is known for each signal field of the *SRAMProperties* class, the IOBs are configured within the *SRAM.implement()* method using *Board* calls, thereby hiding the configuration calls away from the user.

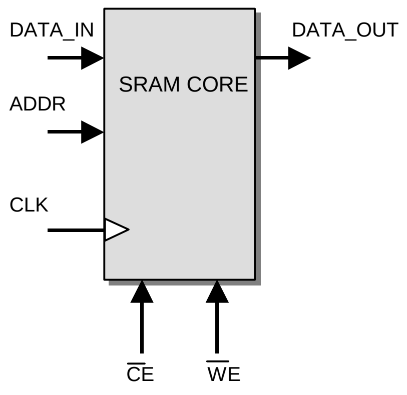


Figure 6.7: SRAM RTP core port interface

6.3.2 The *SlaacWavelet* Class

The *SlaacWavelet* class contains the code necessary for generating a complete two-dimensional DWT system from a null Virtex bitstream. By defining the appropriate cores and signal interconnects, the class interfaces the *DWT2D* core to two SRAM memories on the Slaac1V board. This section describes how the *DWT2D* core is interfaced to the Slaac1V hardware.

Several RTP cores are required for the system implementation. A *DWT2D* core performs the forward transform, and two *SRAM* cores provide memory interfaces that connect to the transform core. A *Slaac1VBoard* class provides an abstraction of the physical Slaac1V platform. Signals are defined that attach to the control, data, and address lines of the two *SRAM* cores. The name attributes given to these signals corresponds to the net names used in the Slaac1V users constraints file. This allows the nets to map to the correct IOBs on the FPGA device.

The memory signals are passed to the *DWT2D* constructor and define core I/O. The signals are associated with the physical SRAM signals by instancing and defining two *SRAMProperties* classes. The *SRAMProperites* instances, along with the *Slaac1V Board* instance define the two *SRAM* core external signal connections. Implementing the *SRAM* cores associates the memory signals with I/O cores defined on the *Slaac1VBoard* object. After these cores are implemented, the *Slaac1VBoard* is implemented, thereby attaching JRoute *Pins* to each signal, and also configuring the appropriate IOBs for input or output. The memory signal connections are implemented in the bitstream using separate *Bitstream.connect()* calls for each signal.

6.4 Summary

Two SRAMs are required by the *DWT2D* core to store image data and transform output. Access to these memories is obtained by routing the core directly to the FPGA IOBs that connect to the physical Slaac1V SRAM signals. It is desirable to perform all IOB interfacing and configurations using JBits, rather than using an I/O framework generated by an additional tool flow. To meet this goal, a set of classes was developed that automated the interfacing of a top-level RTP core to surrounding FPGA IOB resources. These classes were developed as a coauthored research effort. An additional *SRAM* core provides a software abstraction of the Slaac1V SRAM and was used in conjunction with the JBits RTR I/O classes to interface the *DWT2D* core to two SRAM banks on the Slaac1V board.

Chapter 7

Discussion of Operation

Performing the discrete wavelet transform on the Slaac1V board required the development of a host application to control the FPGA. This chapter introduces the *RunWavelet* host application and describes its responsibilities in the PC-FPGA shared processing environment. The operation of the *RunWavelet* application is discussed, including the parameters that define its functionality. A brief discussion is given on the two output files generated by the host application.

7.1 The Host Application

The *DWT2D* core was executed in a PC-FPGA shared processing environment. In this case, a host PC controlled the operation of the Slaac1V board. Control of the board was administered through a series of calls to Slaac1V API functions. The Slaac1V API included functions for writing and retrieving data from the Slaac1V memories, as well as loading and reading device configurations. The API also provided functions for performing partial reconfiguration of the FPGA processing elements. These function calls were implemented inside of the host application, *RunWavelet*. The *RunWavlet* application was written in C++. This application was responsible for loading configurations onto the correct processing element, executing the DWT, and storing the output to file.

7.2 Parameterization

Several parameters defined the operation of the *RunWavelet* application. A bitmap file identifier specified the image to be transformed. Only bitmap files were supported by the application. The MS bitmap format was acceptable since it provided an uncompressed representation of the image data. The host application generated two files containing different representations of the transform output. These output files were specified by two filename string parameters. The operating frequency of the Slaac1V's programmable clock was specified by a clock-frequency value in MHz. Finally, the number of DWT decomposition levels to be performed was determined by an application parameter.

7.3 Operation

The *RunWavelet* application began by creating a *Slaac1VBoard* object. Communication with the board via the host code occurred through method calls to the *Slaac1VBoard* instance. An initialization call readied the board for use. The Slaac1V programmable clock, *PCLK*, was stopped before loading the configuration bitstream onto the FPGA. The bitstream containing the *DWT2D* core and additional I/O interfacing was then loaded on processing element X2. An assertion of the global set/reset (GSR) signal placed the bitstream in the initial state, and ensured all register values were cleared.

