IMAGE WAVELET COMPRESSION IMPLEMENTATION USING A RUN-TIME RECONFIGURABLE CUSTOM COMPUTING MACHINE

by

Zhimei Ding

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

APPROVED:

_____

Dr. Peter M. Athanas

_____ _____

Dr. Mark T.Jones Dr. Scott F. Midkiff

June 2000

Blacksburg, Virginia

Keywords: Image Wavelet Compression, Janus, FPGA, RTRC, CCM

IMAGE WAVELET COMPRESSION IMPLEMENTATION USING A RUN-TIME RECONFIGURABLE CUSTOM COMPUTING MACHINE

By

Zhimei Ding

Peter M. Athanas, Chairman

Electrical Engineering

(ABSTRACT)

This thesis presents the design and implementation of the Image Wavelet Compression (IWC) algorithm on Field Programmable Gate Arrays (FPGAs) by using the run-time reconfigurable custom computing machine design tool Janus. The four routines implementing the IWC are discussed. The structure of Janus is introduced and the IWC implementation design framework to use Janus structure is described in detail. The Janus hardware circuit design model, which has been used in the IWC implementation, is demonstrated here. The hardware implementation results are presented and analyzed, focusing on reconfiguration and computing time. Future research areas are suggested to improve the Janus tool.

ACKNOWLEDGEMENTS

I sincerely appreciate the time, advice, and encouragement given by my advisor Dr. Peter M. Athanas, and committee members Dr. Mark T. Jones and Dr. Scott F. Midkiff. Gratitude is also extended to Mr. Yifan Chen, Mr. David Lehn and Ms. Wendy M. Akers for their help and contributions to my graduate research. I am grateful for the opportunity to be a member of the Virginia Tech Information Systems Center (VISC). It has been a tremendous learning experience.

I am indebted to my parents and parents-in-law for their unfailing confidence and support in everything I do.

I would like to thank my husband, Fengfeng Tao, for his love, support, and understanding. I deeply appreciate everything he did for me and I am so glad that he is always on my side.

Especially, this thesis is to my lovely son Liangyu Tao.

Table of Contents

List of Figures

List of Listings and Tables

Chapter 1

Introduction

1.1 Motivation

There exists three popular methodologies for designing computing hardware: application-specific integrated circuits (ASICs), programmable processors, such as microprocessors or DSPs, and configurable computing using Field Programmable Gate Arrays (FPGAs). The tradeoff among these is between flexibility and efficiency. First, ASICs are highly specialized for a given application. They can achieve the best possible performance with the lowest silicon cost, but this high efficiency sacrifices the flexibility for they are only useful for one task. Second, microprocessors and DSPs provide some limited and fixed set of arithmetic and control operations that can be organized and sequenced to achieve computations. These are often inefficient when special computational operations are needed or if the computation needs to be executed in parallel or pipeline. Third, Configurable Computing Machines (CCMs) have emerged as a hybrid between ASICs and programmable processors. They rely on RAM-based field-programmable gate arrays as the mechanism to allow hardware structures to fit the natural organization and dataflow of a computation. CCMs allow developers to design their own special function according to their needs and achieve the concurrency and pipeline advantages inherent in the computation [7].

Some complex applications, such as Image Wavelet Compression (IWC), need several continuous configurations and computations because of the limited resources provided by the CCM board. Configuring the platform during the execution phase is also called Run-Time Reconfigurable Computing (RTRC). In RTRC, the computation is divided into a series of sequenced stages where each stage fits onto the CCM. Intermediate results from each stage are saved into memory and passed on to later stages. The process repeats until all stages are executed and final results are gathered. Widespread use of run-time reconfigurable custom computing depends upon the existence of high-level automated design tools. The Java-based Janus tool, designed at Virginia Tech's Configurable Computing Laboratory, is such a design tool. With this tool, developers create a series of Java classes including a class describing the structural behavior of the application, and classes describing circuits using JHDL (Just another Hardware Description Language). The design framework allows hardware and software modules to be freely intermixed. During the compilation phase of the development process, the Janus tool analyzes the structure of the application and adapts it to the target architecture [1]. Janus also provides the software simulation environment and seamlessly allows the developer to switch to hardware executions without changing the design architecture. The implementation of the IWC is a good example of an application developed using the Janus toolset.

1.2 Contributions of This Research

The purpose of this project is to use and test the Janus tool to implement the IWC application in a run-time environment on the Wildforce configurable computing board. Several different types of 512x512 pixel size images are successfully compressed on FPGAs using our

design. Images are also compressed using multiple processing elements. The hardware execution results are exactly the same as those obtained with the software runs.

Based on the wavelet compression algorithm, the problem has been partitioned into a series of stages and then sequentially reconfigured and processed on the FPGAs. For each stage, the host-side software I/O component and the FPGA-side hardware component have been developed. The steps that developers take when using the Janus tool to design applications are discussed in detail in this thesis. The Janus hardware design model is used in the IWC design. The reconfiguration time, hardware computation time, and overhead times measured in an application's execution are collected and analyzed to show the efficiency of the Janus tool. Future research and enhancement of the Janus tool are suggested in this thesis. In summary, this project has implemented the Image Wavelet Compression algorithm on FPGA in a run-time reconfigurable computing environment. Janus tool have been used and tested ??

1.3 Organization of Thesis

This thesis is organized as follows. Chapter 2 presents the relevant concepts of the Image Wavelet Compression algorithm. Chapter 3 describes the Wildforce board architecture, including a description of the FPGA hardware and the host code used to communicate with the board. The Janus toolset is introduced in Chapter 4. Chapter 5 describes the Image Wavelet Compression implementation on the Wildforce FPGA board. It covers the Janus application design process, application partition, the Janus Memory-PE interface, the Janus hardware design model, and simulation environments. The IWC implementation is used as an example in describing these models. In Chapter 6, IWC hardware execution results will be analyzed. Chapter 7 presents conclusions and identifies issues for future research.

Chapter 2

Image Wavelet Compression Concept

2.1 Software Implementation of Image Wavelet Compression Algorithm

This research is based on the Image Wavelet Compression C code, which is provided by Honeywell Technology Center [2]. The IWC code contains all the routines required for a simple

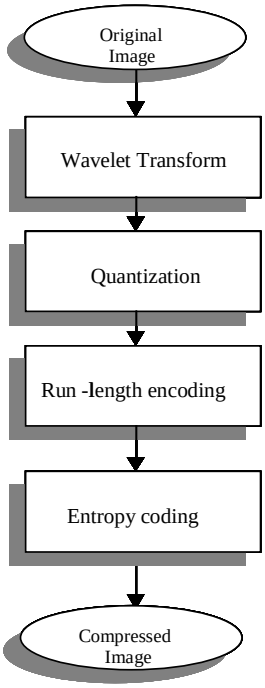


Figure 1. Image Wavelet Compression routines.

wavelet-based image compression of a 512x512 8-bit pixel image in PGM format. The compression process consists of four basic steps: wavelet transform, quantization, run-length encoding, and entropy coding. This is shown in Figure 1.

The IWC code accepts an optional compression-factor parameter specifying how aggressively the image should be compressed. A compression factor indicates minimal compression and maximum image quality. The compression factor of 255 indicates maximum compression with higher degradation to the image quality. If the compression factor is not indicated, a default compression rate of 128 will be used. The compressed output image file format is specific to this program, and has a ".cmp" extension [2].

2.2 Wavelet Transform Routine

The first step, the wavelet transform routine process, is a modified version of the biorthogonal Cohen-Daubechies–Feuvear wavelet. Wavelet transforms have received significant attention and are widely used for signal and image processing. For example, they are widely used in image coding, image compression, and speech discrimination. The basic concept behind wavelet transform is to hierarchically decompose an input signal into a series of successively lower resolution reference signals and their associated detail signals. At each level, the reference signal and the detail signal contain the information needed to reconstruct the reference signal at the next higher resolution level [3].

2.2.1 One-Dimensional Wavelet Transform

The one-dimensional discrete wavelet transform can be described in terms of a filter band as shown in Figure 2. An input signal *x[n]* is applied to the low pass filter *l[n]* and to the analysis

high-pass filter *h[n]*. The odd samples of the outputs of these filters are then discarded, corresponding to a decimation factor of two. The decimated outputs of these filters constitute the reference signal *r[k]* and the detail signal *d[k]* for a new-level of decomposition. During reconstruction, interpolation by a factor of two is performed, followed by filtering using the low-pass and high-pass synthesis filters *l[n]* and *h[n]*. Finally, the outputs of the two synthesis filters are added together.

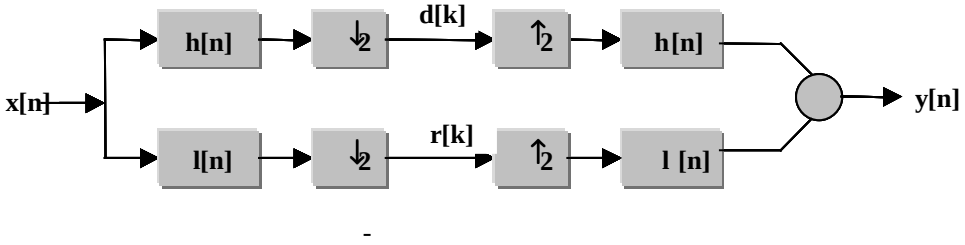


Figure 2. One-dimensional wavelet transform.

The above procedure can be expressed mathimatically as the following equations.

$$d[k] = \sum_n x[n] \bullet h[2k - n] \tag{1}$$

$$r[k] = \sum_n x[n] \bullet l[2k - n] \tag{2}$$

$$x[n] = \sum_n (d[k] \bullet g[-n + 2k]) + (r[k] \bullet h[-n + 2k]) \tag{3}$$

2.2.2 Multilevel Decomposition Wavelet Transform

For a multilevel decomposition, the above process is repeated. The previous level's lower resolution reference signal $r_i[n]$ becomes the next level sub-sampling input, and its associated detail signal $d_i[n]$ is obtained after each level filtering. Figure 3 illustrates this procedure. The original signal *x[n]* is input into the low-pass filter *l[n]* and the high-pass filter *h[n]*. After three

levels of decomposition, a reference signal $r_3[n]$ with the resolution reduced by a factor of 2^3 and detail signals $d_3[n]$, $d_2[n]$, $d_1[n]$ are obtained. These signals can be used for signal reconstruction.

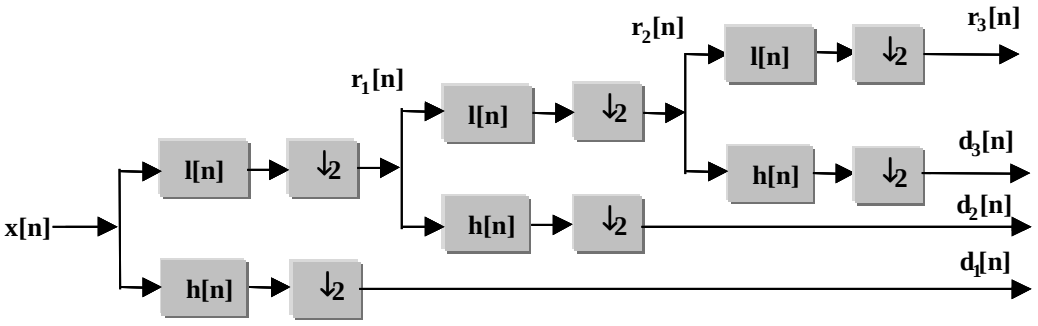


Figure 3. Three-level decomposition for wavelet transform.

The wavelet transform routine in the IWC code employs a shifting scheme to simplify the wavelet implementation. Therefore, it only requires integer adds and shifts, which make it easier to implement on hardware. The computation of the wavelet filter is performed according to the following equations.

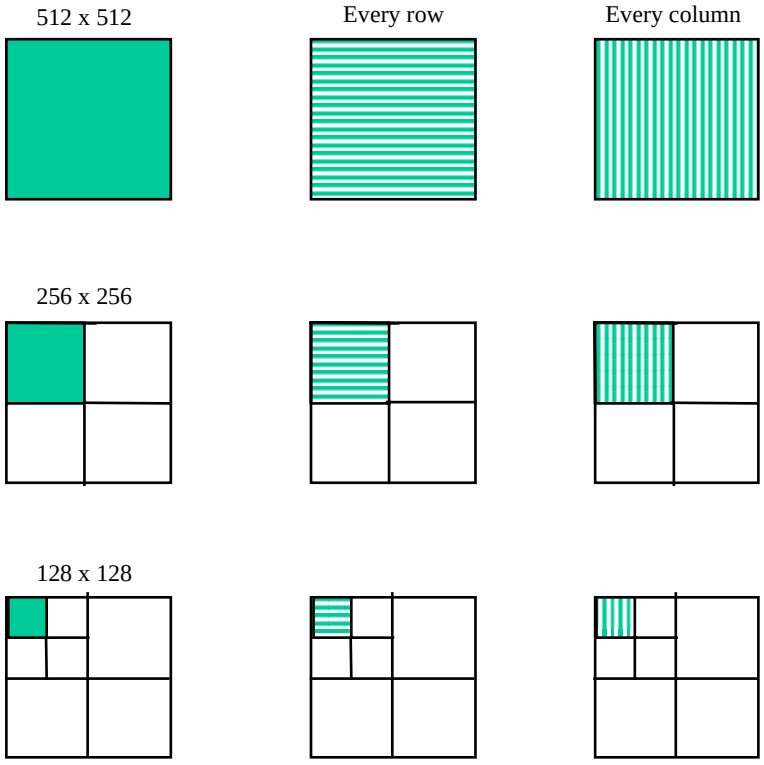
$$D_0 = D_0 + D_0 - S_0 - S_0 \tag{4}$$

$$S_0 = S_0 + (2 * D_0 / 8) \tag{5}$$

$$D_i = D_i + D_i - S_i - S_{i+1} \tag{6}$$

$$S_i = S_i + ((D_{i-1} + D_i) / 8) \tag{7}$$

In the above equations, D_i and S_i are odd and even pixels taken from one row or column,



For every row or column do:

S0 D0 S1 D1 S2 D2 S3 D3 …

$$D_0 = D_0 + D_0 - S_0 - S_0$$
$$S_0 = S_0 + (2 * D_0 / 8)$$
$$D_i = D_i + D_i - S_i - S_{i+1}$$
$$S_i = S_i + ((D_{i-1} + D_i) /8)$$

Figure 4. Wavelet transform implementation.

respectively. In image compression, one row or column of an image is regarded as a signal. Calculation of the wavelet transform requires pixels taken from one row or column at a time. In

Equations (4) – (7). D_i should be calculated before processing S_i. Therefore, the odd pixel should be processed first, then the even pixel due to the data dependency. There are a total of three levels based on the 3-level decomposition wavelet transform algorithm discussed above. In each level, the rows are processed first then the columns. Each level's signal length (amount of each row/column pixels) is half of the previous level.

Equations (4) – (7) are grouped into a function called `Forward-wavelet` in the C code. Figure 4. illustrates the three levels of wavelet transform implementation.

2.3 Quantization Routine

After the three levels of the wavelet transform, the quantization routine follows. During the quantization routine, the image is divided into 10 blocks; the first four will be 64 x 64 pixels (4096 pixels), then three will be 128 x 128 (16384 pixels), and the remaining three of 256 x 256 pixels (65536 pixels). Every block executes the same quantization process. Figure 5 illustrates this as a block diagram.

Before processing each block, some parameters should be prepared. First is the *blockthresh*, which should be provided by the developer. An array is used to hold these 10 block *blockthreshes*:

Blockthresh[10] = { 0, 39, 27, 104, 79, 51, 191, 99999, 99999, 99999}

For example: Block 1 's *blockthresh* is 0, Block 10's *blockthresh* is 99999.

The next values that need to be calculated are the sixteen thresholds for each block, $thresh_1$~$thresh_{16}$; the formula to calculate these value is as follows.

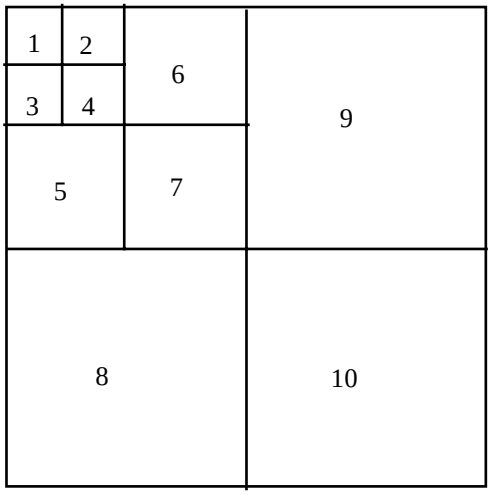


Figure 5-6. Quantization and run-length encoding block diagram.

$$thresh_n = \min + \frac{(\max - \min)\cdot n + 8}{2^4} \qquad (8)$$
$$n : 1 \sim 16$$

The $thresh_n$ is the nth thresh value in a block, and n value is from *1* to *16*. Values *min* and *max* are the minimal and maximum pixel values within this block. After these numbers are computed, each block can run the quantization process. First, each input pixel's absolute value is compared with its corresponding *blockthresh[n]*, if it is smaller than the *blockthresh* value, the original pixel value is assigned to a constant value *ZERO_MARK*, which should be defined by the user. In this project, this is assigned a value of *16*. If the *abs(pixel)* value is not smaller then the *blockthresh* value, the pixel will be passed to a subroutine called *classify(int val)* in *comprss.c* program. The original pixel value will then be changed into its corresponding $thresh_n$ value after this call. Listing 2-1 is the pseudo code to represent the above calculation.

```
if(abs(val)< blockthresh[num]) quant_buf[num][qsize++]=ZERO_MARK;
else
      quant_buf[num][qsize]=classify(val);

/*********************************************************************
 * Routine: int classify(int val)
 *********************************************************************/
int classify(int val)
{
  if(val>thresh8) {
    if(val>thresh12) {
      if(val>thresh14) {
        if(val>thresh15) return 15;
                    else return 14;
      } else {
        if(val>thresh13) return 13;
                    else return 12;
      } } else {
      if(val>thresh10) {
        if(val>thresh11) return 11;
                    else return 10;
      } else {
        if(val>thresh9 ) return 9;
                    else return 8;
      } } } else {
    if(val>thresh4) {
      if(val>thresh6 ) {
        if(val>thresh7 ) return 7;
                    else return 6;
      } else {
        if(val>thresh5 ) return 5;
                    else return 4;
      } } else {
      if(val>thresh2 ) {
        if(val>thresh3 ) return 3;
                    else return 2;
      } else {
        if(val>thresh1 ) return 1;
```

```
                    else return 0;
      } } }
}
```

Listing 2-1: Pseudo code for quantization algorithm.

As described above, the pixel values of the total image have been changed into integer values between *0* to *16* without changing the total image size. Also, because the values *blockthresh[8]*, *blockthresh[9]* and *blockthresh[10]* are very large (value 99999), the absolute pixel value calculated from the wavelet transform routine will not exceed 99999. Therefore, these three block pixel values after the quantization will always be 16. Because of this, the last three blocks do not need to be processed in the IWC implementation design.

2.4 Run-Length Encoding Routine (RLER)

Run-length encoding is the next routine following the quantization process. The purpose of this step is to compress the image size based on the pixel values from quantization, which are between integer value *0* to *16*. The image can be compressed to 10% of the original after the run-length encoding. As in the quantization routine, the image is also divided into 10 blocks. Each block will run the same run-length encoding algorithm. Figure 6 shows the RLER block diagram. Because values in blocks 8, 9, and 10 after quantization are all equal to 16, these three blocks will not be processed. The process pseudo code for each block is shown in Listing 2-2.

```
While (pixel is in this block)
  {
   if ( pixel !=ZERO_MARK (16) )
          saving pixel value in rle-buffer;
    else    {
          count = 0;
             while( pixel ==ZERO_MARK(16))
```

```
                {
                  count++;
                  if((count  ==  256-ZERO_MARK)||(no  more  pixels  in  this
                    block))
                    break;
                  read the next pixel;
                 }
          save (count+ZERO_MARK-1) in rle-buffer;
      }
           read the next pixel;
  }
```

Listing 2-2: Pseudo code for each block run-length encoding algorithm.

