| **REPORT DOCUMENTATION PAGE** | *Form Approved* *OMB No. 0704-0188* |
|---|---|

**1. REPORT DATE** *(DD-MM-YYYY)*
05-05-2009

**2. REPORT TYPE**

**3. DATES COVERED** *(From - To)*

**4. TITLE AND SUBTITLE**

Iris Recognition Using Parallel and Sequential Logic in a Reconfigurable Logic Device

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**

Ulis, Bradley J. (Bradley James), 1985-

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
U.S. Naval Academy
Annapolis, MD 21402

**8. PERFORMING ORGANIZATION REPORT NUMBER**

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**

**10. SPONSOR/MONITOR'S ACRONYM(S)**

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**
Trident Scholar Project Report no. 384 (2009)

**12. DISTRIBUTION / AVAILABILITY STATEMENT**

This document has been approved for public release; its distribution is UNLIMITED

**13. SUPPLEMENTARY NOTES**

**14. ABSTRACT**
Iris recognition demonstrates superior performance as a biometric, far exceeding fingerprint recognition techniques. Unfortunately, iris recognition is computationally intensive, requiring near state-of-the-art traditional processing methods and equipment. Because of the complexity of the iris recognition systems, portable iris scanners are often bulky, cumbersome and expensive. This is due in part to a reliance on sequential processing of recognition algorithms. However, an alternative exists with parallel processing using multicore processors, field-programmable gate arrays (FPGAs) or application specific integrated circuits (ASICs). These devices can speed algorithms through parallel processing. In this work, portions of an algorithm developed at the United States Naval Academy were translated for parallel processing and were placed into a FPGA system. Additionally, the feasibility of entirely embedding an iris recognition system on a single FPGA chip and a discrete memory module was evaluated. The resulting hardware is shown to be between 10 and 1000 times faster than current methods and could be run independent of a host system for

**15. SUBJECT TERMS**
Biometrics, Iris Recognition, Systolic Architecture, Finite Impulse Response Filtering, Binary Morphology

| **16. SECURITY CLASSIFICATION OF:** | | | **17. LIMITATION OF ABSTRACT** | **18. NUMBER OF PAGES** | **19a. NAME OF RESPONSIBLE PERSON** |
|---|---|---|---|---|---|
| **a. REPORT** | **b. ABSTRACT** | **c. THIS PAGE** | | 72 | **19b. TELEPHONE NUMBER** *(include area code)* |

U.S.N.A.-Trident Scholar project report; no. 384 (2009)

# IRIS RECOGNITION USING PARALLEL AND SEQUENTIAL LOGIC IN A RECONFIGURABLE LOGIC DEVICE

by

Midshipman 1/c Bradley J. Ulis
United States Naval Academy
Annapolis, Maryland

(signature)

Certification of Advisers Approval

Assistant Professor Ryan N. Rakvic
Department of Electrical and Computer Engineering

(signature)

(date)

Assistant Professor Randy P. Broussard
Department of Weapons and Systems Engineering

(signature)

(date)

Acceptance for the Trident Scholar Committee

Professor Carl Wick
Associate Director of Midshipman Research

(signature)

(date)

USNA-1531-2

# Abstract

Biometrics technologies have grown considerably in recent years with better computing and an expanding realm in which these tools are deployed. Among these, iris recognition demonstrates superior performance as a biometric, perhaps far exceeding the standard fingerprint recognition of past decades. Unfortunately, iris recognition is very computationally intensive, requiring near state-of-the-art traditional processing methods. Because of the complexity of iris recognition systems, many portable iris scanners are bulky, cumbersome and very expensive, often requiring laptop computers to carry out the computations. This is due to a reliance on sequential processing, the manner of computing we see in a typical personal computer. However, there is an alternative with parallel processing using multicore processors, field-programmable gate arrays (FPGAs) or application specific integrated circuits (ASICs). These devices can speed algorithms through parallel processing. Taking the algorithm developed by Dr. Robert Ives et al. of the United States Naval Academy for iris recognition, parallelizable parts of the algorithm can be translated for parallel processing. A parallel version of the algorithm may be substantially faster and physically much smaller implemented. This implementation is placed into an FPGA system in order to evaluate the performance of specific parts of the algorithm converted from sequential C code to parallel hardware logic with respect to speed and hardware footprint. Additionally, this project seeks to evaluate the feasibility of an entirely embedded iris recognition system comprised of both sequential C software and parallel hardware on a single chip and a discrete memory module. The resulting hardware is shown to be between 10 and 1000 times faster than current methods while being entirely embedded and independent of a host system for processing. These chips could be deployed to offer a handheld, high-speed and palm-sized iris recognition system with all the necessary functionality expected in a commercial iris recognition system.

**Keywords:** Biometrics, Iris Recognition, Systolic Architecture, Finite Impulse Response Filtering, Binary Morphology

## Acknowledgements

I would like to thank the Electrical and Computer Engineering department at the United States Naval Academy and particularly my advisers Dr. Ryan Rakvic and Dr. Randy Broussard for educating me and patiently listening to my theories. Also, thank you to Neil Steiner and Will Kritikos of the Information Sciences Institute (ISI) East for your patience educating me in the discipline of hardware design.

# Contents

## List of Figures

## List of Tables

# Chapter 1

# Introduction

Biometric identification has exploded in recent years as the next generation of security and authentication systems. Biometrics is the study and measurement of the physical characteristics that define an individual's "uniqueness."[1] For example, a common form of biometrics that humans practice daily is facial recognition. In our everyday interactions, we are able recognize friends and family by their familiar faces, which we measure and interpret with our eyes. Humans can also readily recognize familiar people by their height and body shape, voice or even fashion sense. Biometrics sums up all these traits and many more for qualitative or, in the case of computers, quantitative analysis.

While humans have mastered the art of biometric analysis through our natural means, computers offer us exceptionally powerful means of recognition that we have only recently been able to explore. For example, humans lack the capacity to recognize the fine details of a friend's iris beyond eye color, but a computer can document the complex, unique pattern of the iris and recall it later for recognition. This process of recording biometric data into a computer is called enrollment. Computers have also proven capable of other biometric identification tasks such as fingerprinting, DNA analysis, facial recognition, and even palm venous structure recognition.[2] Also, with the exception of DNA analysis, computers obtain most of this data in unobtrusive ways, resulting in no harm to the person being measured. In some cases, a computer can enroll and recognize a person on the fly, which is the focus of portable iris recognition systems.[3]

Iris recognition stands out as one of the most reliable and growing biometric methods in existence. The iris is an internal part of the eye that is protected by the cornea, sustaining its appearance after decades even after facial features and fingerprints have changed. It consists of the sphincter pupillae and the dilator pupillae muscles to vary the diameter of the pupil. Together, these muscles span a flat surface that exhibits a highly unique pattern with nearly infinite variability.[4] An example iris photograph is depicted below.

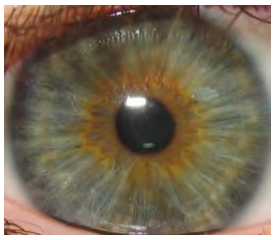


**Figure 1.1: Typical human iris. Also visible are eyelids, eyelashes, and glare.**

Furthermore, biometric technology advancements have made iris recognition systems nearly impossible to fool with high resolution photographs, printed contact lenses or even more menacing means. The iris can be photographed at high resolution at a distance of up to a few meters, even through glasses or contact lenses and in varying light conditions which can alter the visible area of the iris.

Basic iris recognition systems photograph the iris using near infrared (NIR) light and process it for matching. Infrared light can be captured by a typical camera with a special filter lens and allows the camera to capture the structural features of the iris unimpeded by pigmentation or color lighting conditions. Once a digital image of the iris is captured, the system begins processing the image via algorithms to transform it from a two dimensional array of pixels (picture elements) to a one dimensional encoded string of bits for comparison. This transformation from pixels to information is a common digital signal processing technique. Iris recognition systems assume that the features of the iris are so unique that they are like a biological password. This password is the information contained in the iris' structural features and can be determined by noting the frequency and phase information of the iris' features such as in John Daugman's algorithm or other possible methods that will be explored further in this paper. Different iris recognition algorithms exist but Dr. Robert Ives et al. of the United States Naval Academy has developed an algorithm, known as the Ridge Energy Direction (RED) algorithm, which has proven a reliable method for iris recognition.[5] In this algorithm, the first step is to identify the iris among other facial elements such as the eyelids, sclera (white part of the eye), pupil (dark circle in the center of the eye) and eyelashes. The algorithm accomplishes this by scanning the image for the center of the pupil using several segmentation techniques and then uses a statistical approach to identify the outer radius of the iris (limbic boundary). This establishes the central point of the iris within the image's x and y coordinates, allowing the computer to extract only the meaningful portions of the iris.

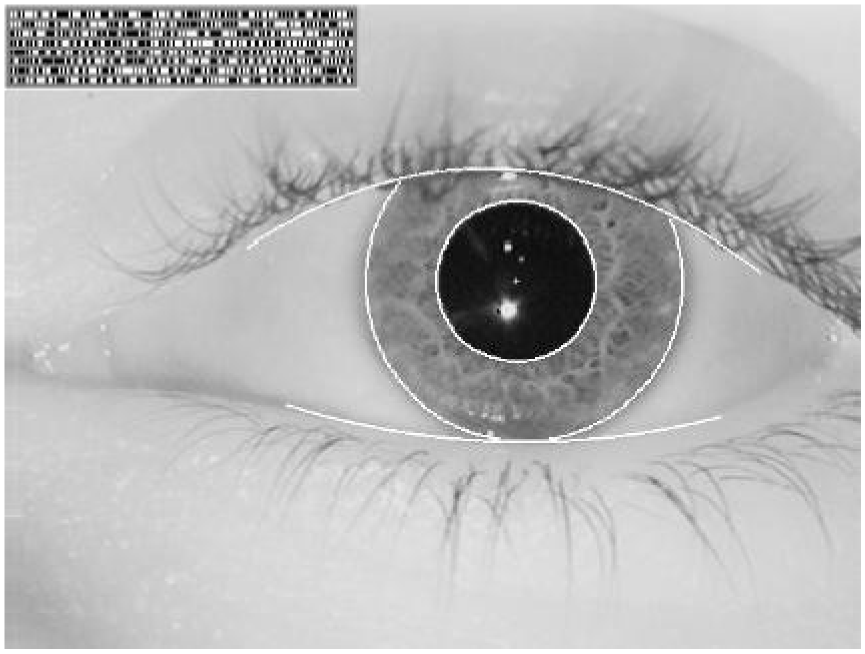


**Figure 1.2: Measurable iris area detected by the algorithm. Also visible is the associated one dimensional encoding of the iris image in the top left.**[6]

The next step is to encode the iris image from two dimensions down to one, also known as a template, in order to compare the iris to a previously enrolled iris template. A template is a bit vector that uniquely represents the information contained in the iris. In biometric algorithms, it is necessary to record the information that the pixels of an image convey rather than the pixels themselves. This is because pixels will differ from one image to another but the information should be consistent if the iris within the image is in fact the same. In the RED algorithm, the iris is unwrapped using polar coordinates and the energy of each pixel is measured (the energy of each pixel is merely the square of the value of the infrared intensity within the pixel). Then, the energy of m x n pixels is passed into four filters to emphasize the existence of ridges and their orientation. The filter with the highest output indicates this orientation and the algorithm represents the orientation using a single bit which is passed into the template. This process is demonstrated below.