The host program loaded an image into memory before executing the transform. The image coefficients were read from a bitmap file and stored in the SRAM 0 of the Slaac1V's processing element X2. Functions were written in C++ to load and save MS bitmap files. Only 256 grayscale images were used in the testing of the transform core. Each pixel value was masked with 0xff to obtain the corresponding eight-bit grayscale value as it was read from the input file.

After loading the image, the transform was ready for execution. The host application determined the appropriate number of cycles to step the PE, based on the specified number of DWT decomposition levels. Additional clock steps were included to compensate for the latency generated by the filters and SRAM. The clock was then run for the corresponding number of clock cycles.

After executing the transform, the host application recorded the core output into two separate files using different formats. In the first format, the output was saved as a bitmap image representation. This allowed easy viewing of the transformed output using standard graphics packages. Saving the output in bitmap format required scaling the coefficients to fit in the range of 0 to 255, however. Because the scaling process was lossy, the data stored in bitmap file could not be used in the inverse transform. The second output format was a raw, unscaled integer dump of the transformed image coefficients. The coefficients in this file were used for in the reconstruction process.

Chapter 8

Results and Analysis

8.1 Overview

This chapter presents the results and findings of this research. The performance of the *DWT2D* core is evaluated using several performance metrics. These metrics include speed of operation, time required for reconfiguration, and resource consumption. The speed of the *DWT2D* core is compared and contrasted against other wavelet transform implementations. The maximum clock speeds of FIR filter cores with varied resolution and number of taps are given. The chapter also provides the reconfiguration times obtained using partial reconfiguration. Bitstream generation and reconfiguration times were obtained using JDK1.2.2 with the HotSpot Performance Engine under Windows 2000. The machine used was a one GHz Pentium III with one gigabyte of RAM.

8.2 Validation

A bitstream was generated using the *SlaacWavelet* Java application. The *DWT2D* core was parameterized with 12-bit filters using the orthogonal Daubechies's N=3 wavelet filter coefficients. The filters used eight bits of coefficient precision. The bitstream was executed using the *RunWavelet* application. A three-level DWT decomposition was performed on the *peppers* grayscale image. Figure 8.1 shows a three-dimensional plot of the pixel intensity values for the untransformed *peppers* image. It is important to note that the image energy is distributed across the pixel array. With this in mind, the output of the *DWT2D* core was observed after the DWT was executed.

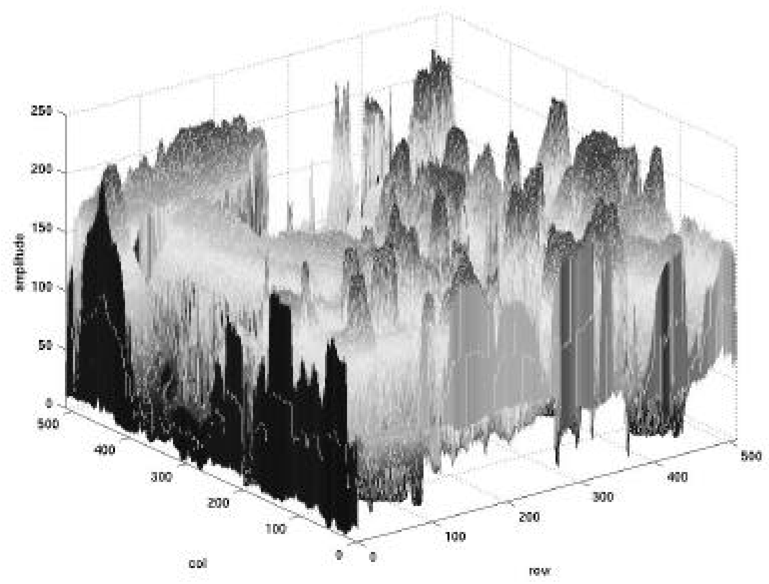


Figure 8.1: Plot showing the pixel values for the original *peppers* grayscale image

Figure 8.2 shows a three-dimensional plot of the output generated by the *RunWavelet* application after a three-level DWT transform of the *peppers* image. Note that the *x* and *y* axis of the plot correspond to the rows and columns of the 512×512 output array. The graph shows that the signal energy from the original image has been decorrelated and then concentrated in a much smaller region corresponding to upper-left sub image as expected. This concentration of signal energy shows how large compression ratios can be achieved by removing the extraneous information contained in the less-important subbands.