2.5 Entropy Coding Routine

The last routine in the Image Wavelet Compression algorithm is entropy coding. This process is based on the calculation results from run-length encoding. Using the Huffman encoding algorithm for the entropy coding, the resulting image file can be compressed to 2.33% of the original image size. In the algorithm, two 256 size integer arrays are used to hold the parameters for a fixed Huffman encoding tree. These parameters are expected to work reasonably well with most images. The definition of these parameters is as follows.

```
Int HufSize[256] = { 9, 7, 6, ….}
Int HufVal[256] = {0x00097, 0x00066 …}
```

The Huffman coding call in the C code refers to the following listing2-3.

```
/*****************************************************************
 * Routine: void hufenc(unsigned char ich, int *nb)
 *  *****************************************************************/
void hufenc(unsigned char ich, int *nb)
```

```
{
  int i,nbits,val;
  int bytn,bitn;

  nbits=HufSize[ich];
  val=HufVal[ich];
  for(i=0; i<nbits; i++)
  {
    bytn=(*nb)>>3;
    bitn=(*nb)&7;
    if((val&(1<<i))>0) codep[bytn]|=(1<<bitn);
    (*nb)++;
  }
}
```

Listing 2-3: C code in entropy coding algorithm.

As shown above, *char* is the input pixel value, **nb* is called by reference from the caller. The whole process includes checking values from the Huffman tree parameters, shifting, multiplication, addition, and comparison, to get the final compressed value.

Chapter 3

Wildforce Board and FPGAs

The Image Wavelet Compression application has been executed on a commercial configurable computing platform, the Wildforce board in this research. The board is installed with Xilinx XC4000 series of Field Programmable Gate Arrays (FPGAs) as Processing Elements (PEs) [5]. This chapter explains the architecture of the Wildforce board, FPGA organization, signals used for host, PE memory communication, and the Wildforce board host-control application program interface (API). Host control code is presented to demonstrate the API. JHDL, a hardware description language used in the Janus tool design, is also introduced here.

3.1 Wildforce Board Architecture

The Wildforce Reconfigurable Computing Engine is a member of the Wildforce family developed and provided by Annapolis Micro Systems. Figure 7 shows the block diagram of the Wildforce board. The Wildforce board contains five processing elements (PEs): PE1 through PE4, connected via a 36x36 crossbar switch, which is regulated by the fifth FPGA control element, CPE0. The four processing elements are designed in a linear way in which each is connected to its neighbors to the left and right via 36-bit wide data paths. The

crossbar can be configured by CPE0 to allow broadcast or transfer of data among PEs on a nibble basis. Each PE also has external memory elements that are connected through

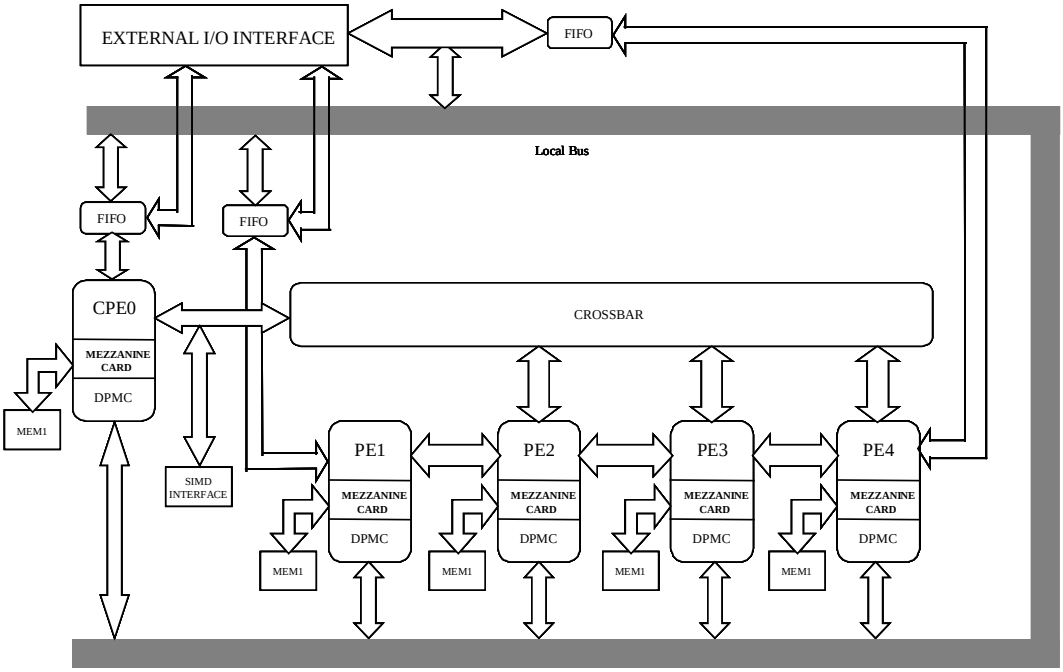


Figure 7. Block diagram of Wildforce board.

mezzanine expansion connectors. These memories can be accessed by host or processing elements arbitrated by the dual port memory controller DPMC which can be controlled by the host. If a PE is not allowed to access memory, the DPMC is in *block* mode, otherwise it is in *arbitrate* mode. The host CPU can exchange data with CPE0, PE1 or PE4 through FIFO 36 bit-wide buffers. It can also fetch and load data with the PEs memory directly through 32 bit-wide bus.

The processing element (PE) on the board consists of a Field Programmable Gate Array (FPGA) and an attached memory. The FPGA includes two major configurable elements:

configurable logic blocks (CLBs) and input/out blocks (IOBs). The developer can use CLB functional elements to construct their logic. CLBs are interconnected by a powerful hierarchy of versatile routing resources, and surrounded by a perimeter of programmable Input/Output Blocks (IOBS). There are two ways that the FPGA can be customized: the FPGA can either actively read its configuration data from an external serial or byte-parallel PROM (master modes), or the configuration data can be written into the FPGA from an external device (slave and peripheral modes) [8]. The XC4000X devices can run at synchronous system clock rates of up to 80 MHz, and internal performance can exceed 150 MHz. In this project, Xilinx XC4062XL FPGAs, each consisting of 2304 CLBs and a fast 2MB static memory, are used on the board. The IWC application executing on the Xilinx4062XL FPGAs operates at 14 MHz.

Figure 8 shows the block diagram among the PE, PE memory and host. PEs use those signals to access memory and communicate with host. PE and PE memory create a block

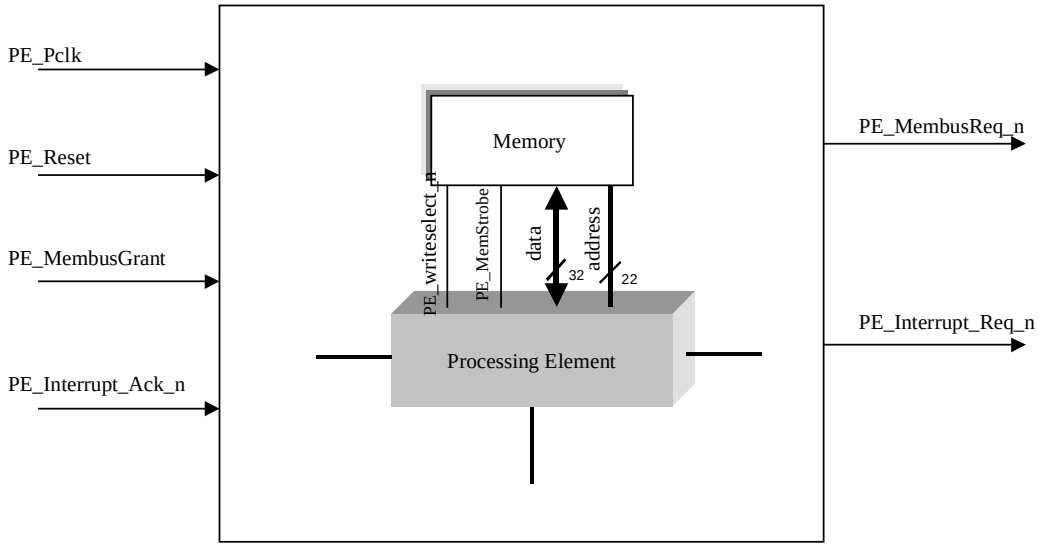


Figure 8. The block diagram for the PE, PE memory and host signal communication.

interface to the outside world (host). Signals having arrows pointing to the block denote the input signals to PE and PE memory interface. Signals having arrows leaving the block denote the output signals from interface to host.

Figure 9 illustrates the three-way handshaking communication between the PE and the host. When the PE wants to send an interrupt signal, it drives **PE_Interrupt_Req** low until the host drives a **PE_Interrupt_Ack_n** signal low which informs PE that its interruption has been acknowledged by the host system. PE can use this three-way handshaking method to inform the host when it has finished its calculation.

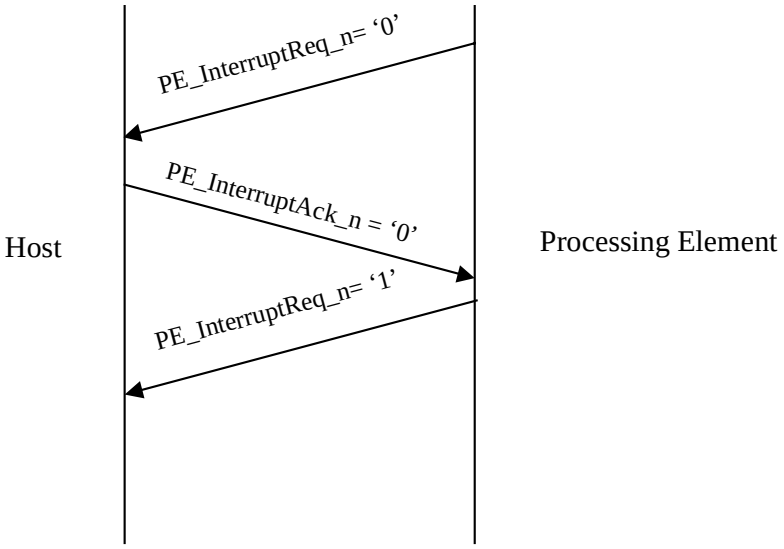


Figure 9. Three-way handshaking communication between PE and Host.

The timing diagram for PE accessing its memory is shown in Figure 10. In Figure 8, each PE communicates with its memory through a 22-bit *address* bus, a 32-bit bi-directional *data* bus, a read-write select control signal **PE_Writeselect_n,** and a strobe signal **PE_MemStrobe**. When the PE wants to access memory, it should drive **PE_MemBusReq_n** signal to a '0'. This

tells the dual port memory controller that it requires access to the local bus. If the PE is not blocked to access the local memory, the dual port memory controller will send a bus grant to PE by driving **Pe_MemBusGant_n** signal to '0'. This usually needs *three* clock cycles after asserting the request. **PE_MemStrobe_n** should be set to '0' when PE is accessing the memory. **PE_MemWriteSel_n** signal '0' denotes a write access and '1' denotes a read access. PE is required to assert the address of the access on the **address** signal. If PE wants to write memory, the **data** to be written must be asserted in the same circle as **address** assertion. If PE wants to read data from memory, the **data** will be available *three* clock cycles after the PE asserts the **address**. The host has absolute access to the memory in blocked mode, whereas in arbitrated mode, the host can access memory only when PE has not been granted access [5].

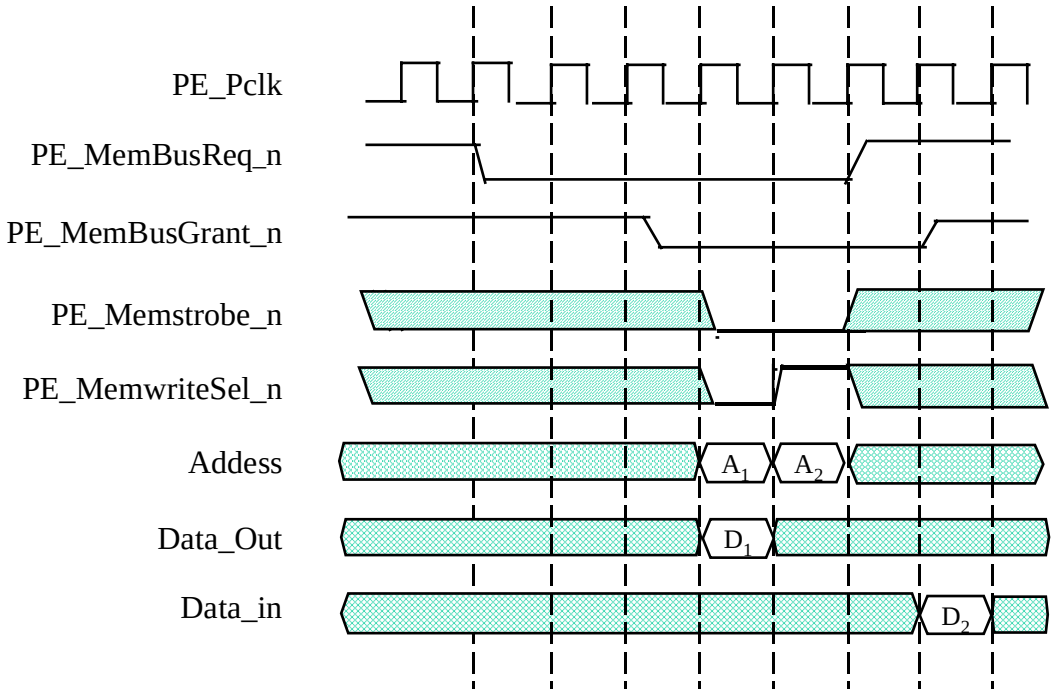


Figure 10. Timing diagram for memory read after a write access.

Xilinx FPGAs can be reprogrammed an unlimited number of times. They can be used in a design where hardware is changed dynamically. The user can take advantage of FPGA's reconfigurable computing property to design his particular algorithms and run application at hardware speed. By using the high-level automated design tool such as Janus [Section 4], the user can execute his application in a run-time environment. Another advantage of Wildforce board architecture is its ability to provide parallel-processing paths compared with the traditional single stream architecture used in Von Neumann Machine.

3.2 Wildforce Host Software Control Application Programming Interface (API)

Annapolis Micro Systems, Inc. has provided the Wildforce board host-control software package. This host-control application programming interface is a set of C function calls. The API provides functions to open and close the Wildforce FPGA board, program the PEs, start and stop the system clock, reset signals, make interrupts, access memory, etc.

Figure 11 shows a simple routine to control a processing element activity. The host CPU loads the configuration data into the PE, loads data to the PE memory, lets the PE do calculations, waits for the PE to finish the calculation, then fetches the data from the memory.

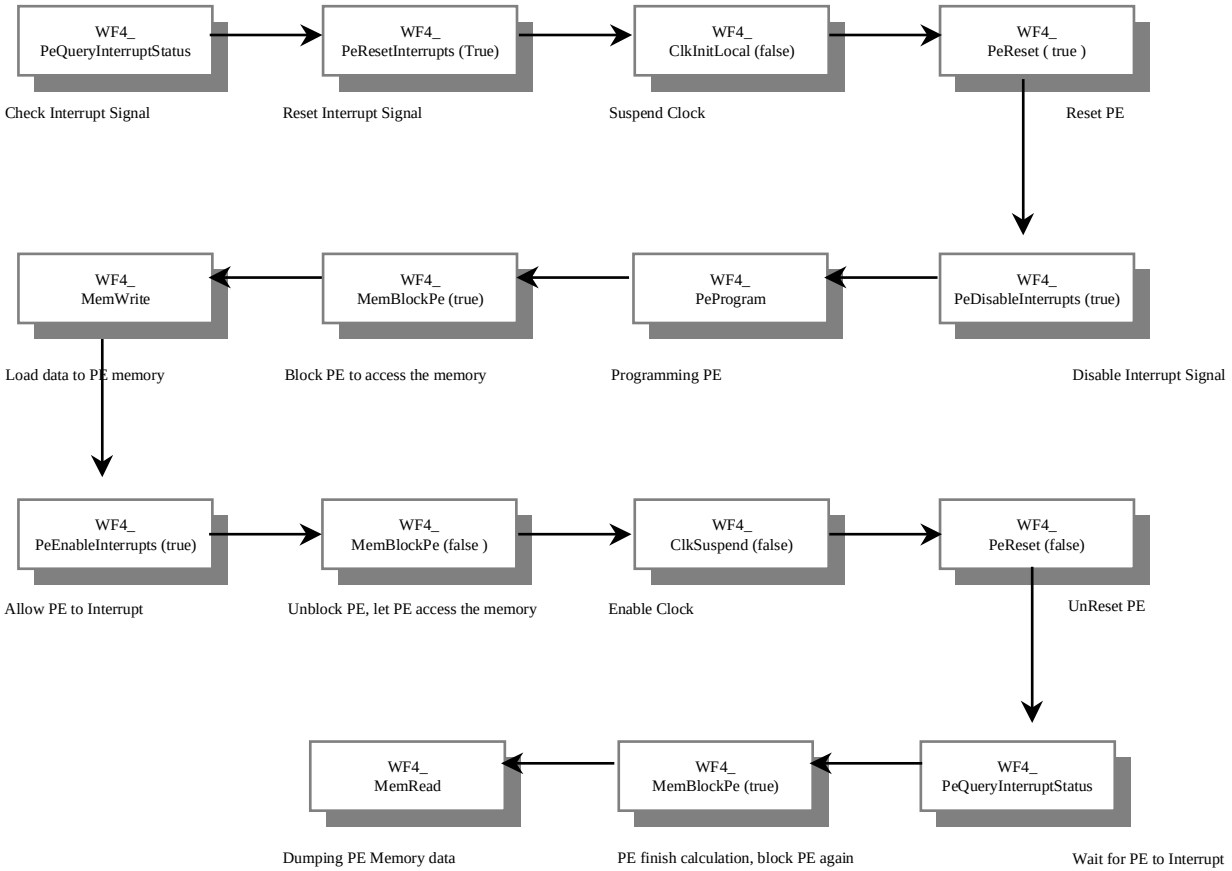


Figure 11. A simple host routine to control processing element Activity.

3.3 Programming the PE Using JHDL

Just Another Description Language (JHDL) is a set of FPGA CAD tools developed at Birgham Young University (BYU) [9]. JHDL is similar to the hardware description language VHSIC Hardware Description Language (VHDL). It allows developers to design the structure of the circuit, provides the environment for simulation before executing a design, supports tools for netlisting, and lets the developer compile the netlist into programming files. Moreover, JHDL has advantages compared to other tools. It is easy to learn and allows for platform-independent design. JHDL is a Java based object-oriented hardware description language that can be used by the developer to express circuit organization. Developers who have experience in object-oriented design (OOD) should be able to grasp the language quickly. Also, since JHDL is based on Java, it can take advantage of Java's distributed standard compilers and library packages that guarantee cross-platform programming and execution. JHDL is naturally suitable for developing applications in reconfigurable systems. JHDL creators take advantage of the OOD construction and destruction mechanisms to simulate the creation and deletion of a circuit object. This approach naturally leads to a unified simulation and execution environment in which the designer can easily switch between software simulation and hardware execution on a CCM with a single application description [6]. JHDL has a software simulation kernel to provide a clock-by-clock simulation of user circuits. When executing in hardware, the same constructors load circuit descriptions from a circuit library to control the execution of the CCM. The destructors remove circuits by replacing existing circuits with "blank" configurations. As discussed in Chapter 1, run-time reconfigurable custom computing depends upon high-level automated design tools that can naturally combine the static circuit description and the control program. The tool should dynamically control reconfiguration and computation each time. JHDL can help with this single

integrated description since it uses Java-based HDL to describe the circuit constructs executed on the CCM. The remaining components, such as the graphical user interface (GUI) and host control code can also be programmed in Java. JHDL requires no language extensions. JHDL is also intended to work with any standard CCM. JHDL supports both partial and global configuration. In this research, the partial configuration property has not been exploited. This approach should be used in future research to reduce reconfiguration time.