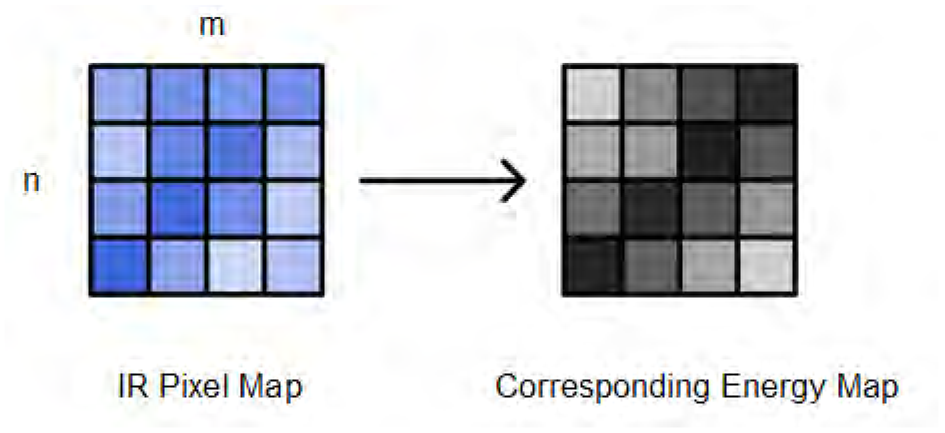


**Figure 1.3a: Example transformation from pixel map to energy map (4x4 block of pixels)**

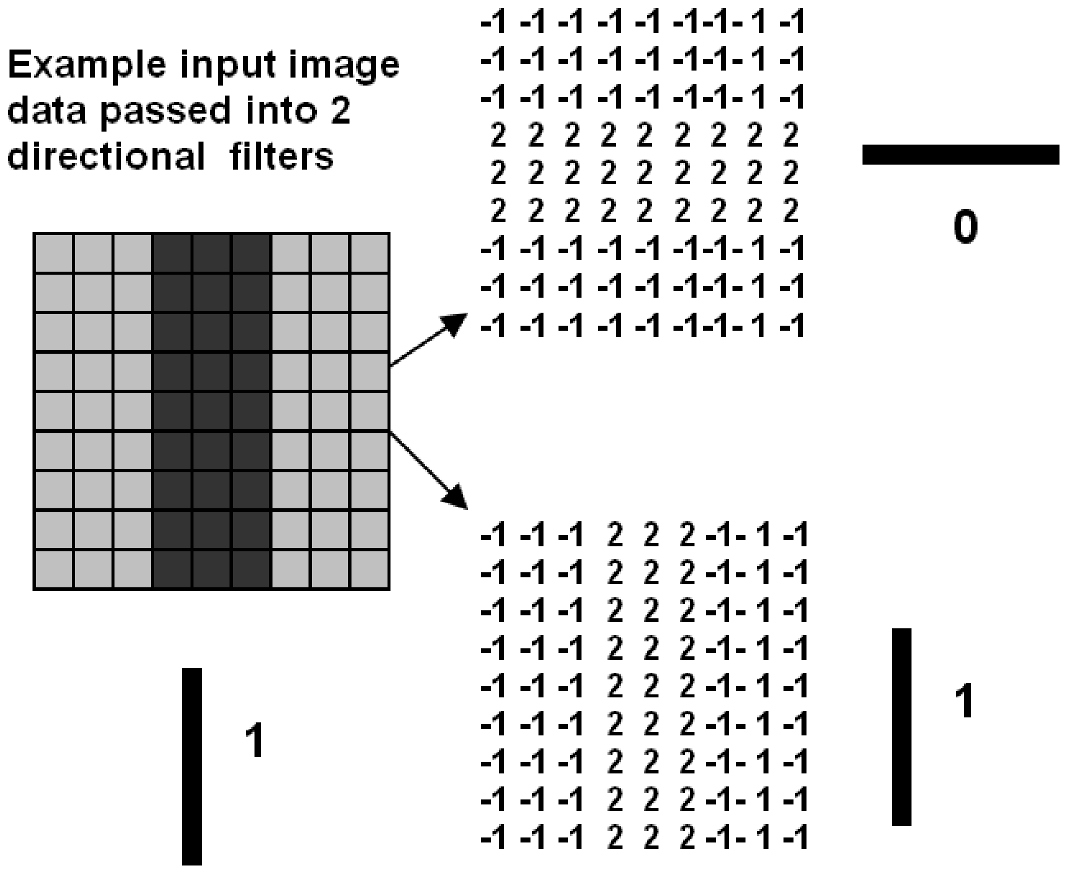


**Figure 1.3b: Demonstration of how the filtering is applied to the energy map and the corresponding bits that are passed into the template representing a piece of the iris. Note: this is not matrix multiplication but instead element by element multiplication and then summation for comparison (convolution).** [7]

The characterization ignores variations in image size by scaling the area of interest with respect to the actual size of the captured iris. This is intuitive due to the fact that everyone's iris is roughly the same size and actual dimensions are assumed and normalized. The characterization encodes only 2,048 bits or 256 bytes in the RED algorithm, like the famous algorithm written by Dr. John Daugman, which records enough data to distinguish between all the irises in the world more than billions of times over.[8] Therefore, the algorithm requires 1,024 iterations of this process to encode a full iris image into a template which can be a slow and demanding computation using a single sequential process.

Once encoded, the iris recognition system must be able to reliably match the result with a previously enrolled person. The newly encoded iris and its previously enrolled mate (or not) are matched using Hamming Distance (HD) and their associated masks (used to omit bits of data corrupted by eyelashes or noise). Hamming distance is calculated within the algorithm using the following Boolean logic formula:

$$\text{HD}=\frac{\left\|\left(codeA \oplus codeB\right) \cap maskA \cap maskB\right\|}{\left\|maskA \cap maskB\right\|} \qquad (1.1)$$

In this formula, Hamming distance is measured by first seeing how many bits between A and B agree and then modulating out those bits that may be corrupted. The mask bits for a template (codeA or codeB here) are determined simultaneously with the template. A mask bit signifies corrupted data because at the time of filtering, the template generation process could not determine a definite horizontal or vertical ridge. Sidestepping an in depth statistical analysis of Hamming distance, the matching algorithm resembles a binomial distribution of 249 degrees of freedom (N=249) with the chance of a 1 or 0 in the encoded iris being p = .500. From this understanding, the system can be fine tuned to accept varying HD values that correspond to different degrees of false acceptance rate (FAR) and false rejection rate (FRR) using the equivalent negative binomial distribution.[9] Lower Hamming Distance correlates to exponentially lower odds of a false match.

Each step of the algorithm requires varying times of execution. Depending on the number of templates that need to be matched, template matching can require the most time. Matching the information of a few people such as those in a household or a small business would take a negligible amount of time while a database of an entire nation's people would require significant time to process. The other major part of the algorithm to consume processing resources is the segmentation process. To accurately detect the location of an iris within an image, the algorithm uses extensive image processing techniques. Image processing without the assistance of a co-processor such as a video card in a general purpose system can be noticeably slow. This processing time can even be more pronounced when migrating the iris recognition system from a general purpose system with a high efficiency and high speed processor to a slower "mobile" solution.

# Chapter 2

# Background

## 2.1 Field-Programmable Gate Arrays and Programmable Logic Devices

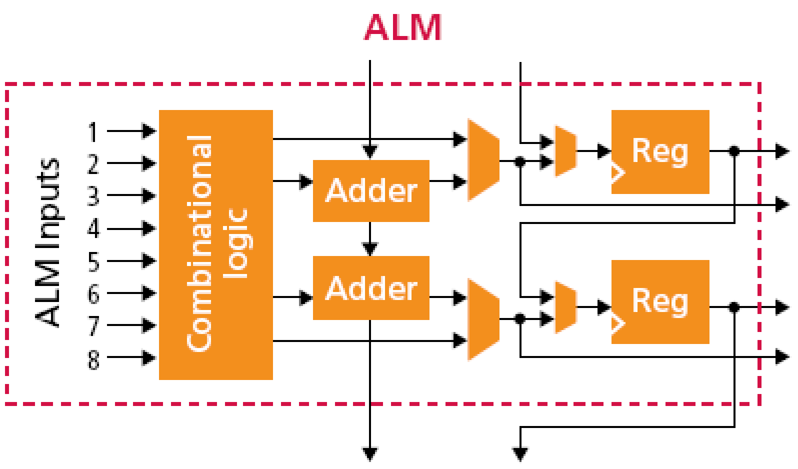


**Figure 2.1: Adaptive Logic Modules (ALMs) of the Stratix III FPGA are general purpose logic elements and are physically inside the Stratix III. When the FPGA is configured, these logic elements are interconnected by the 1000's to implement any higher level hardware function.[11]**

Designing and prototyping integrated circuits were once a challenging and extremely expensive process. In the past, all integrated circuits designed for a specific application (application specific integrated circuits or ASICs) had to be carefully designed to avoid flaws and once fabricated, could not be changed. Recently, with the development of flash densities and fabrication processes, vendors such as Xilinx and Altera have begun producing integrated circuits that have general purpose logic such as the logic seen in Figure 2.1 with programmable interconnectivity inside the chip shown in Figure 2.2 of the Stratix III architecture. These chips are called programmable logic devices (PLDs) because the connections between the internal logic elements can be programmed much like loading data into flash memory. The logic elements of these PLDs are designed to serve any function from basic 'AND' and 'OR' gates to adders and even registers and memory. Thus, an engineer has the opportunity to configure the interaction and connectivity of the logic elements long after the PLD has been fabricated, a huge step forward from traditional hardware design with ASICs.

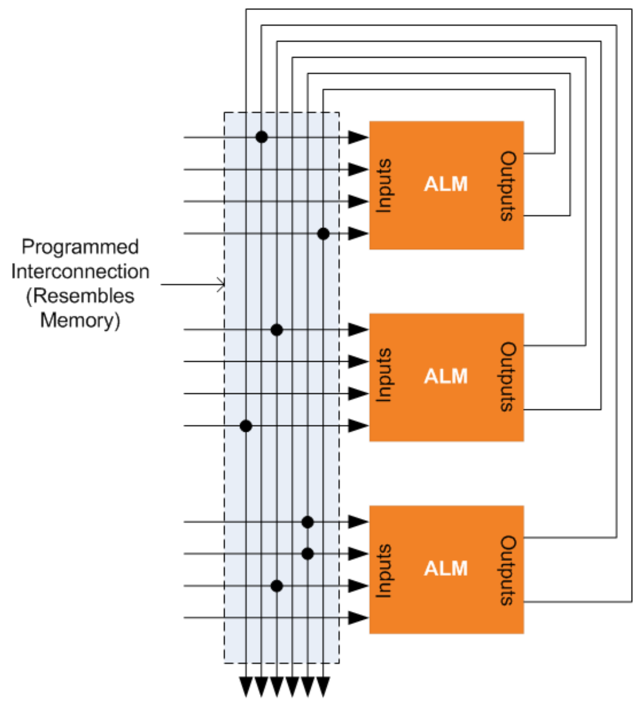


**Figure 2.2: Interconnection within the Stratix III FPGA**

Larger PLDs with more advanced internal logic elements and design are called complex programmable logic devices (CPLDs) and even more advanced programmable logic with greater applicability are called field programmable gate-arrays (FPGAs). While PLDs were simple and had limited applicability in production level designs, FPGAs are getting so advanced that they are able to compete with high level systems, an area once exclusively the realm of microprocessors, and are even being incorporated into supercomputers as co-processors. Someday, FPGAs or their future iterations may become so advanced that they fully supplant the need for ASIC design leaving only high end microprocessors and FPGAs.