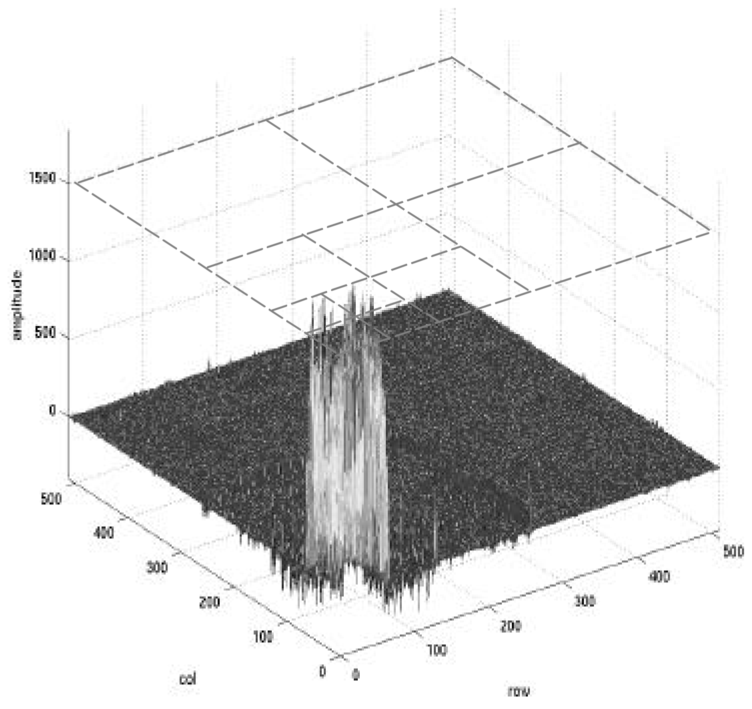


Figure 8.2: Plot showing transformed output resulting from a three-level DWT decomposition of the *peppers* image using Daubechies's N=3 wavelet.

8.3 *DWT2D* Results

Table 2 shows the performance statistics of the *DWT2D* core using 12-bit high-pass and low-pass filters for a 512×512 grayscale image using several common wavelet filter configurations. The first column specifies the number of filter taps in terms of *l/h*, where *l* represents the number of low-pass filter taps, and *h* represents the number of high-pass filter taps. The 5/3 and 9/7 configurations correspond to the lossless and lossy JPEG200 wavelet filters, respectively. The orthogonal Haar wavelet filters defined the 2/2 configuration. The 6/6 filter bank used the Daubechies's N=3 orthogonal, compactly supported wavelet filters. The second column of Table 2 shows the maximum clock speed of the *DWT2D* core for each filter bank configuration. The *JBits to Bitstream* column provides the time interval required to run the *SlaacWavelet* Java application the time interval required to run the *SlaacWavelet* Java application class, which includes reading a null bitstream, instantiating a *DWT2D* core, routing the core, making the necessary bitstream modifications, and writing the modified bitstream to a file. It does not, however, include routing to IOBs. Inclusion of the IOB routing calls required an additional 23.554 seconds to execute. The maximum clock frequency, therefore, is based on routing delays internal to the core, and does not include any delays incurred from external interfacing. The *Filter Configuration* column gives the time required to implement both FIR filter cores. The *CLBs* column indicates how many CLBs resources were consumed by each *DWT2D* core configuration.

Filters	Frequency (MHz)	JBits to Bitstream (sec)	Filter Configuration (sec)	CLBs
5/3	84.154	12.978	2.524	450
2/2	84.154	11.909	1.242	280
9/7	84.154	15.642	5.258	770
6/6	84.154	13.910	3.575	600

Table 2: Performance results for the *DWT2D* core, including maximum clock frequency, time to configure a null bitstream, filter bank instantiation time, and CLBs consumed by the core. Clock frequency values were computed using an XCV1000 device with a speed grade of six as parameters for M3.1 timing analyzer.

From Table 2 it is apparent that the time it takes to instantiate the filter bank is a small portion of the total time it takes to execute the *SlaacWavelet* application. The 6/6 12-bit filter bank consumes 480 CLBs. The 6/6 filter bank configuration, therefore represents 80 percent of the total core resources (600), however instantiation of the core only takes 25.7 percent of the total time required to generate the bitstream. This comparison shows that there is significant overhead outside of the filter core instantiations. Figure 8.3 compares the amount of time needed by different processes (i.e. time required for a bitstream read, time needed to implement the *DWT2D* core, etc.) relative to the total time required for the *SlaacWavelet* application to run.