Because of these inherent advantages, Janus is designed to let a developer use JHDL to describe the hardware circuit.

Chapter 4

Janus - A High-Level Automated Design Tool

Normally, when an application engineer develops an application on an FPGA-based computing machine, two tasks must be performed. One task is to use a hardware description language to design the circuits for a special functionality machine. The second task is to write a control host program (usually in C/C++) based on the API provided by the hardware board provider (Section 3.2). In some cases, this control code is very simple [Figure 11]. If a complex application such as Image Wavelet Compression (IWC) want to be executed in a run-time reconfiguration environment, it requires a high-level automated design and execution tool. This tool should be able to load a variety of configurations and data, on demand, as the application proceeds. Janus is a tool that allows RTR CCM applications to be developed, simulated, and executed as a consecutive process. Janus allows the developer to use Java to design the circuit module and to describe the structure (stages) of the application to fit into its framework. Janus schedules events such as circuit configuration, fetch data, load data, and control signal management automatically when executing the application on board. Janus also hides the detailed hardware structure from the developer so that the developer can design his or her application

without much knowledge of the target architecture. This is important because it means that CCM design on an FPGA board will not just be the hardware engineer's privilege.

4.1 Janus Structure

The Janus tool is based on the object-oriented language Java. On the host-side, Janus schedules the application stages using *Queue*, *Stack*, and *Hash* table data structures provided by Java packages. The host and Wildforce board communication API calls are coded into a dynamic link library (DLL). This dynamic linking provides a mechanism for connection Janus applications to the API calls at run time when an application is loaded and executed on the board. DLL code will be modified only when the hardware API changes. Janus takes advantage of JHDL properties to provide seamless software simulation and hardware execution switching without changing the application's architecture. Moreover, the user can choose to run the application on several PEs to calculate at the same time. An GUI, provided by Janus, can be used to control and monitor the software simulation and hardware board execution activity. It also allows the user to choose the run pattern and execution parameters by simply pushing buttons and entering values. A detailed Janus package structure is illustrated in Figure 12 (modified from [1]). A brief introduction to this structure and the package functionality is beneficial for the developer to understand how to use the Janus tool.

The whole Janus tool consists of six main packages, which are presented in the remainder of this section. The user's applications are dependent upon these packages.

`hardware` package: The **`hardware`** package provides an abstraction of hardware architectures, processing elements and memory.

`runtime` package: The **`runtime`** package provides data structures for describing the sequence of run-time events and interfaces to the hardware architecture for executing them.

`schedulers` package: The **`schedulers`** package contains the compile-time code for translating the user's description of an application into a sequence of run-time events.

`base` package: The **`base`** package contains the application's framework that the developer extends to create his or her own application. The **`base`** package includes an abstract

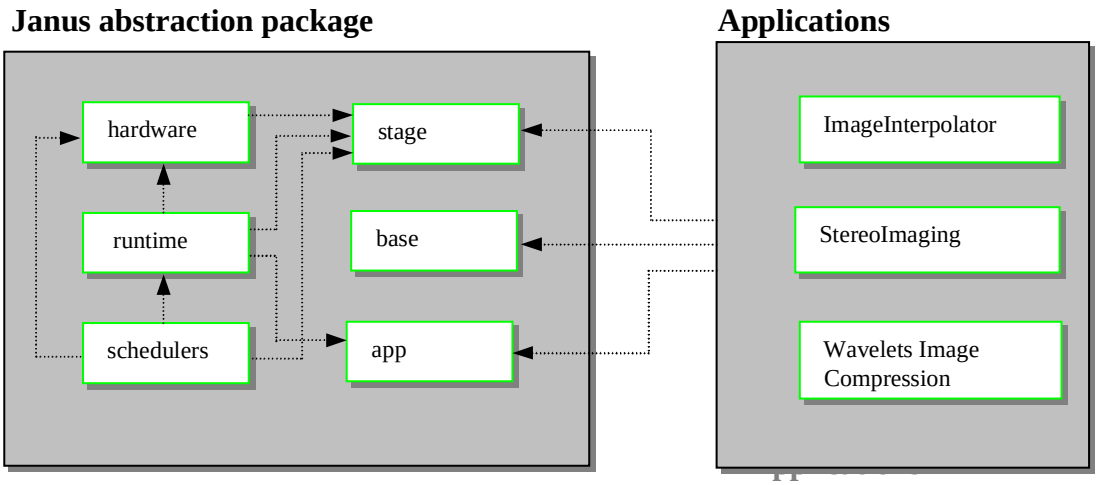


Figure 12. Janus package abstraction diagram and relation to applications

interface **`Application`**. All Janus applications must implement this interface and the developer should begin coding from there. Listing 4-1 shows the application interface.

```
public interface Application {
      public void init(String args[]);
      public int getLogicRequirement();
      public int getMemorySizeRequirement();
      public int[] getValidMemoryWidthRequirement();
      public StageOrdered build();
}
```

Listing 4-1: The **`Application`** interface.

The `init(String args[])` method is called to allow the application to parse command line arguments. It allows the application object to initialize its internal state and to perform any other tasks that it might require.

The `get` functions are used to determine whether a particular target platform is capable of executing the application.

The `build()` method actually creates the ordered set of stages required by the application. The IWC's `build()`method implementation is shown in Listing 5-1.

`stages` package: The **`stage`** package defines the interfaces that the application developer can use to group together the computational operations to model the application. Section 4.2 has the detailed description of this package.

`app` package: The app package responsible for the creating the Janus user interface.

(The above descriptions are based on [1])

The **hardware**, **runtime**, and **schedulers** packages are hidden from the application developer who uses Janus tool to design an application. In other words, applications are not directly dependent upon these three packages and developers do not need to know much about these packages. Only when new architectures are developed and new FPGA technologies are produced will the backend developer need to change these packages. This is one of the advantages of Janus; it allows the application developer that has no detailed knowledge of a particular RTR CCM architecture, to design and run applications.

The **base**, **stages**, and **app** packages provide the interfaces that a user should use to develop his or her application. Chapter 5 presents a detailed discussion on how to use these packages to develop the Image Wavelet Compression application.

4.2 Stage Package Class Structure

The **stage** package is discussed in detail here because the developer must directly use and implement this package's interfaces to design his or her application. Figure 13 shows the class diagram of the **stage** packages. The developer can implement the **Simulatable** interface with the **stage** package to do software platform debugging and simulation.

The **stage** package has a **Stage** interface class. **StageSoftware, StageOrdered,** and **StageUnordered** are three classes implement the **Stage** interface. These three classes, and another interface called **Operation**, form the basic building blocks that an application developer uses to model computations. The developer describes the computation by creating a hierarchy of computational stages [1]. The **StageOrdered** object maintains an ordered list of ordered dependent stages while the **StageUnordered** object collects unordered independent stages in each ordered stage. The **StageSoftware** class is used to create software implementations of computational stages. The **Operation** interface has a software component and a hardware component. On the software-side, **Operation** provides functions

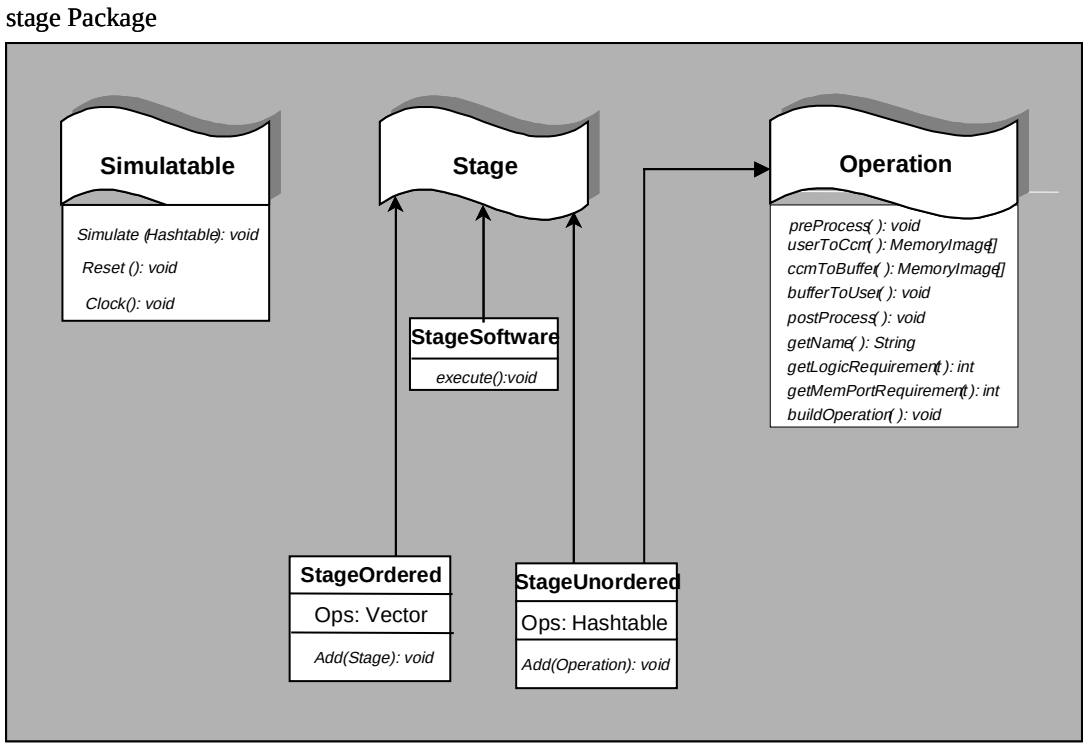


Figure 13. Class diagram of the stage package.

such as `preProcess()`, `userToCCM()`, `ccmToBuffer()`, `bufferToUser()`, and `postProcess()` to describe the data transformation between the host and the processing elements on the RTR CCM. The developer can use `getLogicRequirement()` and `getMemPortRequirement()` methods to get the operation hardware resource requirements. This information is used by the **`scheduler`** to assign operations in processing element. On the hardware side, the developer can implement `buildOperation()` virtual function to design hardware structures. These models can be netlisted, placed-and-routed on the target FPGA.

Chapter 5

Image Wavelet Compression

Implementation on FPGAs

The Janus tool provides an environment that developers can use to design, simulate and execute their RTR CCM applications. This chapter describes the Janus application design process, Janus hardware circuits design model, the simulation environment, the synthesis tools used to map the designs to XILINX FPGA devices, and the debugging facilities available to the developer. The IWC implementation is used as an example to explain this process.

5.1 Janus Application Design Process

The Janus tool provides a complete programming, simulation, and execution environment for an application. Figure 14 illustrates the basic design steps used in developing a typical Janus application.

The first step in developing a Janus application is to understand the problem definition. Sometimes it is much easier to model the task with behavioral descriptions, such as using C programming language, to implement the operation first. For example, the application presented

here is to implement the wavelet based image compression algorithm with a 512x512 8-bit pixel image in PGM format. The algorithm is programmed in C code first to test its operation. Moreover, the program is coded by using only integer adds and shifts to make it easier to implement when using hardware.

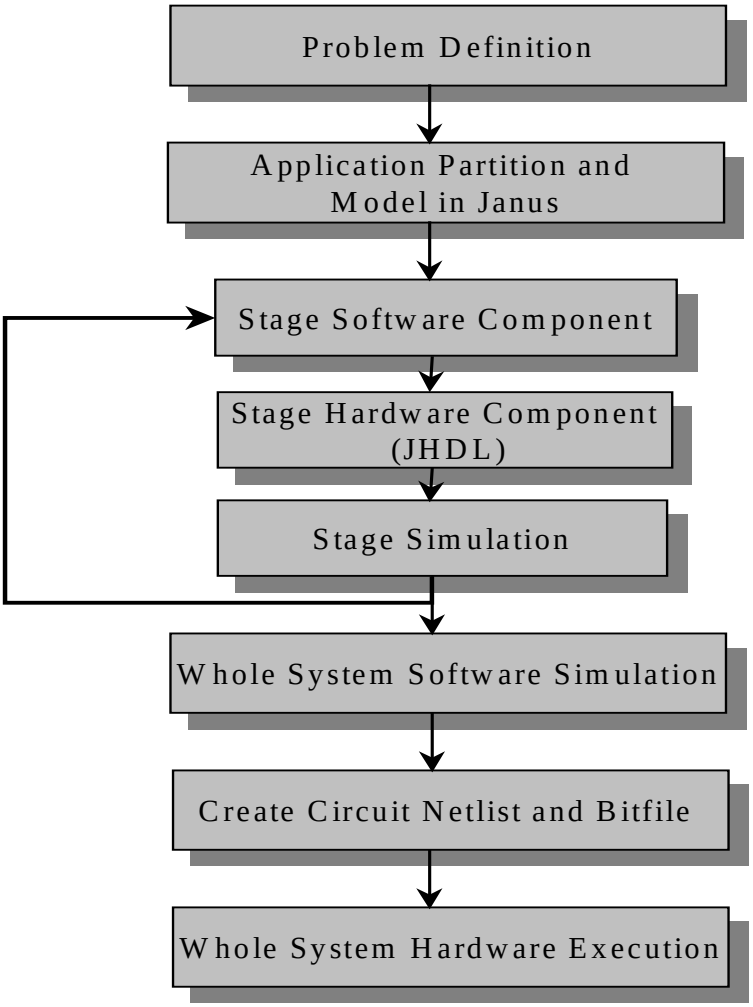


Figure 14. Janus application design process.

The next step is to partition the application and model it with the Janus interface. Some factors should be considered when partitioning the application and mapping it into the FPGA board. One factor is *time*, which determines how much computation could be designed in per

clock cycle. This factor decides the circuit delay, which then affects the system's maximum frequency. Often, the designer does not know the exact clock frequency that the circuit can endure until run on the hardware. For example, the highest frequency that the IWC implementation operates is 14 MHz. Another factor that should be considered is *area*, which relates to how many PE resources must be used in each reconfiguration. For example, Xilinx XC4062XL FPGA consists of 2304 CLBs. Each time that the PE is reconfigured, the designer should make sure his design does not exceed this amount. Table 1 in Chapter 6 demonstrates the number of CLBs utilized each time during reconfiguration in the IWC implementation. The third factor relates to the Janus tool. The partition should be based on the application's natural computation operation and utilize the Janus **`stage`** package. The whole application should be divided into interdependent ordered stages (**`StageOrdered`** in **`stage`** package), where each stage has some independent unordered stages (**`StageUnordered`** in **`stage`** package). Figure 15 illustrates this model. The ordered stages are inter-dependent, while the unordered stages in each ordered stage should all be computationally independent of each other. Section 5.2 provides a detailed description on how to create the stages in the IWC implementation. The Java class `ImageCompression` in Figure 16 and Listing 5-1 is the corresponding class and code created for Image Wavelet Compression application partition.

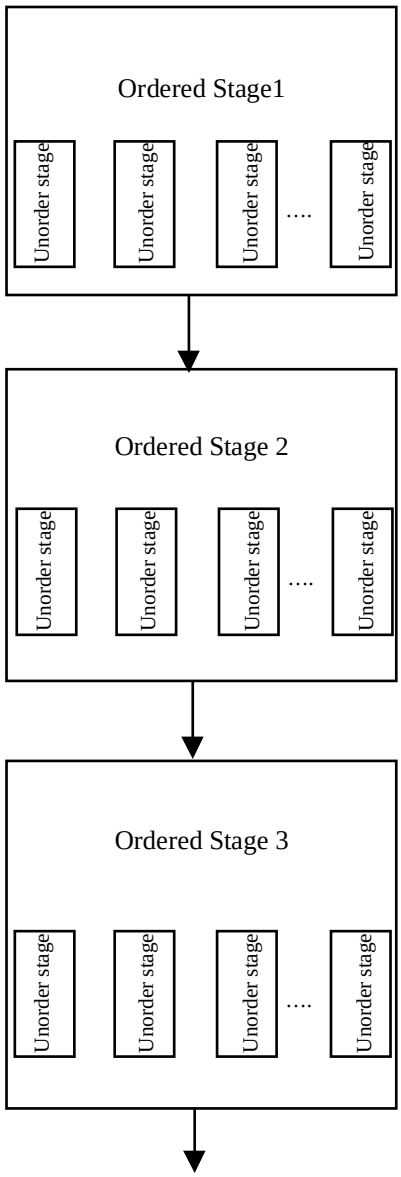


Figure 15. Janus application partition model. Ordered stage(n) depending on the result of ordered stage(n-1). Each ordered stage has some unordered stages that computationally independent of each other. Therefore, parallel processing in the FPGAs can be used to process these unordered stages.

Steps three, four, and five in Figure 14 are for each stage design process. The arrow in the reverse direction represents the iterations needed in the design process to finish all ordered stages designs. The purpose of the host-side software component on each stage is to prepare the input data and to hold the intermediate results. The stage hardware component denotes the hardware side circuit design for each stage, which is specified in JHDL. Sometimes, the unordered stages execute the same operation; hence, they have the same hardware component design. In the IWC implementation of the wavelet transform routine, Java classes `WTRowOp` and `WTColOp` are host-side software component, and `DWT` and `WtMemInterface` are the hardware component files [Figure 16]. Each stage is verified in both software and hardware before integrating into the whole system.

After all stages have been designed and verified, the whole system can be combined and simulated using Janus tool. The six step in Janus application design is to create netlists and bitfiles for all stages. The final step is to execute the whole system using hardware. This step is simple when using the Janus tool. Developers simply put bitfiles in the application directory, link Janus with the dynamic link library (DLL) for host and board communication, and use the Janus GUI to run the application on the target configurable computing board. The GUI will be introduced in Chapter 6.

5.2 Classes Structure of the IWC Implementation in Janus

Figure 16 illustrates the class structure of Wavelet Image Compression implementation in Janus.

As stated before, `ImageCompression` class deals with the application partition. It implements the **`base`** package's **`Application`** abstract interface and uses the **`stage`**

package's **StageSoftware, StageOrdered,** and **StageUnordered** classes. The developer should begin coding from here [Section 4.1]. The `init()` method implemented in the `ImageCompppression` class reads the image pixels into a large array. In the `build()` method implementation, the instances of `WTOp`, `QuantOp, RleOp` and `EntOp` classes are created and connected into a hierarchy of computational ordered and unordered stages [Listing 5-1]. `StageSetup, StageDone, AppSetup,` and `AppDone` are instances of **StageSoftware** objects. These are software stages whose operations are executed on the host instead of on the CCM. For example, `StageSetup` and `StageDone` take care of updating the GUI; `AppDone` can be used to collect and form the final computation result and output it to a file.

`ImageCompressionOp` class implements both **Operation** and **Simulatable** interfaces, which belong to the **stage** package [Section 4.2]. The **Simulatable** interface handles the software simulation and includes function calls such as: `simulate(), reset(), clock()`. The **Simulatable** interface is implemented in `ImageCompressionOp` class. The developer can debug and simulate his design here. The **Operation** interface is responsible for generating each operation's I/O behavior on a standard memory interface specification defined by the Janus tool framework. It also takes care of the hardware side circuit design [Section 4.2]. The **Operation** interface is not implemented in the `ImageCompressionOp` class but in its inherited classes since different operations in the IWC implementation have different hardware design and I/O operations.