This rise in FPGA capability has given the electrical engineering design community an entirely new venue to build and test complex machines. Thus, not only are FPGAs themselves a focus of research but also provide a foundation for research in hardware design. State-of-the-art FPGAs allow rapid prototyping of entire systems in minutes through intuitive design software such as Altera's Quartus II software and hardware description languages. Hardware description languages such as Verilog Hardware Description Language (Verilog HDL) and Very-high speed integrated circuit Hardware Description Language (VHDL) can describe combination properties of hardware (logic gates). Hardware description languages allow engineers to describe the transformation of data from register to register inside the FPGA, inferring logic such as adders and multiplexers along the way.

This research uses an FPGA to implement and prototype an entire system, designing hardware specifically targeted at iris recognition with the RED algorithm. Altera design software allows the use of Intellectual Property (IP) Core technology consisting of encrypted hardware description files that instantiate into the FPGA many common and advanced components specially designed for Altera devices. For example, rather than designing an entire 32 bit microprocessor and its C compiler from the ground up, we can configure the FPGA with Altera's already tried and true Nios II processor. Subsequently, we can also connect peripheral components either user-defined or pre-defined in the Altera Quartus II software as memory mapped input and output (I/O) for the Nios. The Nios II processor and the peripheral components together are called an *embedded system* and this will be investigated in more detail later. An example Nios II system is shown in Figure 2.3.

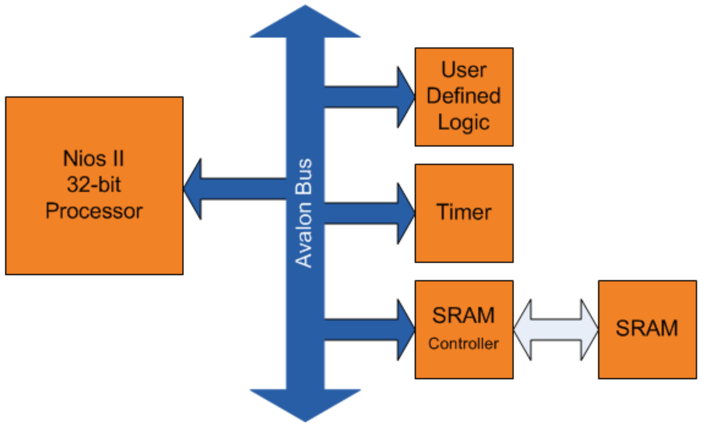


**Figure 2.3: Example Nios based embedded system**

In the above example, the Nios II is instantiated with three peripheral components that serve various functions. The SRAM Controller is a common component that would give the Nios II processor access to program and data memory. Typically, a Nios based system requires at least one memory peripheral for program and/or data memory since these are essential to any microprocessor that executes instructions. The timer can be set up to send an interrupt to the Nios II based on a predefined interval relative to the clock speed of the timer component. The timer can be configured to interrupt at 1 microsecond intervals by counting up to 50 with a 50 MHz clock at the timer clock input. A timer is useful for calculating execution times (particularly for this research) by opening up the "ctime" library to the Nios processor. Lastly, user defined logic can be incorporated into a Nios based system to perform application specific computations. An engineer could design any type of component from his/her own SRAM controller or timer to much more advanced components such as a direct memory access (DMA) controller or a VGA/HDMI interface for the Nios. All of this is possible because of the flexibility of Altera's Avalon Bus which brings all these individual components together into a single system. The Avalon Bus bridges the Nios and its peripheral components by assigning base addresses and address ranges to each component. The Nios can then address the components and talk to them by reading and writing directly into the component from the Avalon Bus.

## *2.2 Software vs. Hardware Implementation*

Functions can be implemented in either software or hardware. Typically, software is at a higher level than any hardware because, to put it simply, software is a roadmap to how existing hardware in a microprocessor works. Software is made up of instructions that engage different parts of a microprocessor's hardware in a step by step process to accomplish more complex functions than with the individual hardware components alone. For example, a hardware adder can only ‘add’ but with a set of instructions an adder could be used repeatedly to multiply or divide. However, depending on the algorithm or software in question, software execution can take short amounts of time or continue ad infinitum.

Software can implement any hardware function but hardware can only execute hard-bounded software functions. For example, software multiplication such as squaring could make use of an ‘add’ and a ‘count’ variable while hardware could use a look-up table to find the answer as shown in Figure 2.4. Software multiplication is more portable since it can make use of the same basic binary logical add as long as the registers that hold the variables are large enough to prevent overflow. Hardware on the other hand requires the input to be hard bounded to the size of the input of the look up table and would produce a hard bounded output with twice as many bits as the input. Therefore, software can operate on variable inputs and outputs such as packets while hardware implementation would be impractical. Yet, hardware is capable of tasks software can have difficulty executing. In the multiplication example, the software requires a loop to repeatedly make use of an add function while the hardware equivalent could have the answer in a single step. In systems where the hardware design can be changed to fit the application (such as an ASIC or FPGA), having a hardware squaring function compute the square could be advantageous for speed and power if the software often squares large numbers.

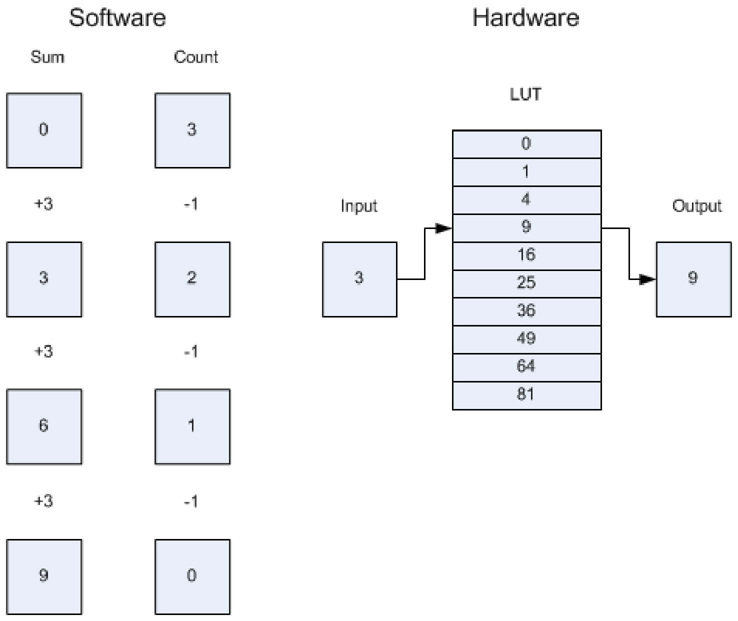


**Figure 2.4: Software vs. hardware implementation of a basic squaring function.**

One could argue that a look-up table is not actually hardware. In fact, it is a little of both worlds since the hardware for a look-up table already exists for any system that can execute software. Compilers can instantiate a look-up table in memory that the software can then reference at run-time. However, look-up tables are extensively used in hardware implementations for various reasons (some even to act as a kind of mini-software for a hardware device). In both hardware and software, a look-up table consists of a decoder that converts the “question” to an address that points to the “answer”. In the previous example with the look-up table, this is a very basic alternative to using software to repeatedly use an adder for multiplicative computations.

## 2.3 Systolic Architecture

Before designing any hardware specific to a function already implemented in software (the kind of implementation addressed in this research), we must investigate a concept in hardware architecture that is very useful when approaching a hardware design problem known as *systolic architecture*. Both software and hardware offer *many* degrees of freedom in implementation and there is rarely if ever only *one* way to implement something in either case. However, since an implementation in software is fairly easy to change and an implementation in hardware is nearly impossible to change (by nature of how hardware design is like painting a picture), we must narrow the hardware implementation design flow to a specific approach. A key difference between software and hardware is the opportunity for hardware to perform many operations in *parallel*.

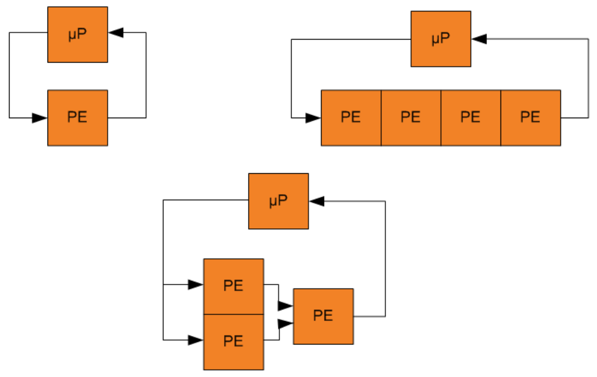


**Figure 2.5: Systolic architecture examples. The top left is a generic interaction between a microprocessor (µP) and a processing element (PE). The top right is a pipeline style systolic architecture. The bottom is a parallel input and pipelined style systolic architecture.**

Designing hardware to execute something in parallel is a vague concept. A formal approach to define what the parallel hardware seeks to accomplish can be very helpful. For this, a systolic architecture is essentially defined as an architecture that accomplishes a task with the maximum throughput given finite optimization constraints.[12] While this sounds even more vague than parallel hardware, it means that when designing parallel hardware, we want to perform the most computations we can around the greatest bottleneck in the system. Optimization constraints could be the interconnectivity of elements or the structure being pipelined or parallel. In most systems, this is usually input and output (I/O) between the microprocessor and memory or the microprocessor and its peripherals. The processor can only transfer for example, 8, 16, 32 or 64 bits at a time depending on its architecture. That means that if the microprocessor were to transmit 8 bits to a peripheral component, that component would want to perform the most computations possible given that 8-bit input. There are many different forms of systolic architecture and a few possible architectures are shown in Figure 2.5. Thus, the systolic architecture that this research will use is one that is designed around performing maximum parallel computations at the input of the peripheral device.

## 2.4 Embedded System

Building an iris recognition system using exclusively hardware would be exceptionally difficult. Naturally, since general purpose systems lack the ability to have their hardware reconfigured easily, the hardware designs cannot be placed in a general purpose system. Thus, this research will first set up a test system that migrates the algorithm code from a general purpose system such a PC running Windows to a system where application specific hardware can be implemented along with the software of the iris recognition algorithm. As mentioned before, the Nios II microprocessor can be implemented along with many peripheral components to form an embedded system on an FPGA. These peripherals can be hardware versions of functions in the algorithm C code. This paper will refer to C code as the entire category of C programming languages with C++ code or simply C++ being more specific. In other papers and documentation, C code may refer to a totally separate programming language from C++ but that is not the case here. First, we look at the overall system design using Altera's SOPC Builder shown in Figure 2.6 which is a design suite intended for putting together embedded systems based around Altera's Nios II processor.