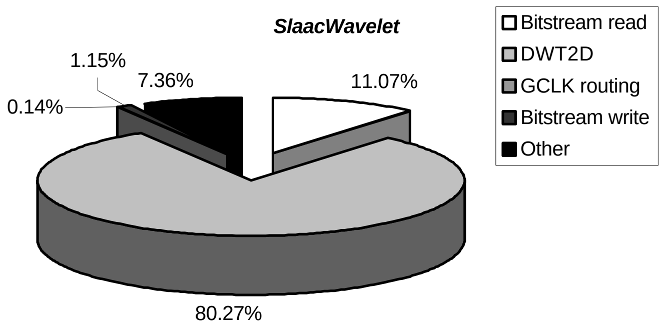


Figure 8.3: Comparison of the times required for several processes to execute within the *SlaacWavelet* application. The application required a total of 13.91 seconds using the *DWT2D* core with the 6/6 filter bank configuration.

As shown in Figure 8.3, the time required to read the configuration is significantly longer than the time required to write the modified bitstream to file. It is important to note the implication of this comparison. If several reconfigurations are required during the duration of an application's lifetime, the associated reconfiguration times can be reduced if the bitstream is only read once and cached in memory. Because the bitstream write times are small, multiple bitstream writes will not significantly impact the reconfiguration time. Figure 8.3 shows that implementing the *DWT2D* core required 80.27 percent of the *SlaacWavelet*'s execution time. Figure 8.4 provides a similar comparison of the required process times within the *DWT2D* core's *implement()* method.

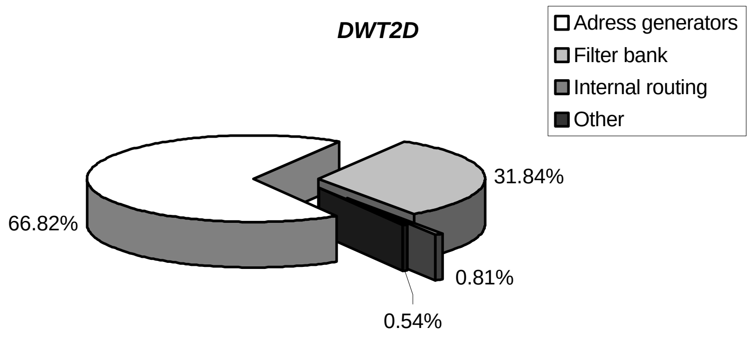


Figure 8.4: Comparison of the times required for several processes to execute within the *DWT2D implement()* method. The method required a total of 11.17 seconds using a 6/6 filter bank parameter.

Figure 8.4 shows the time required to implement the filter bank is 31.84 percent of the total *DWT2D implement()* time. A change in wavelet requires only a reconfiguration of the filter bank, and the rest of the design can remain static assuming the size of the filters remains constant. As a result, the time associated with implementing the address generator cores can be eliminated, since the logic remains static between reconfigurations. Partial reconfiguration can be used to configure only the dynamic part of the design, and further reduce reconfiguration times. This concept is elaborated upon in Section 8.4. The times incurred by internal core routing, relative to the core implementation time, are shown to be negligible.

The maximum clock frequency of the *DWT2D* core was determined to be 84.1 MHz. This number was obtained by using the XDL core output option in JBits. The XDL file was converted to NCD format and then run through Xilinx Foundation Series 3.1 Timing Analyzer tool. At this frequency, the core had a throughput of 1.01 Gbps using 12-bit I/O buses. A one-level transform of a 512×512 image, therefore, required 6.23 msec. Table 3 compares the *DWT2D* one-level transform period of a 512×512 image against other DWT implementations.

	Benkrid et al [39]	*DWT2D*	TMS320C62x [37]	StarCore [36]
Period (msec)	3.50	6.23	15.8	27.2

Table 3: Comparison of one-level DWT transform period for a 512×512 image

Several assumptions were made to make the comparison of transform periods. It was assumed that the StarCore processor operates at 300Mhz, as stated in [36]. It takes approximately 510,000 cycles to transform a 128×128 pixel tile. At 300Mhz, this gives a period of 1.7 msec for a 128×128 pixel tile. A 512×512 image has sixteen 128×128 tiles, and therefore requires 27.2 msec. The vertical wavelet transform function for the TMS320C62x library requires 4144 cycles for 512 columns. The horizontal wavelet transform function requires 2058 cycles. With 512 rows, this gives the total number of cycles required to perform a one-level DWT as *512 x 2058 (row transform) + 512 x 4144* *(column transform) = 3175424 cycles.* Assuming a clock frequency of 200 MHz, this gives a transform period of 15.8 msec.

As shown in Table 3, the *Benkrid et al* design performs a one-level DWT 78 percent faster than the RTR *DWT2D* core. This is a reasonable increase since the *Benkrid et al* design uses a non-folded architecture with a period of only N^2 clock cycles, where N represents the height and width of the image in pixels. Because the RTR *DWT2D* uses the direct approach in which the filter bank is reused for row and column processing, the core requires $2N^2$ cycles. The *Benkrid et al* design using a biorthogonal 9/7 configuration occupied 4720 Virtex-E slices. In contrast the *DWT2D* core using the 9/7 configuration occupied approximately 1540 Virtex slices. The *Benkrid et al* design uses BRAM to store intermediate data, while the *DWT2D* core requires an additional memory of size N^2. The RTR *DWT2D* core outperforms both the DWT software implementations for the StarCore and TMS320C6x series processors.