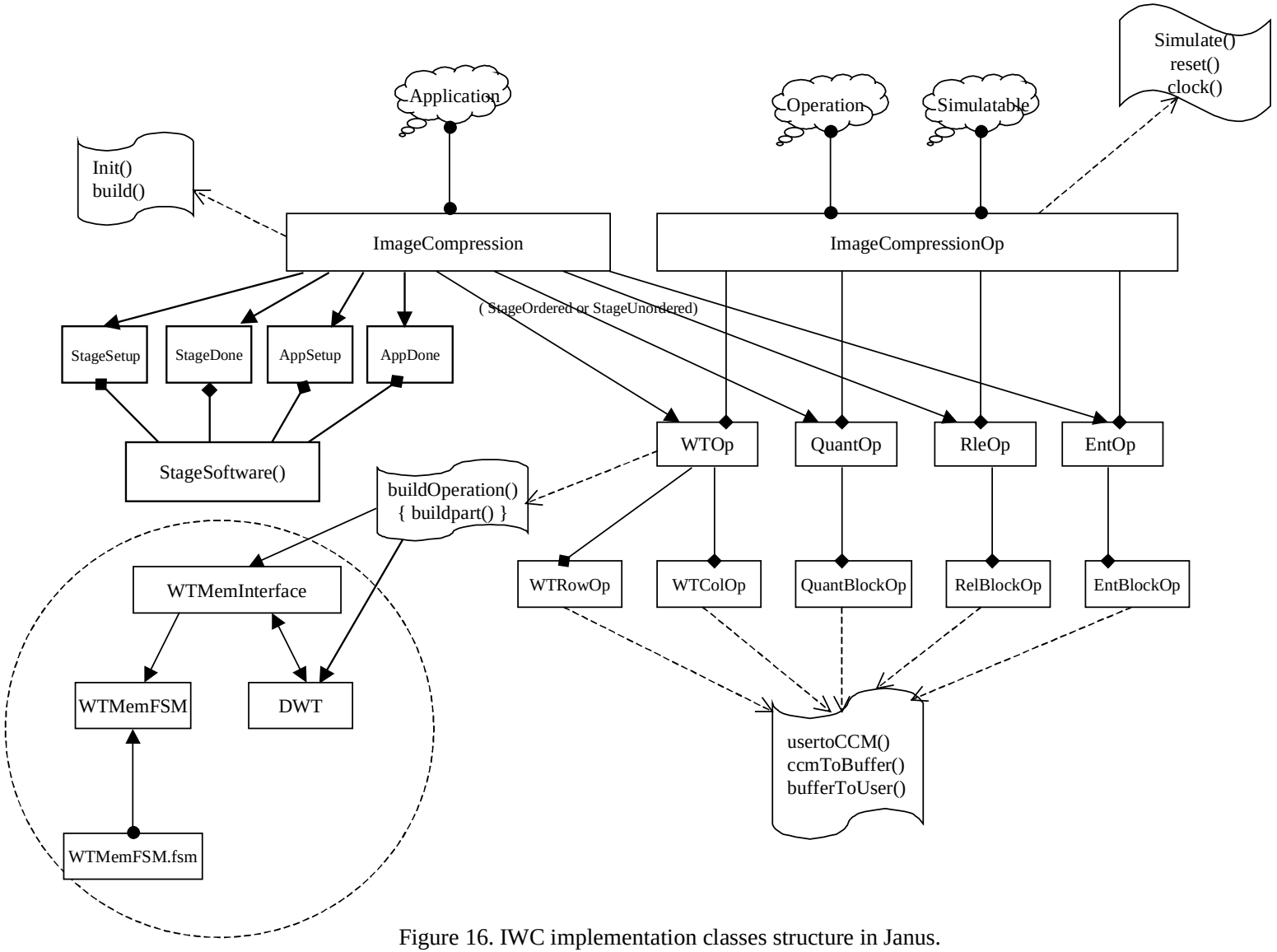


Figure 16. IWC implementation classes structure in Janus.

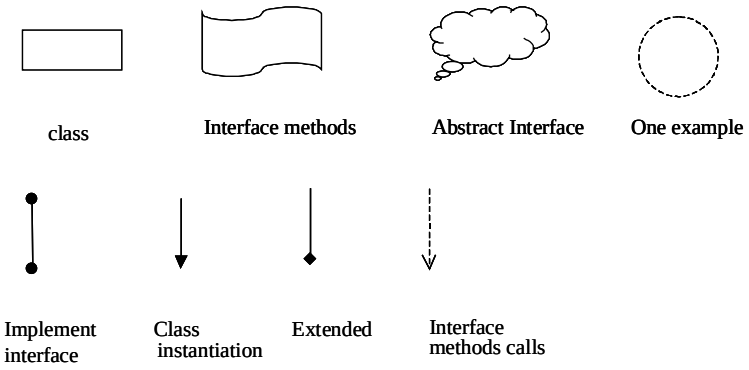


Figure 16 continuous here.

The `WTOp`, `QuantOp`, `RleOp`, and `EntOp` classes are extended from the `ImageCompressionOp` class; the main responsibility of these classes is to implement the **`Operation`** interface's `buildOperation()` method for the hardware side of circuit design. For example, The `WTOp` class creates the `WtMemInterface` object and `DWT` object, `WtMemInterface` calls the `WtMemFSM` object, which uses the `WTMemFSM.fsm` file to create a truth table or transition table to control the circuit activity. `DWT` is a JHDL file used to create the wavelet transform circuit. Section 5.5 explains this in more detail.

`WTRowOp` and `WTColOp` inherit from the `WTOp` class. `QuantBlockQp`, `QleBlockOp` and `EntBlockOp` inherit from the `QuantOp`, `RleOp` and `EntOp` classes. These classes actually implement the **`Operation`** interface software component part that transfers data between host memory and PE memory.

The dashed circle in Figure 16 indicates the wavelet transform routine's hardware circuit design. The other three routines are not shown due to space limitations. While they have the

same concept and structure as the wavelet transform routine, they have different hardware designs.

5.3 IWC Implementation Application Partition and Model in Janus

As discussed in Section 5.1, the application partition is important in the design process. If the developer uses the Janus tool to design the application, the partition should be based on the application's natural computation operations. It should also fit into the Janus **`Stage`** interface.

Chapter 2 presented the four IWC implementation routines: wavelet transform, quantization, run length encoding, and entropy coding. These routines naturally fit into the Janus ordered stages since they are inter-dependent. After careful study of the wavelet transform routine computation, it is observed that the routine consists of three ordered levels. In each level, rows are processed before the columns calculation [Section 2.2.2]. Hence, the IWC implementation is composed of six dependent wavelet transform stages, followed by quantization, run-length encoding and entropy coding stages, resulting in a total of nine dependent stages. Each stage uses the previous stage's output data as its inputs except the first stage, which use the input from the original image. Figure 17 illustrates the computational structure of the Wavelet Image Compression. In this diagram, nine stages are listed on the right side of the figure to demonstrate the stage order that should be scheduled from the top to the down. They are: `wt512Rows`, `wt512Cols`, `wt256Rows`, `wt256Cols`, `wt125Rows`, `wt125Cols`, `blockQuant`, `blockRle`, `blockEnt` corresponding to the algorithms' nine computation operations: level-one wavelet transform Rows, level-one wavelet transform Columns, level-two wavelet transform Rows, level-two wavelet transform Columns, level-three wavelet transform Rows, level-three wavelet transform Columns, quantization, run-length encoding, entropy coding.

For each stage, there exist unordered stages. For example, in `wt512Rows` stage, 512 rows should be processed using the wavelet transform algorithm in an arbitrary order. In the `blockQuant` [Section 2.3] stage, there are seven blocks that all have the same computation, and can be processed independently. In the `BlockRle` stage, there are seven blocks [Section

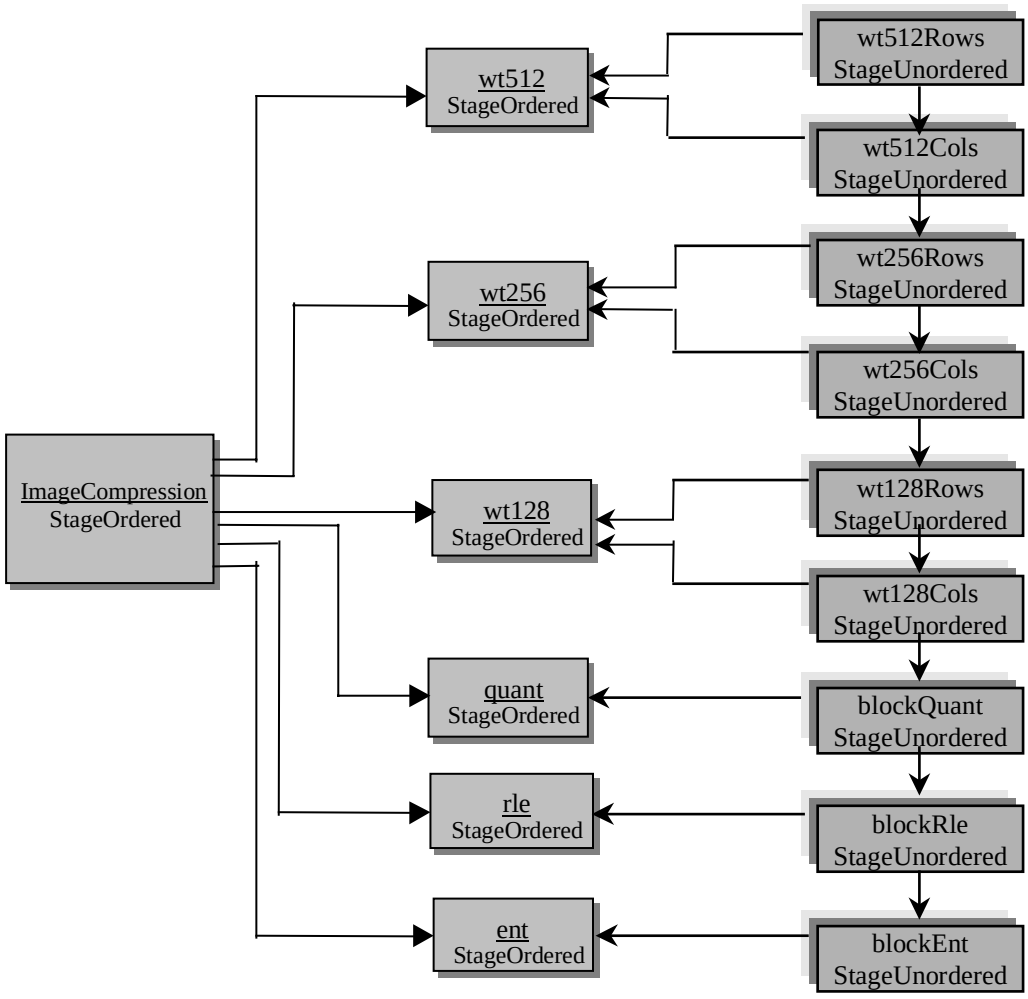


Figure 17. Computation stages for Image Wavelet Compression.

2.4], which can also be processed independently. Those unordered stages could be processed in parallel by several PEs on the target configurable computing board. In the `blockEnt` [Section 2.5] stage, only one block will need computation.

As indicated in Section 5.2, the `build()`method in class `ImageCompression` is where the application's computation stages are created. Listing 5-1 shows the code:

```
public StageOrdered build() throws JanusException {
            return this.build_one();
      }

public StageOrdered build_one() {
  // create all the stages
  StageOrdered compression = new StageOrdered("Image Compression");

  StageOrdered wt512 = new StageOrdered("WT512");
  StageOrdered wt256 = new StageOrdered("WT256");
  StageOrdered wt128 = new StageOrdered("WT128");
  StageOrdered quant = new StageOrdered("Quantization");
  StageOrdered rle   = new StageOrdered("Rle_encoding");
  StageOrdered ent   = new StageOrdered("entropy_encoding");

  StageUnordered wt512Rows = new StageUnordered("WT512 Rows");
  StageUnordered wt512Cols = new StageUnordered("WT512 Columns");
  StageUnordered wt256Rows = new StageUnordered("WT256 Rows");
  StageUnordered wt256Cols = new StageUnordered("WT256 Columns");
  StageUnordered wt128Rows = new StageUnordered("WT128 Rows");
  StageUnordered wt128Cols = new StageUnordered("WT128 Columns");
  StageUnordered blockQuant= new StageUnordered("blockQuant");
  StageUnordered blockRle  = new StageUnordered("blockRle");
  StageUnordered blockEnt  = new StageUnordered("blockEnt");

  // wt512 rows
  for(int i=0; i<inputHeight; i++) {
      WTRowOp row = new WTRowOp(this,"WT512 Row Op", i, inputWidth,
                                inputHeight, InputID, WtID);
      wt512Rows.Add(row);
```

```
}
// wt512 cols
for(int i=0; i<inputWidth; i++) {
    WTColOp col = new WTColOp(this,"WT512 Column Op", i, inputWidth,
                              inputHeight, WtID, WtID);
    wt512Cols.Add(col);
}
// wt256 rows
for(int i=0; i<inputHeight/2; i++) {
    WTRowOp row = new WTRowOp(this,"WT256 Row Op", i, inputWidth/2,
                              inputHeight/2, WtID, WtID);
    wt256Rows.Add(row);
}
// wt256 cols
for(int i=0; i<inputWidth/2; i++) {
    WTColOp col = new WTColOp(this,"WT256 Column Op",i,inputWidth/2,
                              inputHeight/2, WtID, WtID);
    wt256Cols.Add(col);
}
// wt128 rows
for(int i=0; i<inputHeight/4; i++) {
    WTRowOp row = new WTRowOp(this,"WT128 Row Op", i, inputWidth/4,
                              inputHeight/4, WtID, WtID);
    wt128Rows.Add(row);
}
// wt128 cols
for(int i=0; i<inputWidth/4; i++) {
    WTColOp col = new WTColOp(this,"WT128 Column Op", i,inputWidth/4,
                              inputHeight/4, WtID, WtID);
    wt128Cols.Add(col);
}
for(int i=0; i < 7;  i++){
    if(i<=3){
          QuantBlockOp bk_quant = new QuantBlockOp(this,
                            "block_quantization",i,inputWidth/8,
                            inputHeight/8, WtID, QuantID);
          blockQuant.Add(bk_quant);
    }
    else if (i>=4 && i <=6){
          QuantBlockOp bk_quant = new QuantBlockOp(this,
```

```
                        "block_quantization",i, inputWidth/4,
                        inputHeight/4, WtID, QuantID);
            blockQuant.Add(bk_quant);
    }
    else{
            QuantBlockOp bk_quant = new QuantBlockOp(this,
                        "block_quantization",      i,      inputWidth/2,
                        inputHeight/2, WtID, QuantID);
            blockQuant.Add(bk_quant);
    }
}
for(int i=0; i < 7;  i++){
    if(i<=3){
            RleBlockOp bk_rle = new RleBlockOp(this, "block_rle", i,
            inputWidth/8, inputHeight/8, QuantID, RleID);
            blockRle.Add(bk_rle);
    }
    else if (i>=4 && i <=6){
            RleBlockOp bk_rle = new RleBlockOp(this, "block_rle", i,
            inputWidth/4, inputHeight/4, QuantID, RleID);
            blockRle.Add(bk_rle);
    }
    else{
            RleBlockOp bk_rle = new RleBlockOp(this, "block_rle", i,
            inputWidth/2, inputHeight/2, QuantID, RleID);
            blockRle.Add(bk_rle);
    }
}

EntBlockOp bk_ent=new EntBlockOp(this,"block_entropy",RleID, OutputID);
blockEnt.Add(bk_ent);

/* hook stages together
wt512.Add(new StageSetup("wt512 Rows",inputHeight));
wt512.Add(wt512Rows);
wt512.Add(wt512Cols);
wt256.Add(wt256Rows);
wt256.Add(wt256Cols);
wt128.Add(wt128Rows);
wt128.Add(wt128Cols);
```

```
  wt128.Add(new StageDone("wt128 Columns", inputWidth/4, inputHeight/4,
                          WtID, debug));

  quant.Add(new StageSetup("Block Quantization ", inputHeight));
  quant.Add(blockQuant)
  quant.Add(new StageDone("Block Quantization", inputWidth,inputHeight,
                          QuantID, debug));

  rle.Add(new StageSetup("Block Rle_encode", inputHeight));
  rle.Add(blockRle);
  rle.Add(new StageDone("Block Rle_encode", inputWidth, inputHeight,
                       RleID, debug));

  ent.Add(new StageSetup("entropy_encode", inputHeight));
  ent.Add(blockEnt);
  ent.Add(new StageDone("entropy_encode", inputWidth, inputHeight,
                        OutputID, debug));

  compression.Add(new AppSetup(6));
  compression.Add(wt512);
  compression.Add(wt256);
  compression.Add(wt128);
  compression.Add(quant);
  compression.Add(rle);
  compression.Add(ent);
  compression.Add(new AppDone(imageFileName));

  return compression;
 }
```

Listing 5-1. The Image Wavelet Compression's `build()` Method in `ImageCompression` class.

The `build()` method in Listing 5-1 begins by creating six ordered stages: `wt512, wt256, wt128, quant, rle,` and `ent,` and nine unordered stages, which are shown on right side of Figure 17. Then, the `build()` method fills each unordered stage with the operations that should be performed. For example, the first unordered stage `wt512Rows` is filled with `WTRowOp` operation that will correspond to the wavelet transform algorithm on 512 rows of an image. This

is implemented using looping and "new" operations. After all of the unordered stages are filled with operations, it then proceeds to order these unordered stages into the six ordered stage objects. The order that objects are added is important. This is quite different from the unordered stages, where objects can be added in arbitrary order. For example, `wt256Rows` should be added prior to the `wt256Cols`, and `Quant` should be added before the rle object. The `StageSetup`, `StageDone`, `AppSetup`, and `AppDone` objects are added to perform operations that are executed on the host, such as updates to the GUI, and stage memory reordering tasks.

5.4 Stage Software Component Design

During a typical stage's operation cycle, an operator is configured onto the target hardware architecture's processing element and is connected to a memory interface [1]. The host sends (fetches) the input (result) data to (from) the memory. The operator is given signals to start and end its data processing. The Janus framework defines a standard memory interface for its own I/O behavior to map the FPGA memory. The developer should understand this interface before designing the software component.

Figure 18 shows this interface used in the wavelet transform routine's software component design. Appendix 5 contains the `WTRowOp` class showing the wavelet transform routine host-side software component code. In the code, `pre_pixels` and `post_pixels` are arrays that hold the input and output pixel values. Pixels are stored row by row. `pre_mem` and `post_mem` are `MemoryImage` type arrays used in the actual memory copy process. The `UserToCcm()` function call returns `pre_mem` that represents the transfer of data from host to PE memory. The `ccmToBuffer()` function call returns `post_mem` that denotes fetch data from PE memory to host; `bufferToUser()` function call is used to reorder the data and to put it into `post_pixels` array.

`MemoryImage` object should be used in `pre_mem` and `post_mem` since the operation's communication behavior must be defined without the knowledge of the underlying hardware platform. The constructor for the `MemoryImage` object takes its first parameter as a reference to an integer array holding the data to be copied to the memory. The second and third parameters represent the index of this array to begin copying and the number of array elements to copy. The fourth parameter is the destination address on the processing array's memory to begin copying [1]. `Config` is an integer array that holds the hardware control signals. In this example, it holds the `begin read`, `begin write`, `end read`, and `end write` memory addresses. These addresses value will be put into the PE's memory `address[0]` to `address[3]`.