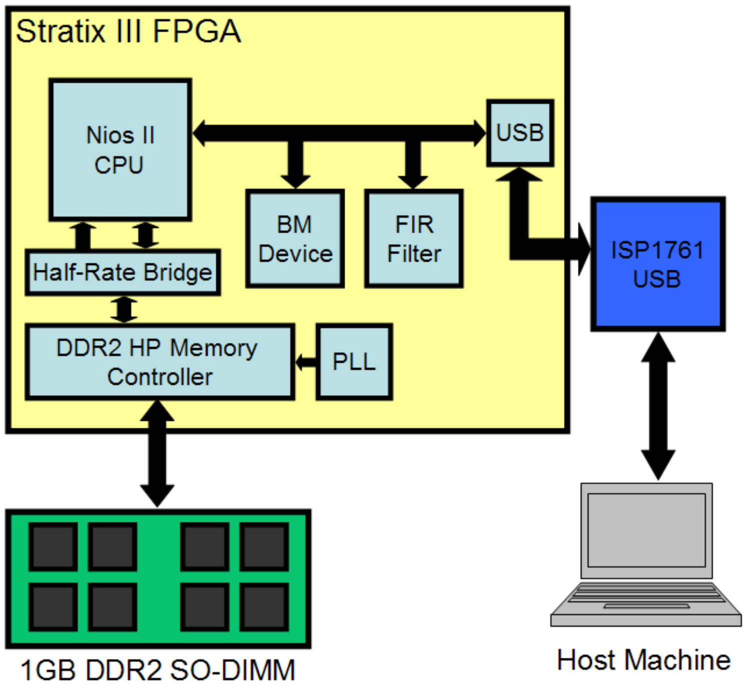
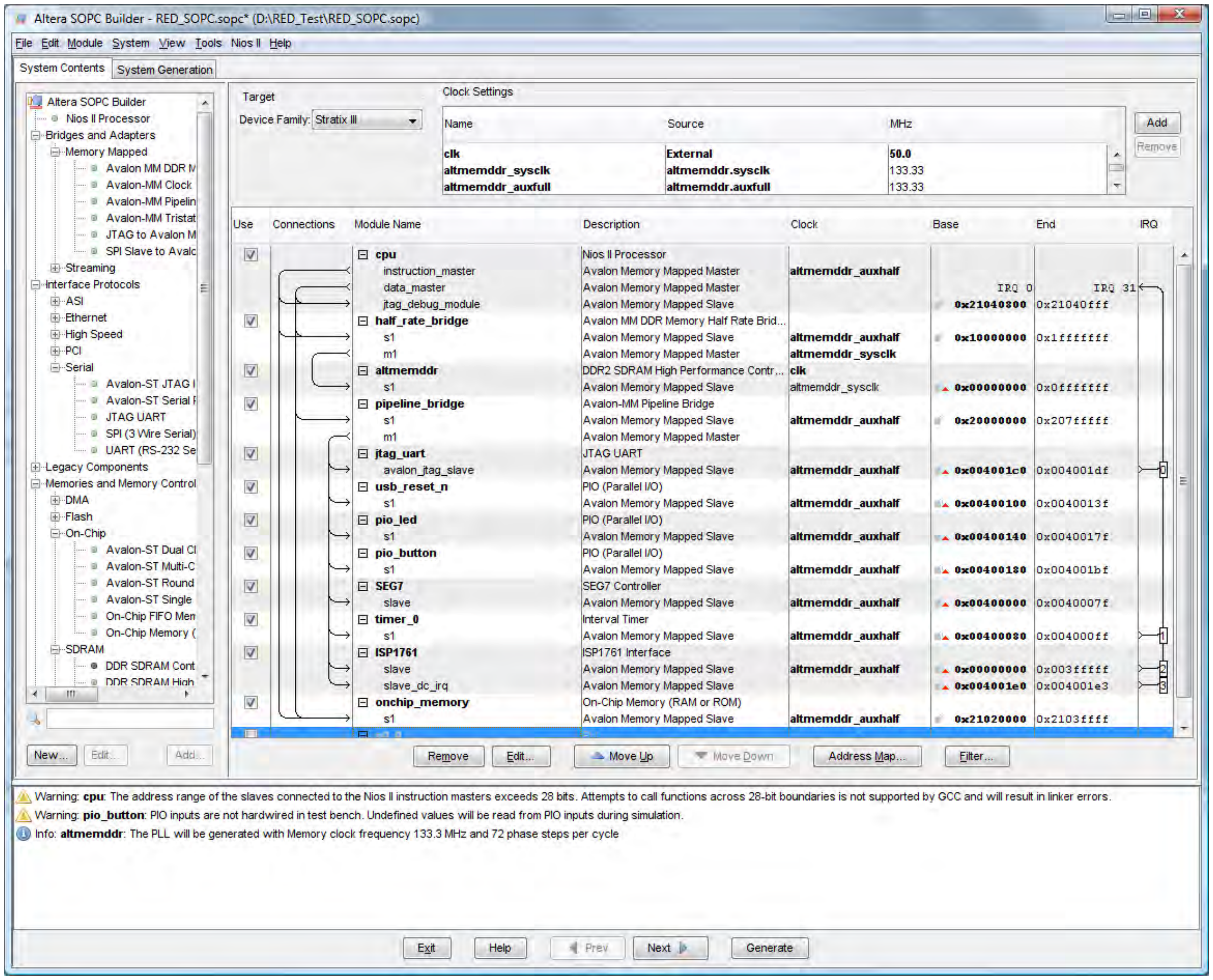


**Figure 2.6a: Embedded system concept**


Altera SOPC Builder - RED_SOPC.sopc* (D:\RED_Test\RED_SOPC.sopc)
File Edit Module System View Tools Nios II Help
System Contents
System Generation
Altera SOPC Builder
Nios II Processor
Bridges and Adapters
Memory Mapped
Streaming
Interface Protocols
Legacy Components
Memories and Memory Controllers
New... Edit... Add...
Remove Edit... Move Up Move Down Address Map... Filter...
Exit Help Prev Next Generate

Target

Device Family: Stratix III

Clock Settings

| Name | Source | MHz |
|---|---|---|
| **clk** | **External** | **50.0** |
| **altmemddr_sysclk** | **altmemddr.sysclk** | 133.33 |
| **altmemddr_auxfull** | **altmemddr.auxfull** | 133.33 |

| Use | Module Name | Description | Clock | Base | End | IRQ |
|---|---|---|---|---|---|---|
| ✓ | **cpu** | Nios II Processor | | | | |
| | instruction_master | Avalon Memory Mapped Master | **altmemddr_auxhalf** | | | |
| | data_master | Avalon Memory Mapped Master | | IRQ 0 | IRQ 31 | |
| | jtag_debug_module | Avalon Memory Mapped Slave | | **0x21040800** | 0x21040fff | |
| ✓ | **half_rate_bridge** | Avalon MM DDR Memory Half Rate Brid... | | | | |
| | s1 | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x10000000** | 0x1fffffff | |
| | m1 | Avalon Memory Mapped Master | **altmemddr_sysclk** | | | |
| ✓ | **altmemddr** | DDR2 SDRAM High Performance Contr... | **clk** | | | |
| | s1 | Avalon Memory Mapped Slave | altmemddr_sysclk | **0x00000000** | 0x0fffffff | |
| ✓ | **pipeline_bridge** | Avalon-MM Pipeline Bridge | | | | |
| | s1 | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x20000000** | 0x207fffff | |
| | m1 | Avalon Memory Mapped Master | | | | |
| ✓ | **jtag_uart** | JTAG UART | | | | |
| | avalon_jtag_slave | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x004001c0** | 0x004001df | 0 |
| ✓ | **usb_reset_n** | PIO (Parallel I/O) | | | | |
| | s1 | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x00400100** | 0x0040013f | |
| ✓ | **pio_led** | PIO (Parallel I/O) | | | | |
| | s1 | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x00400140** | 0x0040017f | |
| ✓ | **pio_button** | PIO (Parallel I/O) | | | | |
| | s1 | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x00400180** | 0x004001bf | |
| ✓ | **SEG7** | SEG7 Controller | | | | |
| | slave | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x00400000** | 0x0040007f | |
| ✓ | **timer_0** | Interval Timer | | | | |
| | s1 | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x00400080** | 0x004000ff | 1 |
| ✓ | **ISP1761** | ISP1761 Interface | | | | |
| | slave | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x00000000** | 0x003fffff | 2 |
| | slave_dc_irq | Avalon Memory Mapped Slave | | **0x004001e0** | 0x004001e3 | 3 |
| ✓ | **onchip_memory** | On-Chip Memory (RAM or ROM) | | | | |
| | s1 | Avalon Memory Mapped Slave | **altmemddr_auxhalf** | **0x21020000** | 0x2103ffff | |

Warning: **cpu**: The address range of the slaves connected to the Nios II instruction masters exceeds 28 bits. Attempts to call functions across 28-bit boundaries is not supported by GCC and will result in linker errors.
Warning: **pio_button**: PIO inputs are not hardwired in test bench. Undefined values will be read from PIO inputs during simulation.
Info: **altmemddr**: The PLL will be generated with Memory clock frequency 133.3 MHz and 72 phase steps per cycle

**Figure 2.6b: Embedded system in the SOPC Builder**

This system is implemented onto a board that allows the FPGA to be configured with hardware and programmed with software. FPGAs rarely come standalone since their integration with other systems is hardly “plug and play.” Thus, many FPGAs for research are purchased as a development board. Development boards allow engineers to design and prototype without having to put together a circuit board (PCB) and all the power supply and analog circuits to support the digital system. For this research, the TerasIC DE3 Development and Education Board was used for board level testing of the iris recognition algorithm and hardware designs. The DE3 board is shown in Figure 2.7.