Table 4 lists the maximum clock speeds and number of required CLBs for *FIRFilter* cores with three different resolutions and varying number of taps. These taps correspond to the wavelet filter configurations given in Table 2. When compared to the frequencies listed given in Table 2, it is apparent the bottleneck occurs in the address generation logic. Further investigation using the M3.1 timing analyzer showed the critical path existing in the output address generation logic.

	8-bit		12-bit		16-bit	
Taps	Freq. (MHz)	CLBs	Freq. (MHz)	CLBs	Freq. (MHz)	CLBs
2	186.71	40	176.44	80	167.67	108
3	177.34	64	172.98	120	166.83	168
5	172.06	104	164.88	210	153.35	276
6	166.81	120	157.36	240	152.86	324
7	171.67	144	151.76	280	145.90	384
9	166.42	192	147.51	370	136.95	504

Table 4: Maximum *FIRFilter* core frequencies and CLBs used for different filter resolutions with varying number of taps. Clock frequency values were computed using an XCV1000 device with a speed grade of six as parameters for the M3.1 timing analyzer.

8.4 Partial Reconfiguration Results

It was desirable to further reduce the reconfiguration times listed in Figure 8.3, as these times were still considerably lengthy. The filter reconfiguration times reported in Figure 8.3 involved a complete instantiation of the filter bank, including core implementation and internal routing. There was significant overhead associated with this approach. If only the filter coefficients were modified, however, the filter reconfiguration times could be reduced significantly. The reduced processing time advantages gained through reconfiguration of the coefficients were extended to the bitstream level using the JBits JRTR partial reconfiguration engine to produce a partial bitstream containing only the frames that have been altered during the reconfiguration process [42].

Given the location of the *DWT2D* core in the FPGA CLB grid, along with a set of new high-pass and low-pass filter coefficients, the JBits application *DWTReconfig* reconfigured the filter coefficients and made the necessary JRTR calls to produce the partial bitstream. Modification of the filter coefficients involved changing the KCM constant values. *DWTReconfig* mimicked the placement behavior used by the FIR filter core to determine the location of each KCM core in the filter. New values were then computed for each ROM within a given KCM, and the underlying LUT arrays were modified accordingly using a series of *JBits.set()* calls. Table 5 shows the time required to perform a reconfiguration of both FIR filters, the time required to write the partial bitstream file, and the size of the resulting partial bitstream under different filter bank configurations. For each configuration, the number of taps in the high-pass and low-pass filters was set equal to balance latency.

	9/9	6/6	5/5	3/3
Filter Reconfiguration	0.122 sec	0.120 sec	0.121 sec	0.120 sec
Partial Bitstream Write	0.071 sec	0.060 sec	0.050 sec	0.040 sec
Partial Bitstream Size	72,234 bytes	48,185 bytes	40,169 bytes	24,137 bytes

Table 5: Times required for filter reconfiguration and writing partial bitstreams for filter banks of varying number of taps. The partial bitstream file size is also reported for each configuration.

It is interesting to note that the reconfiguration times remained fairly constant between the four configurations. These times are a significant improvement on the times reported in Figure 8.3. All four partial bitstream files sizes were smaller when compared to the original *DWT2D* XCV10000 bitstream size of 766,040 bytes. The partial bitstream size decreased with the filter size as well. This is logical, since the JRTR only writes frames to the partial bitstream file that have been altered since the last reconfiguration.

Chapter 9

Conclusions

9.1 Summary

This thesis documented the design of a run-time reconfigurable two-dimensional discrete wavelet transform core for use on Xilinx Virtex FPGAs. The implementation, simulation, debugging, and interfacing design phases were discussed. The design was implemented entirely using the JBits API. This section briefly summarizes the content of each chapter.

In Chapter 1, the concept of RTR was introduced in relation to FPGA designs. Wavelets were discussed in terms of their advantages over traditional transform methodologies and their suitability for image compression and other applications. Following this discussion, the chapter motivated the application of RTR techniques to the wavelet transform. The concept of the shared processing environment was presented to show how an FPGA could function as a coprocessor under the control of a host PC running a JBits application. The JBits environment was presented as an enabler of RTR FPGA designs. The chapter concluded with a discussion of thesis contributions.

Background information was presented in Chapter 2. The JBits environment was further elaborated upon, along with an explanation of the RTP core specification and JBits IOB resources. The DWT was explained in terms of Mallat's pyramid algorithm and its dependency on filter banks. The Slaac1V specifications were provided with a diagram illustrating the system components. A brief survey of existing DWT implementations ended the chapter.