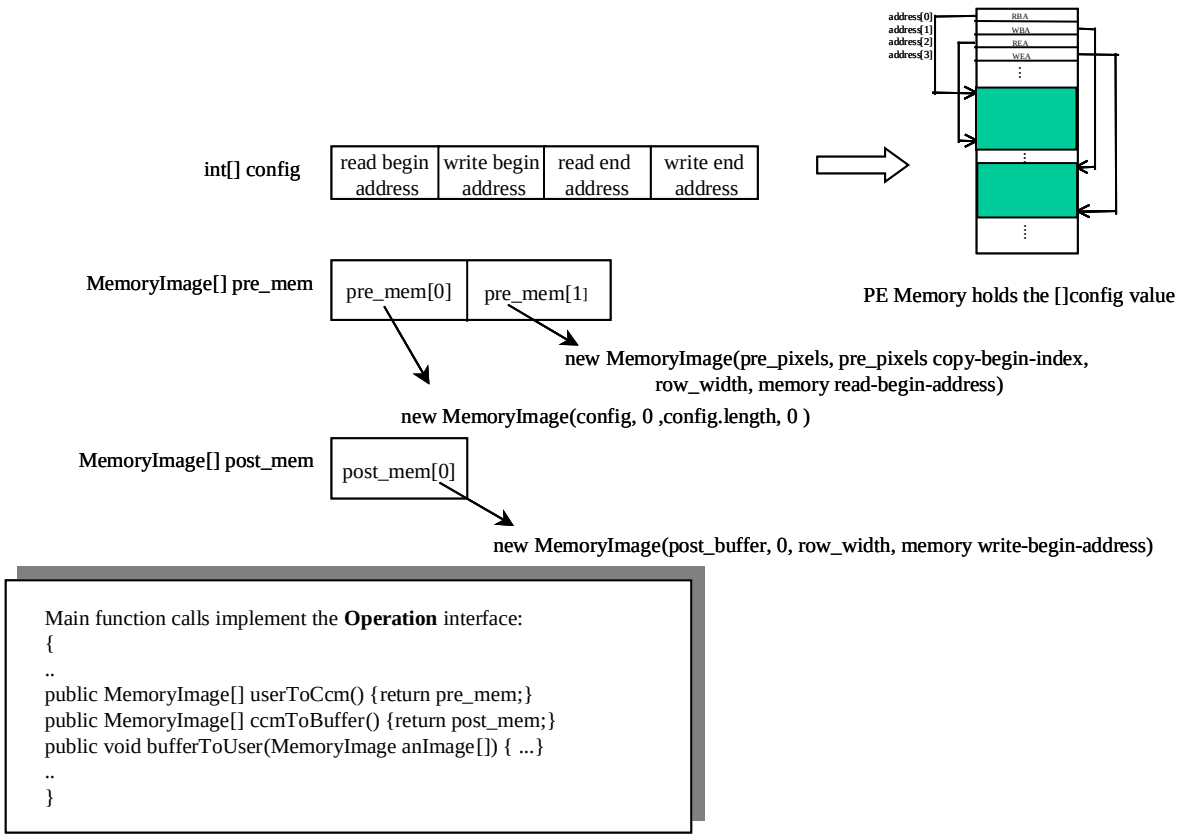


Figure 18. Janus I/O interface in software component design.

5.5 JHDL Hardware Component Design in IWC Implementation

As discussed in Section 4.2, the developer can implement the `buildOperation()` virtual function in the **`Operation`** interface to design a hardware component. `BuildOperation()` uses the Janus Memory-PE Interface to map into the FPGA hardware. Two components are included when the developer designs the hardware in this function call. One component creates the description of the combination logic circuit where the developer can use the JHDL logic API by instantiating gates and wires objects. The other component uses the JHDL Finite Stage

Machine (FSM) to provide a description of stage machine in a transition table format. This chapter explains the Janus Memory-PE interface and the design model. Detailed examples in IWC implementation are shown to demonstrate how to design the hardware component in Janus.

5.5.1 Janus Memory-PE Interface

Janus Memory-PE interface should be introduced before discussing the Janus combination logic and the FSM design. In Section 3.1, the PE, PE memory and the host communication model [Figure 8] has been explained. Based on this architecture, Janus creates a simplified interface. The detail of the host communication with the PE is hidden from the Janus developer. By doing this, the developer can concentrate on the hardware circuit and the PE-Memory communication design. Figure 19 demonstrates the block diagram of Janus Memory-PE interface. The **enable** signal is an input to the Janus Memory-PE Interface. When the **enable** is high, the PE can compute and communicate with PE memory. The **Done** signal is issued by the PE-Memory interface. It acts like an **interrupt** signal issued to the outside world when the PE finishes its calculation. **Write_sel** and **strobe** are issued by Janus PE-Memory interface as output if PE wants to read or write data to memory. In hardware, **PE_MemData_InReg** is bidirectional, but in the Janus PE-Memory interface, the **data_in**, and **data_out** bus are separated to simplify the design.

Listing 5-2 is the PE-Memory interface design in the IWC implementation. All stage hardware components use the same interface. The `buildOperation()` virtual function is first called in `ImageCompressionOp` class, which then calls `buildpart()` virtual function. Because `WTOp`, `QuantOp`, `EleOp` and `EntOp` classes inherit from `ImageCompressionOp` class, they inherit the `buildpart()` function and implement it

depending on the computation [Figure 16]. Notice that the parameters in the function call `buildOperation()` and `buildpart()` are interface signals exactly as in Figure 19. The detailed WtMemInterface code refers to Appendix 2.

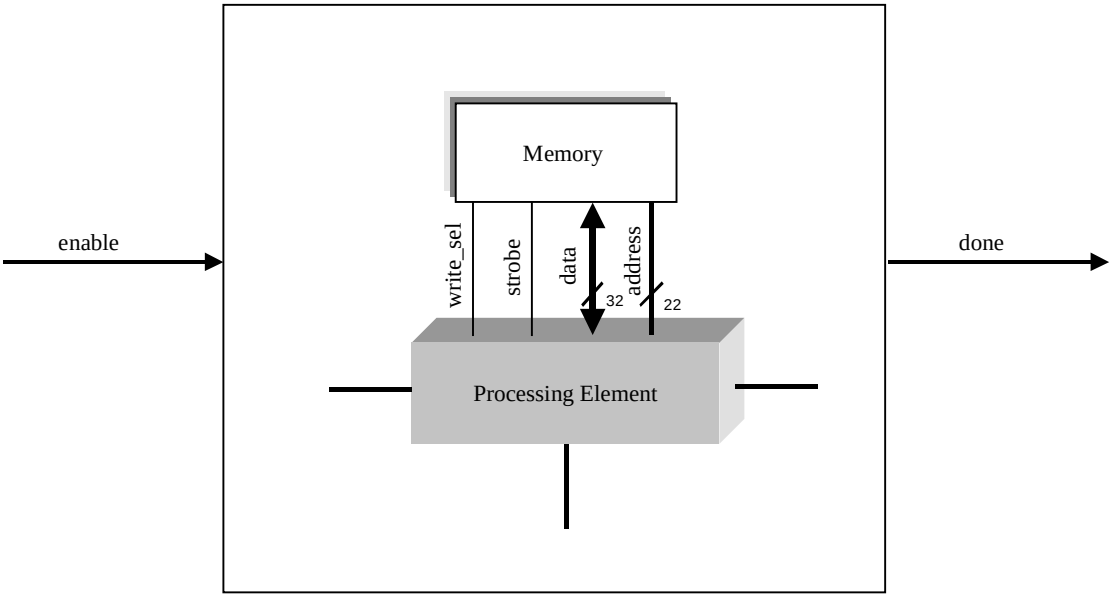


Figure 19. Memory-PE interface block diagram in Janus.

(function buildOperation () call in ImageCompressOp.java):

```
public void buildOperation( Node parent,Wire data_in,Wire data_out,Wire
      addr,Wire write_sel,Wire strobe,Wire enable,Wire done) throws
      JanusException {
```

```
      this.buildpart(parent, data_in, data_out, addr, write_sel, strobe,
enable, done);
}
```

(function buildpart () call in WtOp.java):

```
protected void buildpart(Node parent,Wire data_in,Wire data_out,Wire addr,Wire
     write_sel,Wire strobe,Wire enable,Wire done) throws JanusException {
      Wire done_read_n = Logic.wire((Cell)parent,"done_read_n");
      WtMemInterface                mi                =                new
      WtMemInterface(parent,data_in,addr,write_sel,strobe,enable,done_read_n,d
      one);
                  Wire read_en   = mi.getReadEnable();
                  Wire writeD_en = mi.getWriteDEnable();
                  Wire writeS_en = mi.getWriteSEnable();
                  Wire writeS0_en= mi.getWriteS0Enable();
                  Wire writeLD_en= mi.getWriteLDEnable();
                  Wire writeLS_en= mi.getWriteLSEnable();
      this.buildWT(parent, data_in, read_en, writeD_en, writeS_en, writeS0_en,
      writeLD_en, writeLS_en, data_out);
            }
      protected void buildWT(Node parent, Wire data_in, Wire read_en, Wire
      writeD_en, Wire writeS_en, Wire writeS0_en, Wire writeLD_en, Wire
      writeLS_en,Wire data_out) {
            DWT wt = new DWT(parent, data_in, read_en, writeD_en, writeS_en,
            writeS0_en,  writeLD_en, writeLS_en, data_out);
            }
}
```

Listing 5-2. Janus PE-memory interface implemented in IWC wavelet transform routine.

5.5.2 Janus Combination Logic Circuit and Memory FSM Design Model

In the `buildOperation()` function call, two components of code are combined to form the hardware circuit design. One part is the JHDL combination logic circuit design; the other is the Janus Memory FSM file, which is the transition table file for the state machine. Figure 20

shows the Janus Combination Logic Circuit and Memory FSM Design Model. The purple dashed rectangle represents the Janus Memory-PE interface. Arrows pointing into the rectangle are input signals to the interface, while arrows that are pointing out of the interface are signals generated by interface. **Data_in** holds the data read from the memory. **Data_out** holds the data writing to the memory. The **enable** signal enables the circuit activity. The **Address** signal is the memory address used when PE accesses memory. The green arrows represent the **internal signals** controlling the circuit activity. **Memory Control Signals** include signals such as **write_sel, strobe**. Combination Logic Circuit normally includes two parts, one part deals with the computation logic design such as the implementation of the wavelet transform algorithm. The other deals with memory access. Janus memory FSM is designed to generate **feed back control signals** and the **Memory Control Signals**. The **done** signal is asserted when the circuit finishes the computation and interrupts the host.

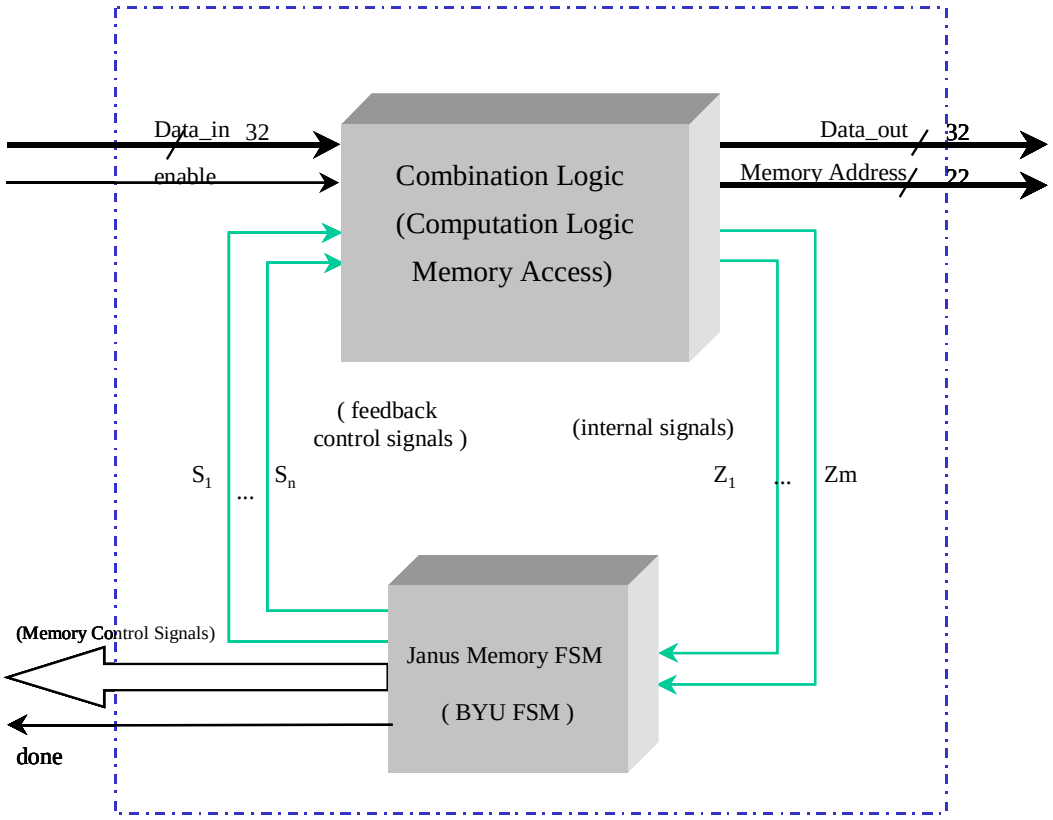


Figure 20. Janus combination logic circuit design and memory FSM model.

5.5.3 Wavelet Transform Hardware Component Design

This section will discuss how the wavelet transform hardware component has been designed using Janus Memory-PE interface and Combination Logic Circuit Memory FSM design model.

In Figure 16, the `WTOp` class has a function call `buildOperation`, which deals with the wavelet transform hardware component design. The `buildOperation` function consists of `WTMeminterface` and `DWT` objects. Figure 21 shows how these two classes match the design

model. The `DWT` class is the wavelet transform algorithm logic circuit design. `WTMeminterface` class has the Janus Memory-PE interface as its parameters [Listing 5-2]. It includes the memory access combination logic design and calls a JHDL finite state machine `WtMemFSM.fsm`.

The pipeline concept is used in the wavelet transform algorithm design. According to equations (4)-(7), the calculation can be implemented in hardware using registers, shifters, and adders. Figure 22 shows the hardware schematic of the wavelet transform. The calculation of the first and last two pixels are a little different than the other pixels; more input wires are used to cope with the boundary condition. The wire **writeS0_en** is used to enable the output of the first pixel, the wire **writeLD_en** for the last pixel, and **writeLS_en** for the last even pixel. The **wirteS_en** and **writeD_en** are used to enable the outputs of the even and odd pixels between the first pixel and the last even pixel. The output is chosen by a multiplexer. The JHDL code used to design this circuit is given in Appendix 1. of the DWT class.

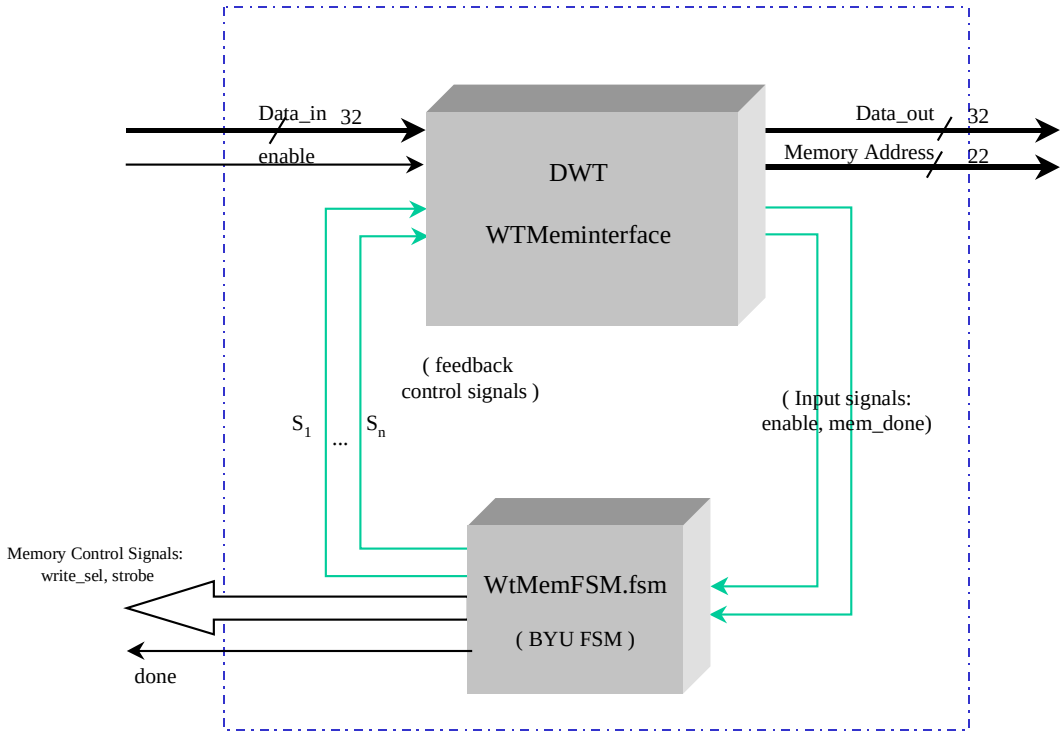


Figure 21. Wavelet transform hardware design using
Janus combination logic circuit and memory FSM design model.

`WtMemInterface` class describes the memory access logic. It does the following tasks: holds the first/last read/write address of the memory, uses the counters to hold the current read/write address, generates current memory access address etc. It also calls the `WtMemFSM` object [Appendix 3], which use a `WtMemFSM.fsm` [Appendix 4] transition table input, to generate the internal circuit and memory access control signals.

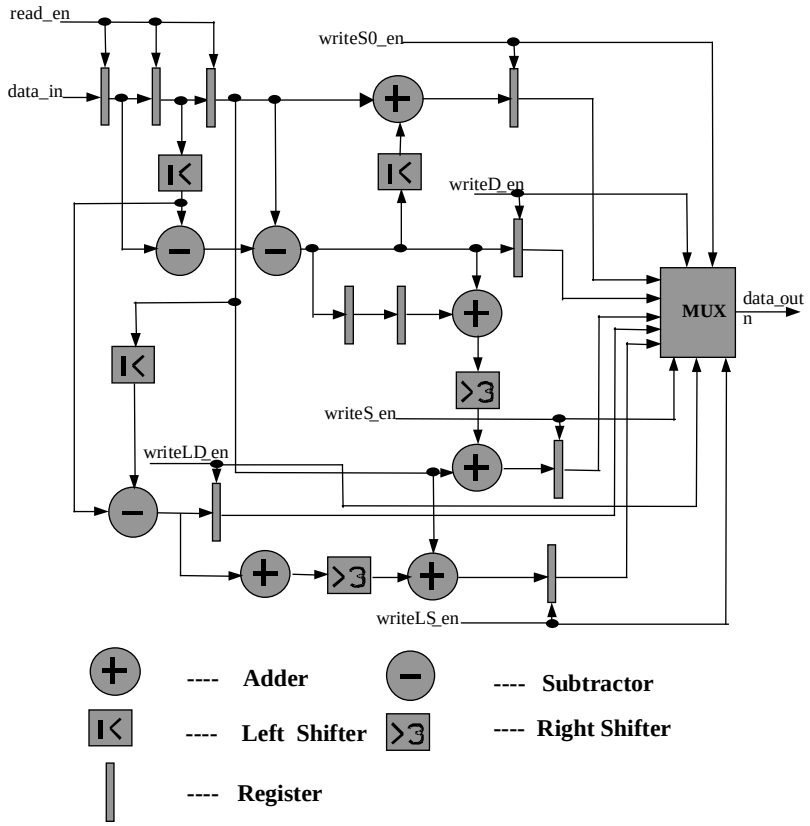


Figure 22. The wavelet transform circuit schematic.