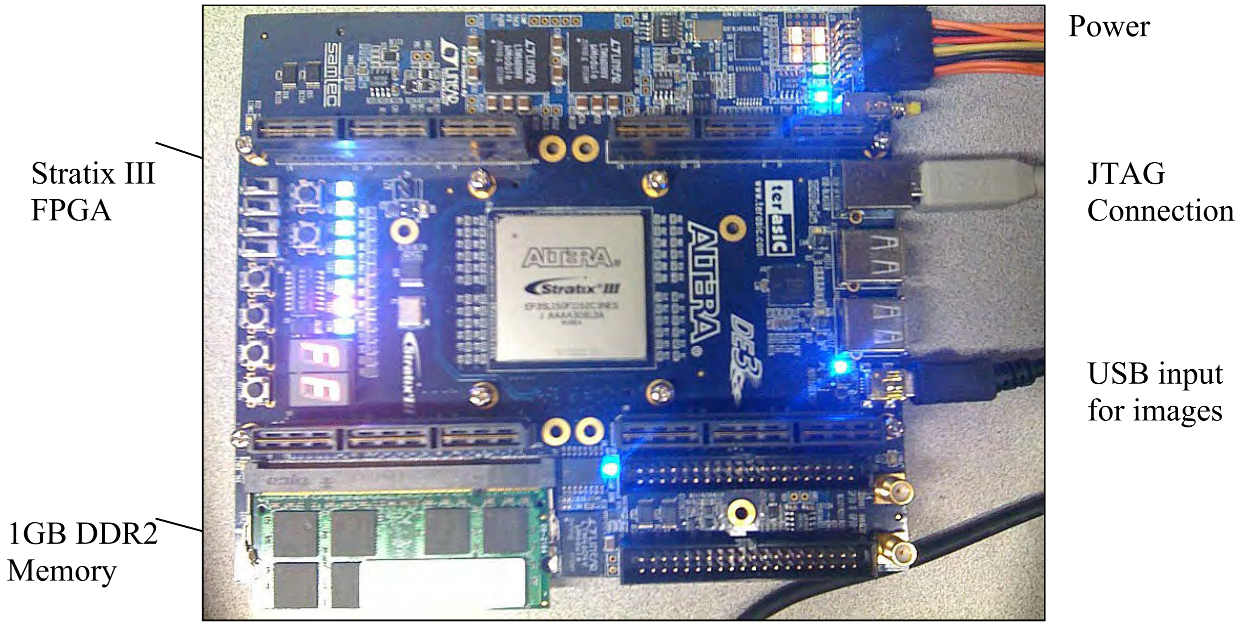


**Figure 2.7: TerasIC DE3 Development and Education Board**

The FPGA is configured with hardware and memory loaded with software through the dedicated JTAG connection which goes directly into the FPGA. This connection also serves as a debugger and allows the user to read console output from programs running on the Nios through a terminal on the JTAG host machine. Since the algorithm's C code is approximately 680kB, the program is much too big for on-chip memory (limited at approximately 260kB) and must be loaded into the only other memory on the DE3 board, the DDR2 SDRAM SO-DIMM (the green module at the bottom left of Figure 2.7). Lastly, the JTAG connection is insufficient for loading images into the DE3 board so a separate USB interface was developed and driver software for Windows written to parse images from a host machine and download them into the DE3 board for processing.

Having set up the environment for which a hardware design can be implemented and tested, we are now ready to proceed with developing hardware alternative functions to the algorithm's software functions. Testing is done by evaluating the software timing by placing start and end timers before and after the software function call respectively and evaluating over thousands of iterations. The software function call is then replaced with a hardware "function call" that interfaces the algorithm process with a hardware alternative instead. The timing for this function is evaluated in the same way the software is evaluated.

Establishing an embedded system for testing the algorithm can be a difficult feat in itself. While the Quartus II software significantly reduces the design process, understanding the dual data rate (DDR) interface and the USB protocol are critical to successful implementation. Additionally, the Nios II IDE compiles an earlier version of C++ than what the existing Windows-based console application C code is written and thus extensive knowledge of C++ is necessary to convert the C code into code that can be compiled for the Nios II processor. This included converting the Windows C++ code from using templates to macro classes for the Nios II IDE. These three challenges, C++ conversion of the original source code, DDR2 interfacing and USB interfacing are the core of this project, proving the feasibility of a chip-based iris recognition system.

### 2.4.1 Programming the RED Algorithm into the FPGA with C++

The original algorithm code was written for MATLAB and later converted to C++ for higher throughput research. The C++ code uses templates in the class declarations which tend to be a more advanced coding practice only supported by leading edge compilers such as Visual Studio by Microsoft. Classes are a container object in C++ for building more abstract data structures such as a queue, matrix or image within memory. Templates are a convenient way to implement a class by declaring the datatypes at the class instantiation. Thus, templates eliminate the need for writing a separate class for every datatype used. Since the Nios Integrated Development Environment (IDE) does not support templates in classes, the C++ templates needed to be converted to classes with *macros* and copied for every possible instantiation of the class. A macro is similar to a template in that when the code is compiled, the compiler replaces all instances of a certain keyword with the definition of that keyword such as an integer or float datatype. This requires significantly larger program footprint by more than quadrupling the amount of code. Yet, once the software is loaded into the Nios II embedded system, the algorithm sequential processes are now independent of a host machine running an operating system environment such as Microsoft Windows or Linux which can be a significant advantage for portability and speed. Because some changes had to be made to the algorithm code during the conversion for the Nios IDE compiler, the RED algorithm software is actually a close match to the original software and produces approximately the same results with less precision in some parts of the algorithm since the macro based classes sometimes used less precise datatypes in some functions. Some datatypes had to be restricted for simplicity and reduced from long to short or from double to float. These changes do not corrupt the algorithm since the less precise datatypes do not alter the distributions of various calculations such as the mean or standard deviation used throughout the software.

### 2.4.2 Integrating Advanced Memory

The DDR2 interface is an advanced memory interface used in commercial PCs available everywhere. Both the algorithm code and the space needed to store multiple working copies of images required far more space than what was available on the Stratix III FPGA. This required the use of the DE3 board’s only other memory option – the DDR2 discrete memory module on the side of the board. Communicating with DDR2 requires an interface because the Nios II addressing and data communication is much different that the addressing and data of the DDR2. This interface requires a controller to convert the language of the Nios II processor into something understood by the DDR2 module. An in-depth explanation of these interfaces is unnecessary and beyond the scope of this paper. However, interfacing the DDR2 proved exceptionally challenging and created many problems for this project.

DDR2 requires extremely complex and precise timing that caused the embedded system to be meta-stable and only meet timing within tens of picoseconds. Data transfer from register to register is stable when the register is not clocked at the same time data is switching. Narrow timing margin means the register is clocked very close to the switching of the data. This meta-stability is the result of a design flaw in the DE3 board due to improper routing of the DDR2 traces and physical connections with the Stratix III, forcing the internal FPGA system to cope with poor pin locations chosen when the board was printed. Timing stability in the picoseconds provides only extremely narrow margins for successful operation even for true ASICs and microprocessors running in excess of gigahertz. For an FPGA which has lower performance logic than these other devices and clock frequencies in the 10's and 100's of megahertz, working with stability in the picoseconds requires other parameters of operation such as temperature to possibly be as narrow as tenths of degrees. Temperature directly influences the silicon die of the FPGA and higher temperatures create more unpredictable timing of a signal through longer length routing. Since the system has functioned for brief periods of time, temperature seems to be a possible factor contributing to the successful operation of the algorithm but cannot be controlled on the DE3 board at the necessary precision. Yet, many things could cause timing instability and since it is impossible to pear inside the chip the actual cause of the system instability is unknown. Since the software is being run from the DDR2 memory, instability either makes the system very difficult to program or the program to halt halfway through due to bad or illegal instructions (software). The data memory is less susceptible to memory instability but errors in data can certainly cause issues in the processing of images. However, this was never investigated since problems with memory caused failure for both instructions (software) and data. If the Nios was able to successfully execute the entire algorithm, it can be assumed that the data was stable and reliable throughout the entire execution.

### 2.4.3 Building Interfaces with the DE3 Board

Lastly, the other challenging development for the embedded system was the USB interface. The embedded system does not have a camera dedicated to capturing images and placing them into memory for processing. Thus, some means of getting images into the DE3 board was necessary. The DE3 board has limited interfacing, with only USB host and device ports readily available. Again, the USB protocol is exceptionally complicated and beyond the scope of this paper. To bring images into the embedded system, driver software had to be written for the Nios II microprocessor to talk to the USB port as well as driver software for a host system (Microsoft Windows Vista based) to detect the DE3 board as a slave device and transmit images. Using the Microsoft Developer Network's WinDDK, driver software was written to parse a bitmap image of any format into a matrix of 8-bit values, open a connection with the DE3 board and transmit the image in a series of 32 byte packets. Transmission of a typical 640x480 VGA image required a little over 9600 packets (1 byte in each packet acts as a header) and took only about 1 second. Due to the simplicity of the driver on the Nios II side, another image cannot be pushed to the DE3 board until the algorithm is fully executed.

For demonstration purposes, a human interface was developed so the user can get feedback of the system operation. The DE3 board is equipped with LEDs that can be red, green or blue or any combination of those colors for each. These LEDs are initially all in the unlit state. When a new unrecognized eye is loaded onto the DE3 board, the system sets one of the LEDs red (up to 8 possible unique eyes). When an eye that is recognized as a previously loaded eye, it will light the corresponding LED to that eye green momentarily and back to red. Images of unrecognized eyes will instead light a new LED red.

## *2.5 Bringing Everything Together*

The fully implemented software and interfaces now allows for the testing of hardware devices integrated into the embedded system. These hardware devices can be parallel implementations and are meant to ease the processing burden of the Nios II processor by pushing the data to a hardware based function rather than using the original software functions. The embedded system allows for testing and evaluation of the hardware based functions and comparisons to be drawn because both implementations can be compared on the same system.

# Chapter 3

# Binary Morphology

Computer systems which this paper will refer to as general purpose systems are electronic machines that execute instructions known as software. Many computer systems have operating systems that run in the background along with many other concurrent processes or programs that can slow the general purpose system down. Iris recognition algorithms are usually written in C code and compiled into an application that the operating system runs among other programs. For the purpose of optimization, the iris recognition software would ideally be isolated from the parent system and concurrent processes. This could allow a processor to focus its full throughput on the execution of the algorithm software. Furthermore, general purpose systems are not especially mobile (with the exception of very lightweight laptop computers) and thus may not serve every venue where iris recognition systems could be deployed. Thus, a hardware implementation of an iris recognition system is especially interesting as it could be exceptionally faster than its general purpose counterpart while also being small enough to be part of a digital camera or camera phone with sufficient resolution to detect fine iris features.

## *3.1 Iris Recognition Segmentation*

Segmentation searches for the iris within an image of an eye. For template generation to be successful, the segmentation process must properly extract the iris from the image. As humans, we have an exceptionally powerful image processor working for us as a result of millions of years of development that allows our brains to effortlessly segment out objects in our field of view. For a computer, segmenting objects from an image is similar to trying to inspect a photograph by looking through a coffee straw. Computers lack the ability to see the "big picture" and must use lengthy and complex image processing functions to break down the image.

The iris is bounded by the pupil at the inner radius and the limbic boundary at the outer radius. Finding these boundaries begins with finding the pupil since it is assumed to be the easiest part of the image to segment with the assumption that it is simply a large black circle somewhere in the image. To identify the pupil, the original image undergoes contrast limited histogram equalization (CLAHE) to adjust contrast and better differentiate the difference between minimum and maximum values across the image before thresholding all the values. All values above the threshold (typically the mean plus one standard deviation of all pixels in the image) are assigned a value of '0' and all values below are assigned a '1.' This leaves only the pupil and other very dark objects such as some eyelashes and perhaps some of the eyebrow. However, due to the nature of the cornea being at its apex right above the pupil, there are often glare artifacts directly over the pupil and cause the pupil to be obfuscated and irregular. To correct for artifacts and find the existence of objects in the image, the segmentation process uses a series of binary morphology functions to dilate, erode and assign values to the objects based on area and perimeter.