The purpose and composition of the DWT core library was presented in Chapter 3. The overall core hierarchy was illustrated with a figure. The functionality and implementation of each core designed for this research was described.

Chapter 4 presented the implementation of the *DWT2D* core. The chapter began with a design overview, including a system-level diagram of the core. The higher abstraction cores were discussed, including the input and output address generators and FIR filter.

The simulation and debugging design phases were discussed in Chapter 5. A Java-based software implementation of the DWT was used for simulation. By modeling the Java classes after hardware components, the simulator also provided a framework for the core-based hardware design. The chapter discussed bitstream level simulation using the BoardScope graphical debugger. Bitstream level test benches were distinguished from traditional HDL test benches.

Chapter 6 described the process in which the *DWT2D* core was interfaced to the Slaac1V board. The chapter began with a discussion of the need for RTR I/O in JBits designs. A set of JBits RTR I/O classes was designed to automate the process of interfacing a top-level RTP core to FPGA IOBs. Each RTR I/O class is discussed and several code excerpts are provided to illustrate key concepts. The chapter continued with interfacing techniques specific to the Slaac1V board, including a discussion on the *SRAM* core and *SlaacWavelet* class.

The operation of the wavelet transform system was described in Chapter 7. This included an explanation of how the host program controlled the execution of the *DWT2D* core on the FPGA. The technique used to bring image data in and out of the system was also mentioned.

Chapter 8 discussed the results of the research. The *DWT2D* core was found to have a maximum clock speed of 84.154 MHz. This implementation under-performed the *Benkrid et al* FPGA DWT design, but outperformed two DSP implementations as expected. The speeds of several filter resolutions with varying number of taps were presented, with the slowest filter, a 16-bit filter with nine taps, running at 135.95 MHz. The advantages of using partial reconfiguration were also presented.

9.2 Future Work

Although the research accomplished the stated goals, there remains room for improvement and exploration. The structural core-based HDL provided by the RTP core specification is not well suited for complex control logic. As a result, the current *DWT2D* core is based on a folded architecture where filters are reused in order to keep control logic relatively simple. This was an important issue that was considered during the initial stages of design. Although this approach is very straightforward, the folded architecture is wasteful in terms of memory usage and number of operations required per transform period. If the core were going to be used in a commercial application, it would require a redesign that uses a more efficient architecture [6,7,39]. Exploiting parallelism in filters relying on a single data bus requires arbitration, which is difficult to implement without behavioral synthesis.

Another issue with the current design is the lack of a row-extension scheme. Again, this was left out to minimize the complexity of the control logic. This lack of extension scheme invalidates the border pixels of the transformed image. Implementing symmetric reflection at the boundaries would correct the border errors.

Because the wavelet transform design is implemented as a core, it can be easily integrated with other components. The core is particularly well suited for image compression applications. In this case, a quantizer core may be added to the *DWT2D* core, along with an entropy coder to produce an image compression system. With the current research being done to integrate JBits with other design flows, the *DWT2D* core could be integrated with systems generated in other languages.

9.3 Conclusions

The majority of goals defined at the start of the research were completed. A functional RTR wavelet transform core was created and tested on the Slaac1V PCI-accelerator board. The core illustrates how RTP cores can be used in a hierarchical manner to create a high-level system with relatively simple control logic. The design was successfully interfaced to two SRAM banks and achieved a high data throughput rate without using other tools.

Although a simple design approach was used, the speed of the core was reasonable when compared against other DWT implementations. A redesign of the control logic could possibly eliminate the critical path, and increase the design speed dramatically.

The reconfiguration times achieved through a complete bitstream reconfiguration are still too lengthy for most applications. Using constant folding to change filter coefficients, and partial reconfiguration to update the bitstream provides more reasonable reconfiguration times, however.

The RTR DWT design distinguishes itself from other designs in two regards. The *DWT2D* core is one of the first FPGA designs with a dependency on external I/O to be developed without any reliance on the traditional vendor tools. Previously, designs using I/O required interfacing to a framework generated externally. Secondly, the design offers a wavelet transform core with a degree of parameterization and reconfigurability that is usually available only in software implementations.

Bibliography

[1] Information Sciences Institute – East, SLAAC Project Page, World Wide Web page, http://www.east.isi.edu/projects/SLAAC.

[2] S. Guccione and D. Levi, "XBI: A Java-Based Interface to FPGA Hardware," *Configurable Computing Technology and its uses in High Performance Computing, DSP and Systems Engineering*, Proc. SPIE Photonics East, SPIE – The International Society for Optical Engineering, Bellingham, WA, November 1998.