5.6 Janus Simulation Environment

Janus applications need to be verified for correct functionality before execution in hardware. This is done through simulation. The simulation includes three parts. One part deals with the hardware component circuit design in JHDL. This is done using the JHDL simulator. The second part is simulation of each stage's system combining the hardware component and host-side's software component. The third part is to run whole system simulation. The developer can use Janus to simulate the system. This section discusses the simulation tools provided by JHDL, and the simulation environment in Janus tool.

In JHDL, a standard way to simulate a circuit is with a testbench. A testbench in JHDL is a Java class that constructs a circuit, puts values onto the input wires, and records values off the output wires [9]. In IWC implementation, testbenches are used to test the designed circuits.

Appendix 6 is a testbench for `DWT` circuit design. The `DWT` circuit must be placed in a `HWSystem` object, which is in the `main()` function at end of the code. The `main()` function is used for the command line simulation. `hw.cycle(15)` means to step the clock 15 times. The testbech must have a `reset()` and a `clock()` method. Input wires must have values in these two functions. The `clock()` method of the testbench is called on every clock cycle. It is there that the developer can debug his or her circuit step by step.

The testbench uses the `put()` and `get()` methods of the `Wire` class to drive a signal and get a signal. The following code is in `clock()` method to assign "1" to `read_en` signal, and check the `data_out` signal value by using `data_out.get(this)` call.

```
read_en.put(this,1);
….
System.out.println("data_out = " + data_out.get(this));
…
```

Testbenches can also be executed in a GUI environment. JHDL provides a tool called JAB (Just Another Browser), which uses a hierarchical tree view and schematics to interactively browse the hierarchy of a JHDL circuit. The developer can visually simulate the circuit and verify the structural correctness of his design by using JAB. Detailed testbench information and JAB are available on BYU's web page [9].

Janus takes advantage of JHDL's simulator and creates its own simulation system. This system combines activities of the hardware component, the software component, and the PE memory. When an application program is simulated, it has the ability to interact with the system exactly as it would with the physical hardware. Class `ImageCompressionOp` in IWCI implements **`Simulatable`** interface. It has `reset(), clock(), and simulate()` methods where developer can simulate and debug the design. Listing 5-3 shows the code.

```
..// declaration  wires
      private transient Sim_tb tb;
      private transient Wire addr;
      private transient Wire data_in;
      private transient Wire data_out;
      private transient Wire write_sel;
      private transient Wire strobe;
      private transient Wire enable;
      private transient Wire done;

      private transient int mem[];
      private transient int data_in_del;
.. .
public void reset() {
            data_in_del = 0;
            data_in.put(tb, data_in_del);
            enable.put(tb, 1);
      }
public void clock() {
            enable.put(tb, 1);
            int a = addr.get(tb);
            System.out.print("address: " + a + "value" + mem[a]);
            data_in.put(tb,data_in_del);
            if( a < mem.length ) {
                  data_in_del = mem[a];
            }
            int out = data_out.get(tb);
```

```
            System.out.print(" out:" + out);
            if( (write_sel.get(tb) == 0) && (strobe.get(tb) == 0) ) {
                  int d = data_out.get(tb);
                  mem[a] = d;
                  data_in_del = d;
            }
public void simulate(Hashtable mems) {
             Enumeration e = mems.elements();
             mem = (int[])e.nextElement();
             this.setupForSim();
             SimModel sim = this.getSimModel();
             sim.reset(this);

          // System.out.println("Simulate: About to cycle");
            int c = 0;
            while( done.get(tb) != 1 ) {
               //System.out.println("c:" + c + " done:" + done.get(tb));
               sim.cycle(1);
               // c++;
            }
            System.out.println("c:" + c + " done:" + done.get(tb));
      }
```

Listing 5-3. Simulation environment in `ImageCompressionOp` class.

As shown above, the declaration of wires at the beginning of `ImageCompressionOp` is the Janus PE-Memory's interface input and output signals. The `clock()` function simulate the hardware's PE-Memory activity. `mem[]` declaration is the simulated PE memory to store the data. `data_in_del` denotes the time delay when PE read data from memory. When `write_sel` and `strobe` are low, the PE writes data to a PE memory. The `Simulate(Hashtable mems)`function is used to simulate the system. `this` points to the `ImageCompressionOp` object. `Sim` is a `SimModel` object; the constructor of `SimModel` class is shown in Listing 5-4. It first creates a new `HWSystem` and then places the testbench inside the `HWSystem` by passing the `HWSystem` pointer to the testbench constructor. After creates the wires for the PE-Memory

interface, the constructor instantiates the Image Wavelet Compression application hardware circuit that is to be simulated. The line `sim.cycle(1)` in `Simulate(Hashtable mems)`function in `ImageCompressionOp` class instructs the `HWSystem` to step its clock until the interrupt signal *done* is assert. In the IWC implementation, the whole system simulation required around four hours to finish when run on a Pentium333 computer with 256MB, and under Microsoft WindowsNT 4.0 operation system.

```
public class SimModel{
..
public SimModel(ImageCompressionOp protoOp) {
      hw = new HWSystem();
      tb = new Sim_tb(hw,null);
      addr = Logic.wire(tb,20,"addr");
      data_in = Logic.wire(tb,32,"data_in");
      data_out = Logic.wire(tb,32,"data_out");
      write_sel = Logic.wire(tb,1,"write_sel");
      strobe = Logic.wire(tb,1,"strobe");
      enable = Logic.wire(tb,1,"enable");
      done = Logic.wire(tb,1,"done");
try {
protoOp.buildOperation(tb,data_in,data_out,addr,write_sel,strobe,enable,done);
      } catch( JanusException ex ) {ex.printStackTrace();}
}
}
..
}
```

Listing 5-4. The class `SimModel`'s constructor.

5.7 JHDL Synthesis

The Janus application programs, programmed in JHDL, are synthesized to Wildforce FPGA logic designs. JHDL provides the logic synthesis package. The synthesis steps, starting

from high-level JHDL descriptions, include conversion from JHDL to EDIF netlist format, and conversion of EDIF to bitstream files.

After “netlisting” the design, the developer can use JHDL’s makefile to back-end compile the netlist. This targets the Xilinx XC4000 family of FPGAs, placing and routing the designs.

Chapter 6

Hardware Execution Results and Analysis

This chapter describes the execution of the Image Wavelet Compression (IWC) application on the Wildforce configurable computing board. The Janus graphical user interface, which is needed to run the application, is also introduced in this chapter. Results include the processing element utilization, configuration time, hardware run time, and other overhead times are analyzed in detail.

6.1 Processing Element (PE) Utilization

The processing element utilization can be obtained when the design is synthesized. Table 1

Table 1. Image Wavelet Compression PE Utilization for Each Routine

	Number of CLBs used	Percentage
Wavelet Transform	443	19%
Quantization	1450	62%
Run Length Encoding	453	19%
Entropy Coding	827	35%

shows the PE utilization for each of the IWC implementation's routine. Since XC4062XL FPGAs are used on the board, there are 2,304 available CLBs in each PE. The percentage in Table 1 is calculated by taking the number of CLBS needed for each reconfiguration and dividing by the total available CLBs in a PE.

6.2 Janus Graphical User Interface (GUI)

Janus provides a friendly graphical user interface, as shown in Figure 23. In the "*WildForce*" panel, the user clicks the *Open* button to open a Wildforce board, then types in the

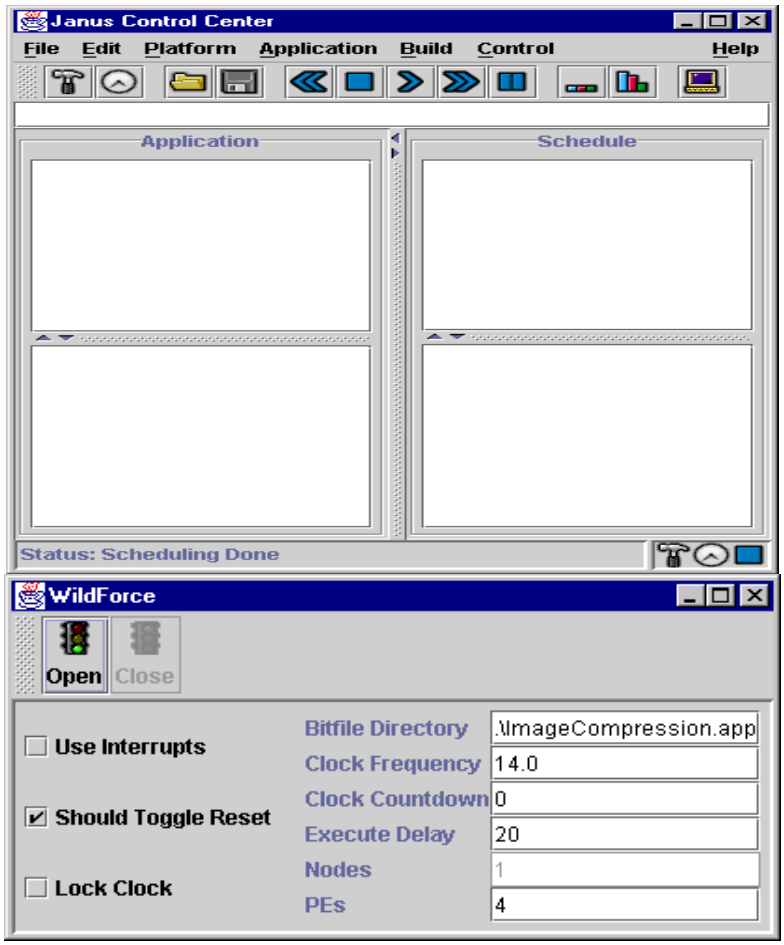


Figure 23. Janus's GUI to control the application's execution.

application's *Bitfiles Directory* and enters the execution *Clock Frequency*. The *Execute delay* value, which is entered as 20ms, determines the host's waiting time for the PE to finish computation. The user can also choose the *Use Interrupts* option. This allows the host communicates with the PE by using interrupt. Unfortunately, this feature currently has flaws. The user can also choose up to four PEs to run the application in parallel. In the "*Janus Control*" Panel, the user can use menus or icons to control the hardware execution. For example, the user can click on the *axe* icon and then the *clock* icon to schedule the application's event, then use the ">>" icon to run the application.

6.3 Hardware Execution Results and Analysis

After the IWC application is executed, a compressed image file is obtained. Figure 24 shows an original image used to be compressed by the application. Figure 25 shows the recovered image after decompressing the compressed image. The decompressed image is a little fuzzy; however, the image is compressed to 2.33% of the original size.

Table 2 summarizes the average execution time using different number of PEs to compress a 512×512 image. These data are collected during application execution and demonstrate the properties of the Janus tool. The host computer, which is used to run the Janus tool to control the Wildforce, is a 300 MHz Pentium II with 256MB memory.

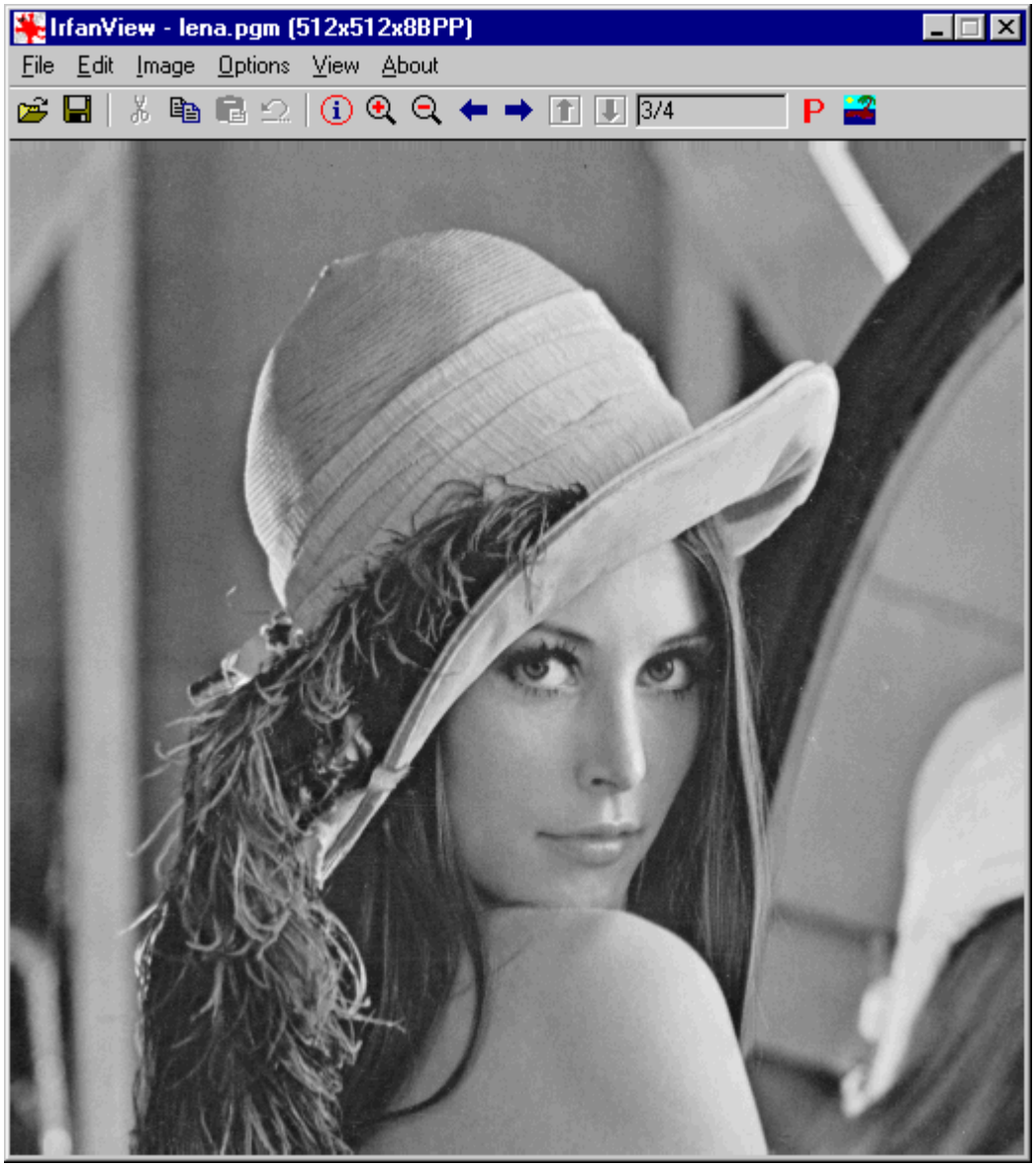


Figure 24. Input image for the Image Wavelet Compression algorithm.
The grayscale image is of size 512 rows x 512 columns.

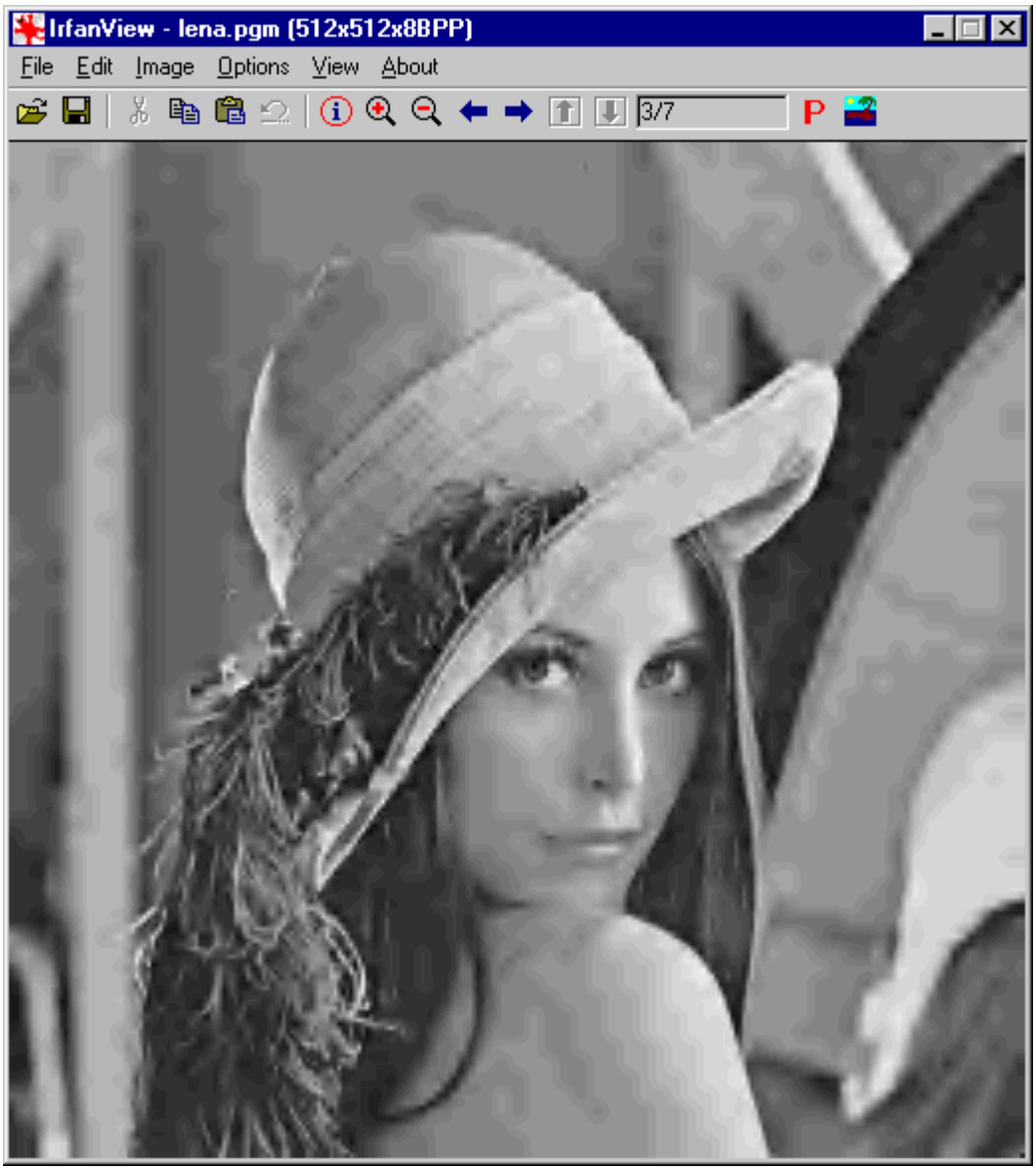


Figure 25. Decompressed image from the compressed image.

.

The application has been tested with a single PE, as well as with two PEs, three PEs, and four PEs. The data collected under these four conditions include the *Run Time* (RT) and the *numbers of Runtime Events* (RE) in PE configuration, hardware interlude, software interlude, retire memory, and fetch memory activities. The Runtime Events are created by the Janus tool to schedule the operations, and the number of events follows the following relation.

$$\textit{Total Runtime Event (RE)} = \textit{Configure PE (RE)} + \textit{Hardware Interlude (RE)} + \textit{Software Interlude (RE)} + \textit{Retire Memory (RE)} + \textit{Fetch Memory (RE)} \quad (9)$$

Table 2. The Average Execution Time for Using Different Number of Pes to Compress a 512×512 Image.