Once the pupil is detected by comparing the different objects in the image, the pupil's center is used as a starting point back in the original image for finding the limbic boundary. Local statistics compute the kurtosis of 3 by 3 windows all around the image. The algorithm then picks values radially from around the area of the limbic boundary and accumulates the values. Segmentation then steps left and right of the pupil center recalculating the sum of the local kurtosis. The smallest value is most likely the true location of the limbic boundary and the iris can be located and unwrapped for template generation.

## 3.2 Binary Morphology

Binary morphology functions operate on the image after it has been thresholded in order to find the existence of relevant objects such as the pupil. When an image is thresholded, the pixel values in the image are reduced to a single bit for logical operation. Approximately half of the algorithm's segmentation execution time is consumed by binary morphology functions. The most difficult of these functions are the dilation and erosion functions which are slow even when pre-processing the image to find objects for dilation and erosion to operate on. Figure 3.1 illustrates a single step dilation and erosion and how the resulting object is roughly the same shape but without the artifact in the center.

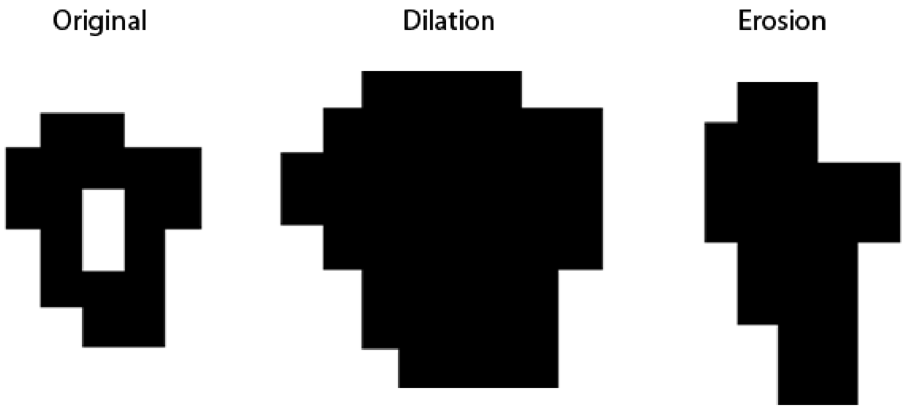


**Figure 3.1: Binary morphology demonstration.**
**Left: image before dilation and erosion. Middle: one step of dilation. Right: one step of erosion. Note that the artifact (hole) was filled in the final image but the general shape is retained.**

Logically, dilation and erosion are the same function. Dilation looks at all the '1' values in the image and places a '1' in every immediately adjacent pixel above, below, left and right. Similarly, erosion places a '0' value pixel adjacent to every '0' pixel in the image. These functions must scan all the relevant pixels and make non-linear adjustments around the image. Therefore, binary morphology even when ideally optimized is a growth function. However, with many objects and many pixels in the image, binary morphology can become increasingly messy and complex, consuming much of the general purpose processor's attention.

## 3.3 Methodology

This paper will take the binary morphology part of the RED algorithm and compare a software and hardware implementation of the dilation and erosion functions in order to demonstrate the effectiveness of a hardware equivalent function. Hardware based segmentation could be faster, slower or more difficult depending on the function being implemented. These functions are directly comparable to other standard software based image processing functions. Also, there are many opportunities for alternate implementation in the algorithm: implementation of the binary morphology dilation and erosion functions are just a couple of many possible functions that can be implemented. In order to properly implement the hardware equivalent, we investigate the software implementation of dilation and erosion in order to understand how the logical function should behave in hardware.

To properly evaluate the conversion from software to hardware, a test system needs to be developed to execute the entire algorithm in software on the target device. The FPGA is programmed with a test system that can execute C code and the details of this implementation are beyond the scope of this paper. The FPGA system is different than a general purpose system because the processor on the FPGA runs only the algorithm code and no other processes but at a slower clock. Additionally, the speed of the FPGA system is generally slower than a general purpose system. This FPGA system is fed an image of an eye where the segmentation, template generation and template matching processes begin just as they would on a general purpose system. Note that the programming of an FPGA is different from the programming of software into a computer. This programming defines the interconnection of basic logic elements inside the FPGA so that the desired logic (processor, memory controller and test hardware device) is realized. However, unlike the general purpose system, the FPGA system has the ability to call on a specially designed peripheral hardware component inside the FPGA to process data and can read the result back almost seamlessly.

Upon converting the function from software to hardware, the software based function call is followed by a new function that calls on the hardware device for data processing. The functions under test are isolated in the algorithm by placing reference points before and after the function call that records start and end times. The output of the software function can be used to assert the valid output of the hardware based function and the execution times can be compared.

## 3.4 Software implementation

The software version of the algorithm preprocesses the image by scanning the image for objects that have an area that is at least the minimum area for the pupil. These objects are individually passed into the dilate and erosion functions for processing in an effort to avoid processing the entire image in a brute force function. The software function calls appear in the algorithm as shown in Tables 3.1 and 3.2.

**Table 3.1: Binary morphology function calls from FindPupil function.**

```
...

label=ThresholdUINT16Image(im,
    (UINT16)threshval);

areas=label->BWLabel(MIN_PUPIL_AREA);

bwobjs=ConvertLabelImageToObjects(label,
    areas);

for (i=0;i<bwobjs->Nobj;i++) {
            BWDilateInPlace(bwobjs->obj[i]->
        data,se1); // Dilate
    BWErodeInPlace(bwobjs->obj[i]->
        data,se2); // Erode

...

}

...
```

**Table 3.2: BWDilateInPlace function definition. Note that the return of the image is inverted**

```
void BWDilateInPlace(Image *in, Image *se)
{

        ...


   for (i=1; i<in->ActualRows-1; i++)
      for (j=1;j<in->ActualCols-1;j++)
         if (in->matrix[i][j])
            {
              in1->
               matrix[i+offset][j+offset]=1;
              if((in->matrix[i-1][j]==0)||
               (in->matrix[i+1][j]==0)||
               (in->matrix[i][j-1]==0)||
               (in->matrix[i][j+1]==0))
                  {
                   ioffset=i+offset;
                   joffset=j+offset;
                   for(ii=0;ii<numones;ii++)
{
                   in1->matrix[ioffset+
                   rowindex[ii]]
                   [joffset+colindex[ii]]=1;

                 };
            };

        ...
```

The software implementation uses three FOR loops to step through the objects in the image and perform dilate/erode processing. Each function call steps through all relevant pixels and assigns the appropriate bit based on adjacent black and white values. Additionally, each function must be called 15 times for a total of 15 dilates and 15 erodes to fully process each object in the image. With a large image that has many objects covering enough area to possibly be the pupil, the functions can take a lot of time to process. Next, we will investigate a hardware alternative to these functions.

The business portion of the software implementation is the looping within each binary morphology functions. These loops use basic logical ‘or’ of the neighboring pixel above, below, left and right to determine each bit assignment. These loops are also the most time consuming since they are the highest order nest in the function calls. A hardware implementation could reduce these loops to a  growth function.

The software implementation takes into account edge effects by padding the outside edges of the image with zeros. The borders of the image complicate the dilation and erosion functions since pixels on the far outside of image will reference pixels not defined within the rows and columns of the image. Instead, these areas are filled with null values so that they do not contribute to the dilation or erosion of objects very close to the borders.

Another advantage with hardware is that a hardware implementation takes advantage of replication to speed up processing power. The preprocessing of the image to find objects before dilation and erosion is not necessary to increase speed. Instead, the initial object detection of the algorithm will be moved after dilating and eroding all objects in the image simultaneously. Theoretically, this reordering of operations should not affect the output.

## *3.5 Hardware Implementation (VHDL)*

Hardware design is inferred similar to the way software is coded using special programming languages such as Verilog Hardware Description Language (Verilog HDL) or Very high speed integrated circuit Hardware Description Language (VHDL). Designs are compiled using a synthesizer and fitter that build a netlist describing how the fundamental logic elements within the FPGA are wired together.

The primary function to be implemented is the logical 'or' function that we identified from the software function loops. These 'or' functions are pixel oriented and directly translate to an elementary unit in hardware that we'll call a BM_Element (BM for Binary Morphology) which represents the pixel being operated on. The value stored within this BM_Element is driven by similar hardware elements holding values for pixels adjacent to the pixel stored in this BM_Element. Thus, these BM_Elements are easiest to imagine as though they were arranged in the 'shape' of an m x n image. The logic for these BM_Elements also takes into account both the dilation and erosion since they are complimentary binary logic functions.

The coding procedure for this hardware is to instantiate a full dilation and erosion hardware entity called BM_Dilate_Erode that will receive the pixel data from the processor one pixel at a time (with a width of only a single bit for '1' or '0') and transmit the resulting pixel data back to the processor one pixel at a time. BM_Dilate_Erode will act as a three-state state machine that receives the pixel data, processes the data and then sends the data back. Within the overarching BM_Dilate_Erode instantiation are the individual BM_Element blocks and an interconnect fabric that allows for pixel data to be loaded into each BM_Element and for each BM_Element exchange relevant pixel data among each other. Data enters the BM_Dilate_Erode device as a shift register as shown in Figure 3.2.

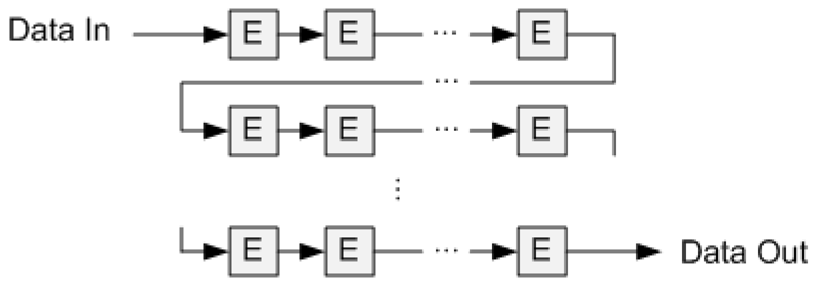


**Figure 3.2: Binary morphology data input train. The input shifts the data into the elements starting at the first and ending at the last BM_Element represented by the labeled "E" blocks. The processor shifts the last pixel of the thresholded image in first and when processing is complete the data is shifted out last pixel first.**

The interconnect fabric within the BM_Dilate_Erode machine is the most challenging part of the device. Unlike the software implementation, this fabric cannot account for edge effects using the technique of padding (which places null values around the edges). Instead, it circularly connects the top and bottom, left and right elements together so that these connections are not undefined. The effects of these circular connections are negligible since a segmentable pupil will be located somewhere within the image borders and ideally not partially cut off at the edge of the image. The interconnect fabric is actually a long string of "nodes" numbered from zero to the total number of BM_Elements within the device. These nodes represent the connect points between different BM_Elements and each BM_Element references adjacent elements by their numbered connection in this fabric. This is illustrated in Figure 3.3.