[3] S. Guccione and D. Levi, "Run-Time Parameterizable Cores," *Proceedings of the 9th International Workshop on Field Programmable Logic and Applications*, Lecture Notes in Computer Science 1673, pp. 215-222, 1999.

[4] J. Proakis and D. Manolakis, *Digital Signal Processing*. Prentice Hall 1996.

[5] Joint Photographic Experts Group, World Wide Web page, http://www.jpeg.org/JPEG2000.htm, 2001.

[6] C. Chakrabarti and C. Mumford, "Efficient Realizations of Encoders and Decoders Based on the 2-D Discrete Wavelet Transform," *IEEE Trans. of VLSI Systems*, pp. 289-298, 1999.

[7] C. Chakrabarti, M. Vishwanath, and R. Owens, "Architectures for Wavelet Transforms: A Survey," *J. of VLSI Signal Processing Systems for Signal Image and Video Technology*, pp. 171-192, Nov. 1996.

[8] I. Daubechies, “The Wavelet Transform, Time-Frequency Localization and Signal Analysis,” *IEEE Trans. Inform. Theory*, vol. 36, pp. 961-1005, Sept. 1990.

[9] A. Croisier, D. Esteban, and C. Galand, “Perfect Channel Splitting by Use of Interpolation/Decimation/Tree Decomposition Techniques,” *International Conference of Information Science and Systems*, 1976.

[10] I. Daubechies, *Ten Lectures on Wavelets*, Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, pp. 195, 1992.

[11] K. Chapman, “Fast Integer Multipliers Fit in FPGAs,” *Electronic Design News*, May 12, 1994.

[12] Virtex^{TM}-E 1.8 V Field Programmable Gate Arrays – Preliminary Product Specification, http://www.xilinx.com/partinfo/ds022.pdf, DS022 (v1.9) February 12, 2001.

[13] E. Keller, "Dynamic Circuit Specialization of a CORDIC Processor," *Reconfigurable Technology: FPGAs for Computing and Applications II*, Proc. SPIE Photonics East, SPIE - The International Society for Optical Engineering, November 7-8, 2000.

[14] E. Lechner and S. Guccione, “The Java Environment for Reconfigurable Computing,” *Field-Programmable Logic and Applications*, pp. 284-293, Springer-Verlag, Berlin, September 1997. Proc. of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997.

[15] S. Nisbet and S. Guccione, “The XC6200DS Development System,” *Field-Programmable Logic and Applications*, pp. 61-68, Springer-Verlag, Berlin, September 1997, Proceedings of the 7th International Workshop on Field-Programmable Logic and Applications, FPL 1997.

[16] G. Knittel, “A PCI-Compatible FPGA-Coprocessor for 2D/3D Image Processing,” *IEEE Symposium on FPGAs for Custom Computing Machines*, pp. 136-145, Los Alamitos, CA, April 1996, IEEE Computer Society Press.

[17] G. Gent, S. Smith, and R. Haviland, "An FPGA-Based Custom Coprocessor for Automatic Image Segmentation Applications," *IEEE Workshop on FPGAs for Custom Computing Machines*, pp. 172-179, Los Alamitos, CA, April 1994, IEEE Computer Society Press.

[18] C. Patterson, "High Performance DES Encryption in Virtex FPGAs using JBits," *IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 2000.

[19] S. Saha and R. Vemuri, "Adaptive Wavelet Coding of Multimedia Images, " In *Proc. ACM Multimedia*, Orlando, Florida 1999.

[20] S. McMillan, B. Blodget, and S. Guccione, "VirtexDS: A Device Simulator for Virtex," *Reconfigurable Technology: FPGAs for Computing and Applications II*, Proc. SPIE 4212, pp. 50-56, Bellingham, WA, November 2000, SPIE – The International Society for Optical Engineering. November 2000.

[21] P. Sundararajan and S. Guccione, "XVPI: A Portable Hardware / Software Interface for Virtex," *Reconfigurable Technology: FPGAs for Computing and Applications II, Proc. SPIE 4212*, pp. 90-95, Bellingham, WA, November 2000, SPIE – The International Society for Optical Engineering. November 2000.

[22] E. Keller, "JRoute: A Run-Time Routing API for FPGA Hardware," *7th Reconfigurable Architectures Workshop*, Lecture Notes in Computer Science 1800, pp. 874-881, Cancun, Mexico, May 2000.

[23] Xilinx Application Note: Virtex Series, *Virtex Series Configuration Architecture User Guide*, XAPP151 (v1.4), August 3, 2000.

[24] S. Mallat, "A Theory for Multiresolution Signal Decomposition: The Wavelet Representation," *IEEE Trans. Pattern Analysis and Mach. Intell*, 11(7), pp. 674-693, July 1989.