	1 PE		2 PE		3 PE		4 PE	
	Event	Time(s)	Event	Time (s)	Event	Time (s)	Event	Time (s)
Application	--	38.926	--	22.703	--	19.128	--	18.557
Configure PE	9	1.932	9 (18)	3.968	9 (27)	6.308	9(36)	8.823
Hardware Interlude	1807	36.324	905	18.135	607	12.168	453	9.073
Software Interlude	20	0.02	20	0.02	20	0.02	20	0.02
Retire Memory	1807	0.28	1807	0.26	1807	0.27	1807	0.28
Fetch Memory	1807	0.37	1807	0.32	1807	0.362	1807	0.341
Runtime Event	5450	--	4548	--	4250	--	4096	--

Application (RT) = Configure PE (RT) + Hardware Interlude (RT) (10)

+ Software Interlude (RT) + Retire Memory (RT) + Fetch Memory (RT)

The total application run time with one, two, three, or four PEs are 38.926, 22.703, 19.128, and18.557 seconds, respectively. It is obvious that as the number of PE's increase, the total time is decreased, creating a smoother flow, and is illustrated in Figure 26.

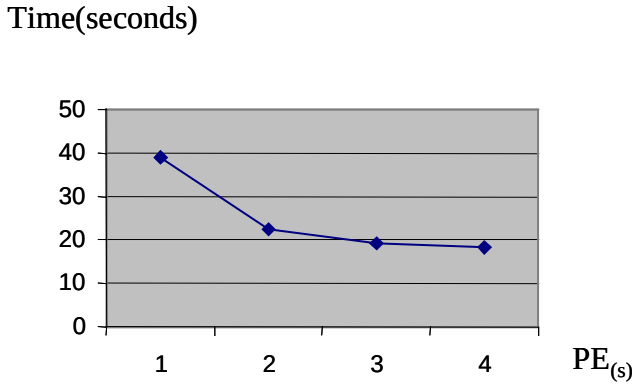


Figure 26. IWC application run time comparison when run on one, two, three, and four PEs.

The Retire Memory and Fetch Memory runtime events tell how many times the host sends data to or fetches data from the PE memories. The numbers are fixed regardless of how many PEs are used. In IWC application, the Retire or Fetch Memory runtime events are always 1807 and are calculated as follows.

Retire Memory (RE) = 512 + 512 + 256+256 + 128 + 128 + 7 + 7 + 1 =1807 (11)

Fetch Memory (RE) = 512 + 512 + 256+256 + 128 + 128 + 7 + 7 + 1 =1807 (12)

Because the application has the same Retire Memory and Fetch Memory runtime events for one, two, three, or four PEs, the host will take the same run time to compute these events.

The data in the Table 2 shows that the Fetch Memory events take around 0.36 seconds while the Retire Memory events take around 0.27 seconds.

Hardware Interlude time denotes how much run time is spent on hardware calculation. This parameter is different under these four conditions because as application runs in parallel it will naturally distribute the calculation time. This run time closely relates to the predicated Hardware Interlude runtime event numbers. The following formula models this relation.

$$\textit{Hardware Interlude (RT)} = \textit{Hardware Interlude (RE) x each computation's time (0.02s)} \quad (13)$$

Each computation time has a fixed value of 0.02s, since 20 ms is entered as the execution delay when the application is executed [Section 6.2]. The Hardware Interlude runtime events are different when application uses a different number of PEs to do the computation. For example, when it runs on one PE, the following calculation is used to obtain the Hardware Interlude event numbers.

$$\textit{Hardware Interlude (RE with 1 PE)} = 512 + 512 + 256 + 256 + 128 + 128 + 7 + 7 + 1 = 1807 \quad (14)$$

When the application run on 4 PEs, the following value is obtained.

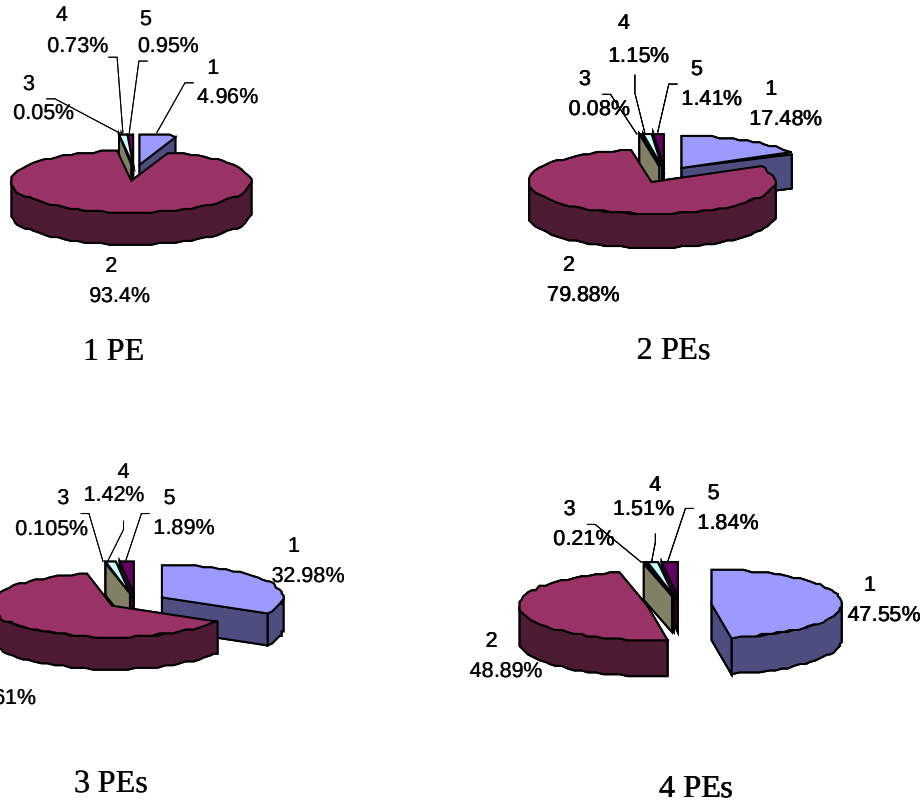
$$\textit{Hardware Interlude (RE with 4 PEs)} = \left\lceil \frac{512}{4} \right\rceil + \left\lceil \frac{512}{4} \right\rceil + \left\lceil \frac{256}{4} \right\rceil + \left\lceil \frac{256}{4} \right\rceil + \left\lceil \frac{128}{4} \right\rceil + \left\lceil \frac{128}{4} \right\rceil + \left\lceil \frac{7}{4} \right\rceil + \left\lceil \frac{7}{4} \right\rceil + \left\lceil \frac{1}{4} \right\rceil = 453 \quad (15)$$

From the calculation, one PE has 1807 hardware interlude runtime events, while two, three, and four PEs have 905, 607, and 453, respectively. After using these values in Equation

13, one PE should use 1807x0.02=36.14 seconds in hardware calculation, two PEs, three PEs, and 4 PEs should use 18.1, 12.1, and 9.06 seconds. These theoretical values are nearly the same as the experimental data: 36.324, 18.135, 12.168 and 9.073 seconds shown in Table 2. It is obvious that as the number of PEs used is increased, the Hardware Interlude run time decreases in inverse proportion. For example, 4 PEs require about 1/4 of one PE's Hardware Interlude run time.

The PE configuration runtime events are the same in four conditions, while these configuration events use a different run time. This is because in parallel computing, the configuration of several PEs is not done in parallel but sequentially. In the IWC implementation, there are nine configuration runtime events. For a single PE, nine configurations were required to execute the entire computation. For *N* PEs, each PE was configured nine times. There are a total of N x 9 reconfigurations needed. From this point, the more PEs used in parallel, the more reconfiguration time is required. Hence, a single PE uses the least reconfiguration time.

Figure 27 shows these five types of run times using different numbers of PEs to compress a 512 ×512 image. From the figure, it can be calculated that the sum of Configuration Time and Hardware Interlude Time occupies more than 95% of the total application run time in all four conditions. This value shows the efficiency of the Janus tool. This means that the overhead in using the Janus tool design is very small. If we consider the Hardware Interlude Time percentage as a parameter to value the computation efficiency, the efficiency is decreased as more PEs are used. Therefore, using one PE has the highest efficiency. However, one PE takes much longer to run the application comparing with that of more PEs. There are tradeoffs among how fast the application should be run, how many resources (number of PEs) to use, and the desired computational efficiency.



	1	2	3	4	5
1 PE	0.0496	0.933	0.0005	0.0073	0.0096
2 PEs	0.1748	0.7988	0.0008	0.0115	0.0141
3 PEs	0.3298	0.6361	0.001	0.0142	0.0189
4 PEs	0.4755	0.4889	0.0021	0.0151	0.0184

Figure 27. Four types of run time occupation percentage
in using different number of PEs to compress a 512 ×512 image.

Chapter 7

Conclusions and Suggestions for Future Work

The Image Wavelet Compression (IWC) algorithm has been introduced. The algorithm has been implemented on a commercial configurable computing platform, the Wildforce board, within a run time reconfigurable computing environment. The example input image is a 512- row by 512-column PGM format file and has been compressed to 2.33% of the original size. To implement the IWC application on the run time reconfigurable computing machine, a high-level automated design tool, Janus, has been used throughout the whole application design, development, and execution process.

The Janus tool was introduced with emphasis on its structure. Packages and interfaces, with which the developer needs to use to design the applications, were described in detail. The Janus application design process was also presented. The Janus application partition model, software component design interface, hardware component design model, and simulation environment have been described. The IWC application has been used as an example to demonstrate these concepts.

Janus uses JHDL as the hardware design language. JHDL is a Java based object-oriented hardware description language that can be used by the developer to express circuit organization.

It is an easy language to learn and naturally suitable for developing applications in a reconfigurable system. JHDL provides a software simulation kernel that can simulate a design using a testbench, or a JAB GUI

The IWC application has been tested on one, two, three, and four PEs through the Janus GUI. The data, including the configuration time, hardware run time, and other overhead times were collected and analyzed. The data has shown that the overhead time due to Janus is no more than 5% in all four tests, which shows the efficiency of the Janus tool. The data also shows the tradeoffs among resource cost, application run speed, and computation efficiency.

After analyzing the execution data, some future improvements are possible. It is obvious that the hardware execution time and the configuration time take the largest percentage of the total application run time. Since the IWC application does not use the interrupt signal for host-hardware communication, interrupts should be considered, instead of using execution delay to reduce the hardware run time. To decrease the configuration time, the Janus tool can be modified to decrease the configuration times. For example, when the application runs on one PE, it reconfigures the PE nine times because it has nine dependent stages. Although the first six stages are dependent (level1-rows, level1-columns, level2-rows, level2-columns, level3-rows, level3-columns), they actually use the same wavelet transform algorithm. Therefore, the total reconfiguration count should reduce to four. Another way to reduce the configuration time is to investigate the possibility of applying the partial configuration property.

References

[1] Rhett D. Hudson, "Architecture-Independent Design for Run-Time Reconfigurable Custom Computing Machines," Ph.D. dissertation, Virginia Polytechnic Institute and State University, in processing.

[2] Luiz Pires and Deepa Pandalai, Honeywell Technology Center, Minnepoli, MN 55412, "compress.c, " October 1988.

[3] J. Villasenor, B. Belzer, J. Liao, "Wavelet Filter: Evaluation for Image Compression," *IEEE Transactions on Image Processing*, vol. 2, pp. 1053-1060, August 1995.

[4] A. Cohen, I. Daubechies, J. Feauveau, "Biorthogonal: Bases of Compactly Supported Wavelets," *Comm. Pure: Math*, 45, 1992.

[5] Annapolis Micro Systems, Inc., " WILDFORCE Reconfigurable Computing Engines Manual," Annapolis, MD, 1998.

[6] Peter Bellows and Brad Hutchings, "JHDL - An HDL for Reconfigurable Systems," *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 175-184, April 1998.

[7] John Villasenor and Brad Hutchings, " The Flexibility of Configurable Computing," *IEEE Signal Processing Magazine*, vol. 15, pp. 67-83, September 1998.

[8] On-line material "XC4000E and XC4000X series Field Progammable Gate Arrays" http://www.coolpld.com/products/xc4000xv.htm May, 2000.

[9] On-line material " BYU Electrical Engineering's Configurable Computing Laboratory" http://splish.ee.byu.edu/lab/jhdl/wildforce/tutorial.html May, 2000.

Appendix 1

JHDL Hardware Circuit Design File (DWT)

This is a sample JHDL file to describe the wavelet transform hardware circuit. The circuit schematic has been shown in Figure 22.

```
/* File:  DWT.java, for wavelet transform routine */

package visc.cc.rtr.apps.ImageCompression; // Input packages will be used
import byucc.jhdl.base.*; //this allows to access all the base JHDL classes
import byucc.jhdl.Logic.*;
import byucc.jhdl.Xilinx.*;

/* The implementation is to run on a Xilinx 4K part, so libraries describing
Xilinx and Xilinx 4K specific structures should be inputted */
import byucc.jhdl.Xilinx.XC4000.*;
import byucc.jhdl.Xilinx.XC4000.carryLogic.*;

public class DWT extends Logic
{
//Standard cell interface
      public static final String[] portnames  = {"data_in", "read_en",
           "writeD_en", "writeS_en", "writeS0_en",  "writeLD_en",
           "writeLS_en", "data_out"};

//define input bit width
      public static final String[] portwidths = {"gw",    "1",     "1",
            "1","1","1","1","gw"};

      public static final String[] portios = {"in",    "in",    "in",   "in",
            "in","in","in",  "out"};
      public static final String[] generics   = {"gw"};
      public static final String cellname ="DWT";
```

```
      public DWT(Node parent, Wire data_in, Wire read_en, Wire writeD_en, Wire
           writeS_en, Wire writeS0_en,  Wire writeLD_en, Wire writeLS_en, Wire
           data_out){

/* This line must be the first line of each of circuit constructors. It is
required so JHDL knows who the hierarchical parent of the circuit is */
            super(parent, cellname);

 // make sure input and output bit width is same
            if ( data_in.getWidth()!= data_out.getWidth()){
                        throw new BuildException(cellname + ":input and output
                        widths must be the same");
            }

            int width = data_in.getWidth();
            this.setGeneric("gw", width);

/*connect wires to the ports*/
            port("data_in", data_in);
            port("read_en", read_en);
            port("writeD_en", writeD_en);
            port("writeS_en", writeS_en);
            port("writeS0_en", writeS0_en);
            port("writeLD_en", writeLD_en);
            port("writeLS_en", writeLS_en);
            port("data_out", data_out);

/* describe wires with the name and bit width */
            Wire S1 =wire(width,"S1");
            Wire S2 =wire(width,"S2");
            Wire Sout =wire(width,"Sout");
            Wire S0out =wire(width,"S0out");
            Wire D1 =wire(width,"D1");
            Wire Dshift =wire(width,"Dshift");
            Wire Sshift =wire(width,"Sshift");
            Wire Dout   =wire(width,"Dout");
            Wire Dpre   =wire(width,"Dpre");
            Wire lastDout =wire(width,"lastDout");
            Wire lastSout =wire(width,"lastSout");
```

```
/* describe the logical function of the cell */
            regce_o(data_in, read_en, S2); // S2 is output
            regce_o(S2, read_en,D1);
            regce_o(D1,read_en,S1);
            Dshift=shiftl(D1,1);
            Sshift=shiftl(S1,1);
            Dout = sub(sub(Dshift, S1),S2);
            Dpre =
regce(regce(Dout,or(writeLD_en,writeD_en)),or(writeLD_en,writeD_en));
// calculate new even pixel
            Sout = add(ashiftr(add(Dout,Dpre),3),S1);
// the first even pixel
            S0out = add(ashiftr(Dout,2),S1);
//calculate the last odd pixel
            lastDout = sub(Dshift,Sshift);
            lastSout =
add(ashiftr(add(sub(Dshift,Sshift),Dpre),3),S1);//calculate last even pixel
//The output multiplexer
      mux_o(mux(mux(mux(mux(constant(width,0),Dout,writeD_en),Sout,writeS_en),
            S0out,writeS0_en),lastDout,writeLD_en), lastSout,writeLS_en,
            data_out);
      }

}
```

Appendix 2

Janus Memory-PE JHDL File (WtMemInterface)