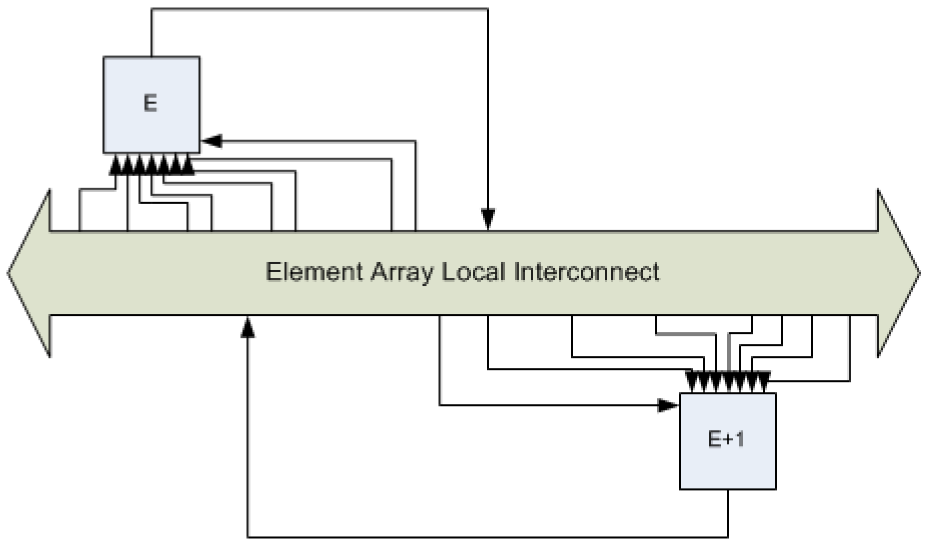


**Figure 3.3: The binary morphology local interconnect fabric is a series of nodes labeled from zero to the total number of BM_Elements in the BM_Dilate_Erode device. Each BM_Element inputs the adjacent pixel values held by neighboring BM_Elements by referencing neighboring nodes in the fabric and outputs the pixel value held within the BM_Element to its associated interconnect fabric node.**

Each BM_Element computes the dilation and erosion using only a few lines of code as shown in Table 3.3 and produces the logic blocks seen in Figure 3.4. Like a function call in software, the BM_Element encapsulates recurring code into a single logic block with finite inputs and outputs. These inputs reference the neighboring four nodes and necessary control signals. It outputs its value back into the local interconnect fabric node allotted for that BM_Element. This is accomplished using the code shown in Table 3.4. The hardware operation is demonstrated in Figure 3.5.

**Table 3.3: BM_Element functional logic**

```
process(aload, adatain, clk)
begin
...
       if (aload = '1') then
              adataout <= pixel;
              pixel_step <= adatain;
       else
              if (rising_edge(clk)) then
                     if(clk_enable = '1') then
                            if(operation = '0') then      -- Dilate
                                   pixel_step <=topmid or midleft or midright or
                                   bottommid;
                            else
                                   pixel_step <= not(topmid or midleft or midright or
                                   bottommid);
                            end if;
              ...
process(clk)
begin
       if(rising_edge(clk)) then
              if(operation = '0') then
                     output <= pixel;
              else
                     output <= not(pixel);
              end if;
       end if;
end process;
```

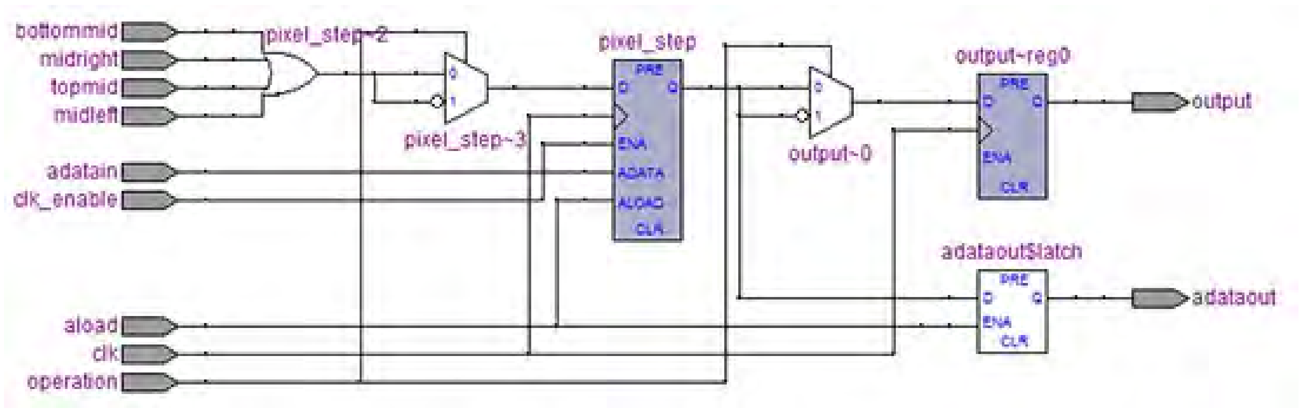


**Figure 3.4: Internal logic of one BM_Element block inferred from the code seen in Table 3.3. The device takes in 4 inputs from the local interconnect and 'or's the logic together. This value is then inverted depending on the operation (dilate or erode) and latched at the pixel_step flip-flop. There is also control logic to shift the value in this BM_Element on to the next BM_Element and input the value from the previous BM_Element in the chain as demonstrated in Figure 3.2.**

**Table 3.4: BM_Element instantiation and interconnect logic**

```
signal local_interconnect : std_logic_vector(ROWS*COLS downto 0);

...

BM_rows: for j in 0 to ROWS - 1 generate
begin
        BM_cols: for i in 0 to COLS - 1 generate
        begin
                U1: BM_Element
                port map(
                        clk => avs_s1_clk,
                        clk_enable => clk_e,
                        operation => operation,
                        aload => shift_load,
                        adatain => local_data(j*COLS + i),
                adataout => local_data(j*COLS + i + 1),
                        topmid => local_interconnect(((j + ROWS - 1) mod ROWS)*COLS + ((i +
                        COLS) mod COLS) ),
                        midleft => local_interconnect((j*COLS)+((i+(COLS-1)) mod COLS)),
                        midright => local_interconnect((j*COLS)+((i+(COLS+1)) mod COLS)),
                        bottommid => local_interconnect(((j + ROWS + 1) mod ROWS)*COLS + ((i
                        + COLS) mod COLS) ),
                        output => local_interconnect( j*COLS + i)
                );
        end generate;

end generate;
```

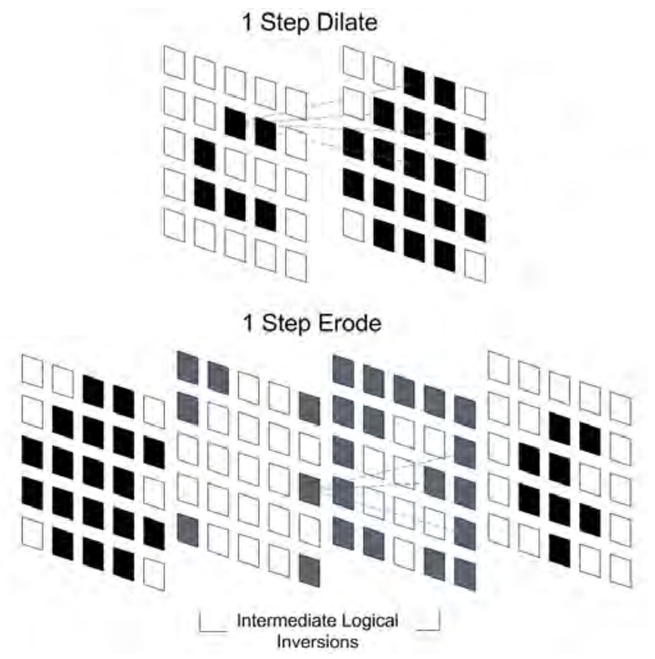


**Figure 3.5: Dilation and erosion in hardware. For dilation, the hardware performs logic 'OR' on all pixel's neighboring pixels in parallel and if any of them are black the particular pixel in question becomes black. For erosion, the inputs and outputs are inverted and thus performs the same logical 'OR' but on white pixels.**

Once the entire design is put together, it must be incorporated into the overall FPGA system so that the performance of the hardware design can be evaluated relative to the original software design. In this system, the design is included as a peripheral component to the microprocessor in the system and the compiler provides the microprocessor with a pointer to the input of the BM_Dilate_Erode device. The algorithm software code is rewritten to send the entire image for dilation and erosion and the result is read back for object detection. This peripheral totally eliminates the dilation and erosion binary morphology functions from the algorithm in exchange for simple copy functions that move data from DDR2 SDRAM into the peripheral device and back out.

# Chapter 4

# FIR Filtering

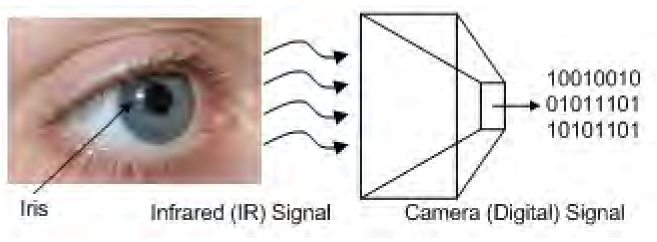


**Figure 4.1: Capturing the image of an eye and storing the result as pixel data**

Iris recognition biometrics measure the human eye as though it were a digital signal conveying a message of identity. The signal is transmitted from the iris, the eye's colored region around the black pupil, visually via infrared radiation and captured using a camera, storing the signal as digital ones and zeroes (picture elements or pixel data) as seen in Figure 4.1. Since pixel data is nearly meaningless to a computer, digital signal processing must be performed to extract meaningful information from ones and zeroes. Biometric algorithms depend on digital filtering, a kind of digital signal processing, to transform simple pixel data into meaningful bits used to match a person's eye with a stored digital counterpart. In current recognition systems, a general purpose processor follows instructions, also known as software, to perform the digital filtering a single step at a time. At high clock frequencies, software driven filtering is not as much a troubling issue; however, when migrating this or any digital filtering to hardware such as a field programmable gate array (FPGA), slower clock frequencies manifest a challenge that otherwise may not be a problem. FPGAs can mitigate having a slow clock by maximizing efficiency in highly application specific hardware design. For an iris recognition system, specially tailored parallel digital filters can help.

## *4.1 Filtering in Hardware*

Parallel filtering can take on many different forms and an engineer may choose the form of parallelization that best fits the needs of the application. In any parallel digital operation, memory is a key factor in design since it is often either a bottleneck or an FPGA device resource hog. Heavy parallelization in hardware will invariably require more memory to support simultaneous reading and writing. Iris recognition algorithms typically use nine by nine (9x9) filters which can rapidly consume an FPGA's available RAM resources needed to keep track of many parallel intermediate steps. Therefore, the form of parallelization used in this design seeks to maximize the amount of parallel computations while keeping the memory needed for intermediate computations optimized and at an absolute minimum.