[25] S. Mallat, "Multifrequency Channel Decompositions of Images and Wavelet Models," *IEEE Trans. Acoustics Speech and Sig. Proc*, 37(12), pp. 2091-2110, Dec 1989.

[26] S. Guccione and D. Levi, "The Advantages of Run-Time Reconfiguration." *Configurable Computing Technology and its Uses in High Performance Computing, DSP and Systems Engineering*, Proc. SPIE Photonics East, SPIE - The International Society for Optical Engineering, pp. 87-92, Bellingham, WA, September 1999.

[27] F. de Dinechin and V. Lefvre, "Constant Multipliers for FPGAs," in *Second International Workshop on Engineering of Reconfigurable Hardware/Software Objects (ENREGLE 2000)*, Las Vegas, Nevada, June 2000.

[28] Xilinx, Inc., World Wide Web page, http:///www.xilinx.com, 2001.

[29] J. Hess, D. Lee, S. Harper, M. Jones, and P. Athanas, "Implementation and Evaluation of a Prototype Reconfigurable Router," *IEEE Symposium on FPGAs for Custom Configurable Computing Machines*, Napa, California, April 1999.

[30] R. Bittner Jr. and P. Athanas, "Wormhole Run-Time Reconfiguration," in *Proc. 5th International Symposium on Field Programmable Gate Arrays*, Monterey, California, 1997.

[31] J. Eldredge and B. Hutchings, "Run-Time Reconfiguration: A Method for Enhancing the Functional Density of SRAM-Based FPGAs," in *Journal of VLSI Signal Processing*, Volume 12, 1996.

[32] S. Guccione, D. Levi, and D. Downs, "A Reconfigurable Content Addressable Memory," *Parallel and Distributed Processing*, Springer-Verlag, Berlin, May 2000, Proceedings of the 15th International Parallel and Distributed Processing Workshops, IPDPS 2000.

[33] JPEG Frequency Asked Questions, World Wide Web page, http://www.faqs.org/faqs/jpeg-faq/, 2001.

[34] Overview of the MPEG-4 Standard, World Wide Web page, http://www.cselt.it/mpeg/standards/mpeg-4/mpeg-4.htm, 2001.

[35] The MathWorks, Inc., World Wide Web page, http://www.mathworks.com, 2001.

[36] S. Twelves, M. Wu, and A. White, *JPEG2000 Wavelet Transform Using StarCore: Application Note*, http://www.motorola.com, 2001.

[37] Texas Instruments, Inc., *TMS32062x Image/Video Processing Library Programmer's Reference*, March 2000.

[38] Analog Devices, Inc., World Wide Web page, http://www.analog.com, 2001.

[39] A. Benkrid, D. Crookes, and K. Benkrid, "Design and Implementation of a Generic 2-D Biorthogonal Discrete Wavelet Transform on an FPGA," *IEEE Symposium on FPGAs for Custom Computing Machine*s, Rohnert Park, CA, April, 2001.

[40] C. Dick, B. Turney, and A. Reza, "Configurable Logic for Digital Signal Processing," Xilinx Application Note: Other FPGA Applications, April 28, 1999.

[41] P. Wasilewski, "Two-Dimensional Discrete Wavelet-transform implementation in FPGA device for real-time image processing," *Wavelet Applications in Signal and Image Processing V*, Proc. SPIE Vol. 3169, SPIE – The International Society for Optical Engineering, Bellingham, WA, August 1997.

[42] S. McMillan and S. Guccione, "Partial Run-Time Reconfiguration Using JRTR," *Field-Programmable Logic and Applications*, pp. 352-360, Proc. of the 10th International Workshop on Field-Programmable Logic and Applications, FPL 2000.

[43] J. Ballagh, P. Athanas, and E. Keller, "Java Debug Hardware Models using JBits," *8th Reconfigurable Architectures Workshop*, San Francisco, CA, April 27, 2001.

Vita

Jonathan Ballagh was born on August 19th 1977 in historic Leesburg, Virginia. He began his collegiate studies at Virginia Tech in the fall of 1995 as a Computer Science major. After spending his freshmen year contemplating future career goals, he decided to switch into engineering. During his undergraduate studies, Jonathan spent three summers working as an intern at E.I.T. designing and programming a variety of test fixtures for printed circuit boards. He completed his undergraduate studies in May 1999 and graduated with a B.S. degree in Computer Engineering. Following graduation, Jonathan remained at Virginia Tech in pursuit of a master's degree in Electrical Engineering. His curiosity and interest in FPGAs led him to the Configurable Computing Laboratory, where he worked primarily on run-time reconfigurable tools and applications for FPGA based hardware platforms. Jonathan spent the summer of 2000 working as an intern at Xilinx in Boulder, Colorado. Upon completion of his M.S. degree in Electrical Engineering, Jonathan plans to return to Colorado, where he'll begin his new job at Xilinx.