This sample file is used to demonstrate how the Janus Memory-PE interface has been used in wavelet transform hardware circuit design. The interface has been introduced in Section 5.5.1.

```
/* File name: WtMemInterface.java, in wavelet transform routine */
public class WtMemInterface extends Logic
{
// cell interface
      public static final String[] portnames  = { "data_in",      "addr",
           "write_sel",  "strobe",  "enable",     "read_en",  "writeD_en",
           "writeS_en", "writeS0_en",      "writeLD_en",     "writeLS_en",
            "done_read_n",    "done" };

      public static final String[] portwidths = { "32","20", "1", "1", "1",
      "1", "1", "1","1","1", "1","1","1" };
      public static final String[] portios    = { "in","out", "out",  "out",
      "in", "out", "out", "out","out", "out", "out", "out","out" };
      public static final String[] generics   = { "dw", "aw" };
      public static final String cellname = "MemoryInterface";
      private Wire read_en, writeD_en, writeS_en, writeS0_en,writeLD_en,
      writeLS_en;

// Janus Memory-PE interface
      public WtMemInterface(Node parent, Wire data_in, Wire addr, Wire
            write_sel, Wire strobe, Wire enable, Wire done_read_n,Wire done)
                  throws JanusException
      {
            super(parent);

/* Check the widths of memory and the number of address to make sure we can
execute on this architecture */
            if( data_in.getWidth()!=32 || addr.getWidth()!=20 )
            {
```

```
                throw new JanusException("Memory interface is not supported
            by application.");
            }

/* describe wires with the name and bit width */
            read_en = Logic.wire((Cell)parent,1,"read_en");
            writeD_en =Logic.wire((Cell)parent,1,"writeD_en");
            writeS_en =Logic.wire((Cell)parent,1,"writeS_en");
            writeS0_en =Logic.wire((Cell)parent,1,"writeS0_en");
            writeLD_en =Logic.wire((Cell)parent,1,"writeLD_en");
            writeLS_en =Logic.wire((Cell)parent,1,"writeLS_en");

/* connect wires to the ports */
            port("data_in",data_in);
            port("addr",addr);
            port("write_sel",write_sel);
            port("strobe",strobe);
            port("enable",enable);
            port("read_en", read_en);
            port("writeD_en", writeD_en);
            port("writeS_en", writeS_en);
            port("writeS0_en", writeS0_en);
            port("writeLD_en", writeLD_en);
            port("writeLS_en", writeLS_en);
            port("done_read_n",done_read_n);
            port("done",done);

/* describe some internal use wires with the name and bit width */
            Wire init = wire(20,"init");
            //register to hold read address
            Wire read_addr = wire(20,"read_addr");
            //register to hold write address
            Wire write_addr = wire(20,"write_addr");
            //register to hold read begin address
            Wire load_read = wire(1,"load_read");
            //register to hold write begin address
            Wire load_write = wire(1,"load_write");
            //control read signal
            Wire cnt_read= wire(1,"cnt_read");
```

```
            //control write signal
            Wire cnt_write= wire(1,"cnt_write");
            // register to hold end read address
            Wire last_read_addr = wire(20,"last_read_addr");
// create registers to hold end write address

            Wire last_write_addr = wire(20,"last_write_addr");
            Wire load_last_read = wire(1,"load_last_read");
            Wire load_last_write = wire(1,"load_last_write");
            Wire done_write = wire(1,"done_write");
            Wire addr_ctr0 = wire(1,"addr_ctr0");
            Wire addr_ctr1 = wire(1,"addr_ctr1");
            Wire writeDout_en =wire(1,"writeDout_en");
            Wire done_writeLD = wire(1,"done_writeLD");

// The composite done signal
            Wire mem_done = done_write;

// The address counters
            new AddrCounter(this,enable,gnd(),constant(20,0),init);

// Initialization address generation counter
            new
                AddrCounter(this,and(enable,cnt_read),load_read,range(data_in,
                19,0),read_addr);
            new
                addrCounter(this,and(enable,cnt_write),load_write,range(data_i
                n,19,0),write_addr);

// load the last read address to register
      regce_o(range(data_in,19,0),and(enable,load_last_read),last_read_addr);
// load the last write address to register
      regce_o(range(data_in,19,0),and(enable,load_last_write),last_write_addr);

// The comparators
      Wire xnor_read = wire(this,20);
      Wire xnor_write = wire(this,20);
      Wire xnor_writeLD = wire(this,20);
      Wire done_read_pulse = wire(this,1);
```

```
      Wire done_read_stick = wire(this,1);

//XNOR to check two addresses are same (11..11)
      not_o(xor(last_read_addr,read_addr),xnor_read); // xnor_read is output
      not_o(xor(last_write_addr,write_addr),xnor_write);
      not_o(xor(sub(last_write_addr,constant(20,1)),write_addr),xnor_writeLD);

// if finish read, the done_read_pulse will be high
      and_o(and(xnor_read.getWire(0),xnor_read.getWire(1),xnor_read.getWire(2)
            ,xnor_read.getWire(3),xnor_read.getWire(4),xnor_read.getWire(5),xn
            or_read.getWire(6),xnor_read.getWire(7),xnor_read.getWire(8)),and(
            xnor_read.getWire(9),xnor_read.getWire(10),xnor_read.getWire(11),x
            nor_read.getWire(12),xnor_read.getWire(13),xnor_read.getWire(14),x
            nor_read.getWire(15),xnor_read.getWire(16),xnor_read.getWire(17)),
            and(xnor_read.getWire(18),xnor_read.getWire(19)),done_read_pulse);

// if finish write, the done_write will be high
      and_o(and(xnor_write.getWire(0),xnor_write.getWire(1),xnor_write.getWire
            (2),xnor_write.getWire(3),xnor_write.getWire(4),xnor_write.getWire
            (5),xnor_write.getWire(6),xnor_write.getWire(7),xnor_write.getWire
            (8)),and(xnor_write.getWire(9),xnor_write.getWire(10),xnor_write.g
            etWire(11),xnor_write.getWire(12),xnor_write.getWire(13),xnor_writ
            e.getWire(14),xnor_write.getWire(15),xnor_write.getWire(16),xnor_w
            rite.getWire(17)),and(xnor_write.getWire(18),xnor_write.getWire(19
            )),done_write);

// if finish write last Di, the done_ writeLD will be high
      and_o(and(xnor_writeLD.getWire(0),xnor_writeLD.getWire(1),xnor_writeLD.g
            etWire(2),xnor_writeLD.getWire(3),xnor_writeLD.getWire(4),xnor_wri
            teLD.getWire(5),xnor_writeLD.getWire(6),xnor_writeLD.getWire(7),xn
            or_writeLD.getWire(8)),and(xnor_writeLD.getWire(9),xnor_writeLD.ge
            tWire(10),xnor_writeLD.getWire(11),xnor_writeLD.getWire(12),xnor_w
            riteLD.getWire(13),xnor_writeLD.getWire(14),xnor_writeLD.getWire(1
            5),xnor_writeLD.getWire(16),xnor_writeLD.getWire(17)),and(xnor_wri
            teLD.getWire(18),xnor_writeLD.getWire(19)),done_writeLD);

      regc_o(or(done_read_pulse,done_read_stick),done_read_stick);
      not_o(or(done_read_stick,done_read_pulse),done_read_n);
```

```
// For the last Dout write enable;
      and_o(done_writeLD, writeDout_en , writeLD_en);

 // For normal Dout write enable;
      and_o(not(done_writeLD), writeDout_en, writeD_en);

// The address multiplexer 00-read 01-write 11-init 10-undefined
      mux_o(mux(read_addr,write_addr,addr_ctr0),mux((Wire)constant(20,0),init,
            addr_ctr0),addr_ctr1,addr);

// call the FSM that drives the whole thing.
      New
            WtMemFSM(this,enable,mem_done,write_sel,strobe,load_read,load_writ
            e,cnt_read,cnt_write,load_last_read,load_last_write,addr_ctr0,addr
            _ctr1,read_en, writeDout_en, writeS_en,writeS0_en, writeLS_en,
            done);
      }

      public Wire getReadEnable() {
            return read_en;
      }
      public Wire getWriteDEnable() {
            return writeD_en;
      }
      public Wire getWriteSEnable(){
            return writeS_en;
      }
      public Wire getWriteS0Enable(){
            return writeS0_en;
      }
      public Wire getWriteLDEnable(){
            return writeLD_en;
      }
      public Wire getWriteLSEnable(){
            return writeLS_en;
      }
}
```

Appendix 3

FSM Used In Hardware Design (WtMemFSM)

Class Fsm (Finite State Machine), a state machine synthesizer from JHDL, provides support for the description of the state machine in a transition table format. To create a state machine, two files - a JHDL object and a text file containing machine's transition table should be programmed. This file is an example of wavelet transform FSM JHDL object file. Appendix 4 shows the text file. Figure 21 shows the model.

```
/* File: WtMemFSM.java,  used in Wavelet Transform routine  */
package visc.cc.rtr.apps.ImageCompression;
import byucc.jhdl.base.*;
import byucc.jhdl.Logic.*;
import byucc.jhdl.Fsm.*;
import java.net.*;

// create a JHDL module which extends Fsm
public class WtMemFSM extends Fsm
{
      public     final     static     String[]     portnames     =          {
          "enable","mem_done","write_sel","strobe","load_read","load_write","
          cnt_read","cnt_write","load_last_read","load_last_write","addr_ctrl
          0", "addr_ctrl1",  "read_en",      writeDout_en",      "writeS_en",
          "writeS0_en",  "writeLS_en", "done"};
      public final static String[] portwidths = { "1",  "1",  "1", "1", "1",
          "1",    "1",   "1",   "1","1",   "1", "1 "1","1",   "1", "1",   "1",
          "1"};
      public final static String[] portios =    { "in",  "in",  "out",  "out",
      "out",  "out",  "out","out",  "out", "out", "out", "out","out", "out",
      "out",  "out", "out","out"};

// Constructor
      public WtMemFSM (Node parent, Wire enable,Wire mem_done, Wire write_sel,
          Wire strobe, Wire load_read, Wire load_write, Wire cnt_read, Wire
          cnt_write,   Wire   load_last_read,   Wire   load_last_write,   Wire
```

```
            addr_ctrl0, Wire addr_ctrl1, Wire read_en, Wire writeDout_en, Wire
            writeS_en, Wire writeS0_en, Wire writeLS_en, Wire done)
      {
            super (parent, "WtMemFSM");
            port("enable", enable);
            port("mem_done", mem_done);
            port("write_sel", write_sel);
            port("strobe", strobe);
            port("load_read", load_read);
            port("load_write", load_write);
            port("cnt_read", cnt_read);
            port("cnt_write", cnt_write);
            port("load_last_read", load_last_read);
            port("load_last_write", load_last_write);
            port("addr_ctrl0", addr_ctrl0);
            port("addr_ctrl1", addr_ctrl1);
            port("read_en", read_en);
            port("writeDout_en", writeDout_en);
            port("writeS_en", writeS_en);
            port("writeS0_en", writeS0_en);
            port("writeLS_en", writeLS_en);
            port("done",done);

// Do the actual Fsm construction
            buildFsm("WtMemFSM.fsm");
      }
}
```

Appendix 4

FSM Used In Hardware Design (WtMemFSM.fsm)

This is an FSM text file used in the wavelet transform hardware circuit design.

```
/* File name: WtMemFSM.fsm, used in Wavelet Transform routine */
. .inputs enable mem_done; // input signals
// output signals include memory control signals and feedback control signals
.outputs write_sel strobe load_read load_write cnt_read cnt_write
load_last_read load_last_write addr_ctrl0 addr_ctrl1 read_en writeDout_en
writeS_en writeS0_en writeLS_en done;

//states
.states a b c d e f g h i j k l m n o p q r s z;
.encodings default;

0- a a 1000000011000000;      //let initial address available
1- a b 1000000011000000;
0- b b 1000000011000000;      //let initial address available
1- b c 1000000011000000;
0- c c 1010100011000000;      //load the first read address
1- c d 1010100011000000;
0- d d 1001010011000000;      //load the first write address
1- d e 1001010011000000;
0- e e 1100001011000000;      //load the last read address
1- e f 1100001011000000;
0- f f 1100000111000000;      //load the last write address
1- f g 1100000111000000;
0- g g 1000100000000000;      //choose the first read address and increments
1- g h 1000100000000000;
0- h h 1000100000000000;      //choose the second read address and increments
1- h i 1000100000000000;
0- i i 1000100000100000;      //read the first data in
1- i j 1000100000100000;
0- j j 1100000000100000;      //strobe and read the second data in
1- j k 1100000000100000;
```

```
0- k k 1100000000100000;      //strobe and read the third data in
1- k l 1100000000100000;
0- l l 0000010010010000;      //write the first data out (boundary condition)
1- l m 0000010010010000;
0- m m 0000010010000100;      //write the second data out (boundary condition)
1- m n 0000010010000100;      //the above is start condition,the below is loop
0- n n 1000100000000000;      //chose the read address 1 and increment.
1- n o 1000100000000000;
0- o o 1000100000000000;      //chose the read address 2 and increment
1- o p 1000100000000000;
0- p p 1100000000100000;      //read data 1 in
1- p q 1100000000100000;
0- q q 1100000000100000;      //read data 2 in
1- q r 1100000000100000;
0- r r 0000010010010000;      //write data 1 out
1- r s 0000010010010000;
00 s s 0000010010001000;      //write data 2 out
10 s n 0000010010001000;
01 s s 0000000010000010;      //write the last data
11 s z 0000000010000010;
-- z z 1100000001000001;      //done
```

Appendix 5

Janus Software Component Design (WTRowOp)

The Janus tool defines a standard memory interface for its own I/O behavior to map the ccm memory. The host-side software component implements this interface on each stage to prepare the input data and to hold the intermediate results with the ccm memory. This file shows the wavelet transform row operations' software component design. Detailed explanation can also refer to Section 5.4.

```
 /* File name: WTRowOp.java used in wavelet transform routine */
// inherit from WTOp class
public class WTRowOp extends WTOp
{
      private int row, rowLen;
      private int width, height;
      private int inID, outID;

// keep these around to avoid memory allocation
      private transient MemoryImage[] pre_mem;
      private transient MemoryImage[] post_mem;
      private transient int[] pre_pixels; // hold input pixels
      private transient int[] post_pixels; // hold output pixels
      private transient int[] post_buffer; // hold the data from PE memory
      private transient int[] config; // hardware control

// constructor
      public WTRowOp(ImageCompression parent, String s, int aRow, int
inputWidth, int inputHeight, int inputID, int outputID) {
            super(parent, s);
            row = aRow;
            width = inputWidth;
            height= inputHeight;
            inID    =inputID;
            outID   =outputID;
            rowLen  =parent.getInputWidth();
            this.init();
```

```
	}

		private void init() {
		int rxw = row * rowLen;
		int rbase = 4; // read begin address
		int wbase = rbase + width; // write begin address
/* Set up hardware initializer: read begin, write begin, read end, write end
addresses*/
		config = new int[] {rbase,wbase,rbase+width-1,wbase+width-1};
		post_buffer = new int[width];

		//call by reference from super class to get the input pixels
		pre_pixels = parent.getPixelsByID(inID);
		post_pixels = parent.getPixelsByID(outID);

		pre_mem = new MemoryImage[2];
		pre_mem[0] = new MemoryImage(config,0,config.length,0);
		pre_mem[1] = new MemoryImage(pre_pixels, rxw, width, rbase);

		post_mem = new MemoryImage[1];
		post_mem[0] = new MemoryImage(post_buffer, 0, width, wbase);
	}

// move data from host memory to ccm memory
	public MemoryImage[] userToCcm() {
		return pre_mem;
	}

// fetch data from ccm memory to host memory
	public MemoryImage[] ccmToBuffer() {
		return post_mem;
	}

// modify the fetched data and put them into the array []post_pixels
	public void bufferToUser(MemoryImage anImage[]) {
		int j=0,k=0;
		for(int i=0; i < width; i++) {
			if(i%2!=0){
// put even pixel on the left
```

```
                  post_pixels[row*rowLen+j] = post_buffer[i];
            j++;
                  }
                  else{
// put odd pixel on the right
                  post_pixels[row*rowLen+(width/2+k)]=post_buffer[i];
                k++;
                  }
            }
      }

}
```

Appendix 6

Janus TestBench Used in Simulation (DWT_tb)

The sample testbench file DWT_tb.java shown is used to simulate the wavelet transform circuit design in Image Wavelet Compression application.

```
package visc.cc.rtr.apps.ImageCompression;
import byucc.jhdl.base.*;
import byucc.jhdl.Xilinx.*;
import byucc.jhdl.Xilinx.XC4000.*;
import java.io.*;
import byucc.jhdl.modgen.*;
import byucc.jhdl.Logic.*;


public class DWT_tb
      extends Synchronous
      implements TestBench
{
// declare local variables and local wire objects connected to circuit object
      private static int count0 = 0;
      private static int count1 = 0;
      private static boolean start = true;
      private static int[] d = {8, 16, 16};
      private Wire data_in;
      private Wire data_out;
      private  Wire  read_en,  writeD_en,writeS_en,  writeS0_en,  writeLD_en,
      writeLS_en;

// constructor builds the desired circuitry
      public DWT_tb(Node parent)
      {
            super(parent);
            data_in = new Xwire(this,32,"data_in");
            data_out = new Xwire(this,32,"data_out");
            read_en = new Xwire(this,1,"read_en");
```

```
            writeD_en= new Xwire(this,1,"writeD_en");
            writeS_en=new Xwire(this,1,"writeS_en");
            writeS0_en=new Xwire(this,1,"writeS0_en");
            writeLS_en=new Xwire(this,1,"writeLS_en");
            writeLD_en=new Xwire(this,1,"wrriteLD_en");

            DWT wt = new DWT(this, data_in, read_en, writeD_en,writeS_en,
            writeS0_en, writeLD_en, writeLS_en, data_out);
      }

// the clock method is called on every clock cycle. It puts integers onto
input wires, then prints out the cycle count and output value
      public void clock()
      {     System.out.println("clock " + count0 + ":");
            if( start ==true ) {
                  if(count1==0){
                        data_in.put(this,d[count0%d.length]);
                        read_en.put(this,1);
                        writeD_en.put(this,0);
                        writeS_en.put(this,0);
                        writeS0_en.put(this,0);
                        writeLD_en.put(this,0);
                        writeLS_en.put(this,0);
                        //data++;
                        count1++;
                        System.out.println("data_in  = " + data_in.get(this));
                        System.out.println("data_out       =       "       +
                        data_out.get(this));
                  }
                  else if(count1==1){
                        data_in.put(this,d[count0%d.length]);
                        read_en.put(this,1);
                        writeD_en.put(this,0);
                        writeS_en.put(this,0);
                        writeS0_en.put(this,0);
                        writeLD_en.put(this,0);
                        writeLS_en.put(this,0);
                        //data++;
                        count1++;
```

```
        System.out.println("data_in  = " + data_in.get(this));
        System.out.println("data_out       =       "       +
        data_out.get(this));
    }
    else if(count1==2){
        data_in.put(this,d[count0%d.length]);
        read_en.put(this,1);
        writeD_en.put(this,0);
        writeS_en.put(this,0);
        writeS0_en.put(this,0);
        writeLD_en.put(this,0);
        writeLS_en.put(this,0);
        //data++;
        count1++;
        System.out.println("data_in  = " + data_in.get(this));
        System.out.println("data_out       =       "       +
        data_out.get(this));
    }
    else if(count1==3){
        data_in.put(this,0);
        read_en.put(this,0);
        writeD_en.put(this,1);
        writeS_en.put(this,0);
        writeS0_en.put(this,0);
        writeLD_en.put(this,0);
        writeLS_en.put(this,0);
        count1++;
        System.out.println("data_in  = " + data_in.get(this));
        System.out.println("data_out       =       "       +
        data_out.get(this));
    }
    else {
        data_in.put(this,0);
        read_en.put(this,0);
        writeD_en.put(this,0);
        writeS_en.put(this,0);
        writeS0_en.put(this,1);
        writeLD_en.put(this,0);
        writeLS_en.put(this,0);
        count1=0;
```

```
                System.out.println("data_in  = " + data_in.get(this));
                System.out.println("data_out       =       "       +
                data_out.get(this));
                start=false;
          }
    }
    else{
          if(count1==0){
                data_in.put(this,d[count0%d.length]);
                read_en.put(this,1);
                writeD_en.put(this,0);
                writeS_en.put(this,0);
                writeS0_en.put(this,0);
                writeLD_en.put(this,0);
                writeLS_en.put(this,0);
                //data++;
                count1++;
                System.out.println("data_in  = " + data_in.get(this));
                System.out.println("data_out       =       "       +
                data_out.get(this));
          }
          else if(count1==1){
                data_in.put(this,d[count0%d.length]);
                read_en.put(this,1);
                writeD_en.put(this,0);
                writeS_en.put(this,0);
                writeS0_en.put(this,0);
                writeLD_en.put(this,0);
                writeLS_en.put(this,0);
                //data++;
                count1++;
                System.out.println("data_in  = " + data_in.get(this));
                System.out.println("data_out       =       "       +
                data_out.get(this));
          }
          else if(count1==2){
                data_in.put(this,0);
                read_en.put(this,0);
                writeD_en.put(this,1);
                writeS_en.put(this,0);
```

```
                        writeS0_en.put(this,0);
                        writeLD_en.put(this,0);
                        writeLS_en.put(this,0);
                        count1++;
                        System.out.println("data_in  = " + data_in.get(this));
                        System.out.println("data_out       =       "       +
                        data_out.get(this));
                  }
                  else {
                        data_in.put(this,0);
                        read_en.put(this,0);
                        writeD_en.put(this,0);
                        writeS_en.put(this,1);
                        writeS0_en.put(this,0);
                        writeLD_en.put(this,0);
                        writeLS_en.put(this,0);
                        count1=0;
                        System.out.println("data_in  = " + data_in.get(this));
                        System.out.println("data_out       =       "       +
                        data_out.get(this));
                  }
            }
            count0++;
      }

// reset all input signals
      public void reset()
      {
            count0 = 0;
            count1=0;
            data_in.put(this,0);
            read_en.put(this,0);
            writeD_en.put(this,0);
            writeS_en.put(this,0);
            writeS0_en.put(this,0);
            writeLD_en.put(this,0);
            writeLS_en.put(this,0);
      }
      public Wire getDataIn() {
```

```
            return data_in;
      }
      public Wire getDataOut() {
            return data_out;
      }

// main() function is used for command line simulation of testbench circuit
      public static void main(String argv[])
      {
// must create a new HWSystem and then place the testbench inside it
            HWSystem hw = new HWSystem();
            DWT_tb tb= new DWT_tb(hw);
// instructs the HWSystem to step its clock 20 times
            hw.cycle(15);
      }
}
```

Vita

The author, Zhimei Ding, received the Bachelor of Engineering degree in Mechanical Engineering Department and minor in Computer Science and Application from Beijing Industry Polytechnic University, Beijing, China, in July 1992. During 1992 to 1996, she worked in ShenZhen Compac Software Inc. and then in Amtronix Inc. as an engineer. In 1997, she joined the Virginia Tech Electrical Engineering Department and worked on the M.S. degree. In 1998, she joined the configurable computing group of Virginia Tech Information Center as a research assistant. Her research interest includes the reconfigurable computing, digital design, and networking.

Upon graduation, she will join Hughes Networking Company, SpaceWay division as a software engineering.