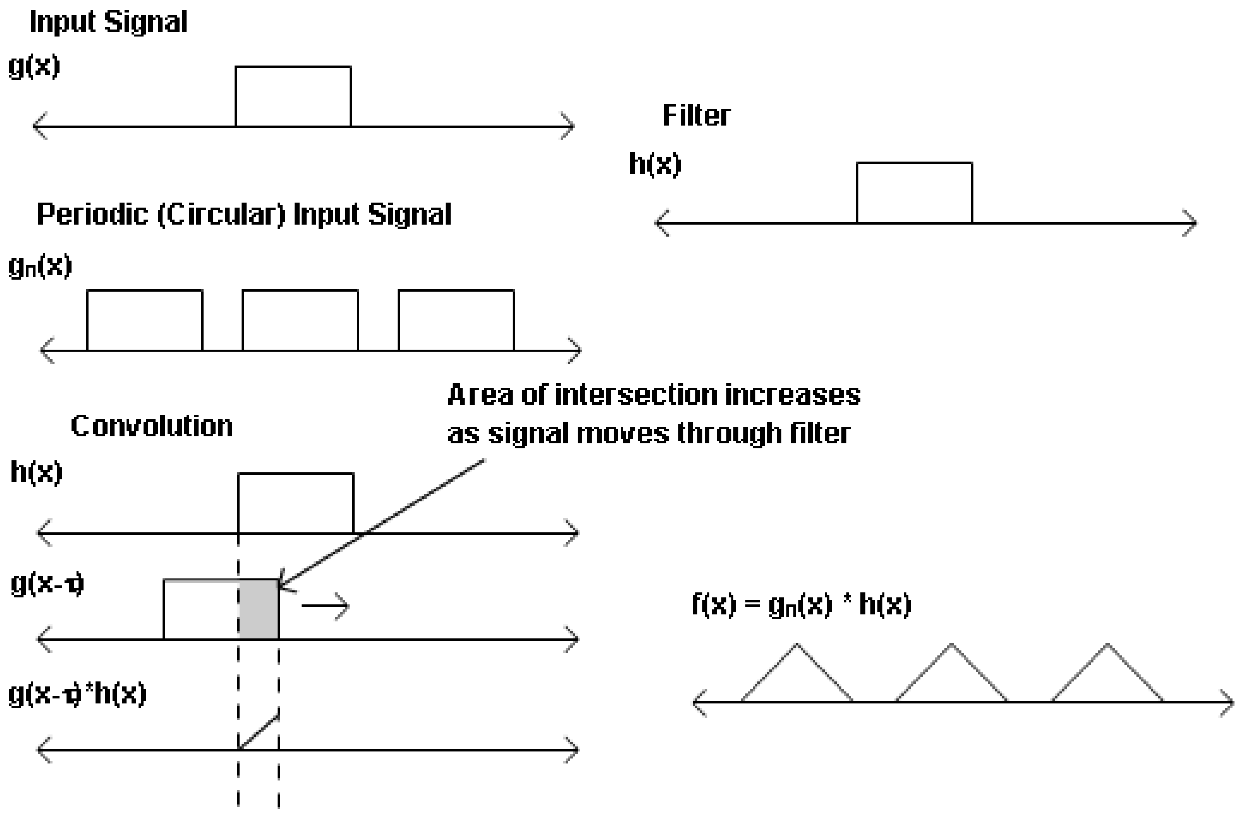

Input Signal
g(x)
Filter
h(x)
Periodic (Circular) Input Signal
gn(x)
Convolution
Area of intersection increases as signal moves through filter
h(x)
g(x-τ)
g(x-τ)*h(x)
f(x) = gn(x) * h(x)

**Figure 4.2: Circular convolution in one dimension**

Before filtering, the inner and outer boundaries of the iris were identified. The Ridge Energy Direction (RED) Algorithm divides the iris into at least 16 concentric circles and 128 radial lines. The iris is then "unwrapped" by picking off values from the original image at each intersection of radial line and concentric circle and recording them value in a 2-D matrix of exclusively iris pixels. This process is likened to converting the iris from polar coordinates to Cartesian coordinates.

The RED Algorithm uses finite impulse response (FIR) filtering to evaluate the information content of an iris. FIR filtering is the type of filtering done by a digital system in the time domain within a finite set of coefficients. The algorithm uses two 9 by 9 (9x9) filters to accentuate the vertical and horizontal features separately. A 9x9 filter will be able to emphasize the existence of ridges as wide as the filter dimension. This assumes that most ridges are features small enough to be detected by the 9x9 filter. Features that are too large will essentially be ignored by the filters and may result in a mask bit modulating out that part of the template if neither filter can determine vertical or horizontal ridges. Furthermore, these filters are unaffected by the original resolution of the image since the density of pixels (number of pixels per feature) are determined directly by the choice number of concentric annuli and radial lines. Higher or lower resolution segmented irises will always result in the same density of pixels per feature since the number of points chosen to transform from polar coordinates in the original image to Cartesian is constant. The output of each filter is compared and for each pixel, a '1' is assigned for strong vertical content or a '0' for strong horizontal content. These bits are concatenated to typically form a 2048 bit vector unique to the "iris signal" that conveyed the identifiable information.

The RED iris recognition algorithm uses a straightforward circular convolution digital filter similar to Figure 4.2 to generate the matchable iris template, a set of bits that meaningfully represents a person's iris. The energy data passed from the iris segmentation process, the process that seeks and extracts the iris from an image of a person's eye, is organized into a rectangular array that is made periodic in both the vertical and horizontal dimensions to account for edge effects. The filter passes over this periodic array taking in 81 values at a time and computing the result for the filter, storing the result in a memory location that corresponds to the centroid of the filter. The digital filter continues to take in values in this manner, stepping right column by column and down row by row until the filtering is complete as shown in Figure 4.3. Finally, the template is generated by comparing the results of two different digital filters (horizontal and vertical) and writing a single bit that represents the filter with the highest output at the equivalent location.

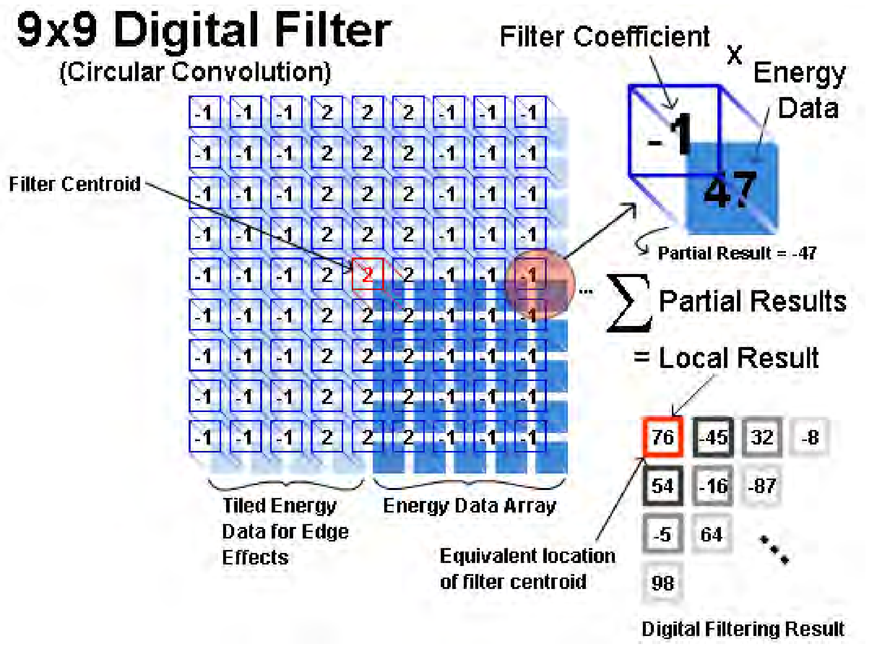


**Figure 4.3: 9x9 digital filter computing the circular convolution of the top left portion of hypothetical energy data. In this instance, each coefficient of the filter is multiplied by the corresponding energy data within the scope of the filter (filter kernel) where some of the data is repeated from the opposing sides. These filter coefficient and energy data products make up a partial result, the sums of which generate a local result corresponding to the centroid of the filter.**

## 4.2 Problem Addressed

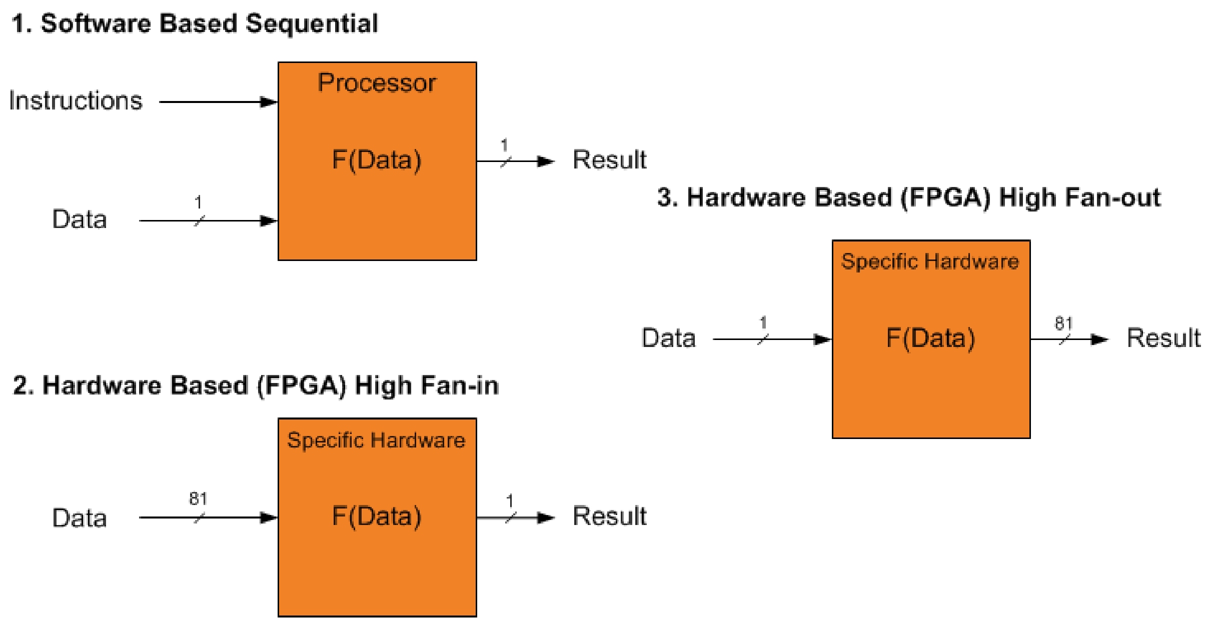


**Figure 4.4: Illustration of different possible digital filtering methods.**

There are three outstanding methods to compute a digital filter as shown in Figure 4.4. The first method uses a general purpose processor to compute the digital filter via software, executing a single instruction at a time. This approach is acceptable if the digital filter is small; however, as the digital filter dimension increases, the computational load rises exponentially. The second approach is to develop hardware specifically designed to compute the digital filter but by first loading all the data into an array of computing elements. By loading the data first (buffering the data into the hardware), this approach is essentially identical to the sequential general purpose processor method due to overhead. Only partial convolutions can be computed until the entire kernel is filled. The third and final approach is to redefine the convolution as a parallel process, allowing the hardware to take in a single byte of data and compute the maximum influence that data would have on the final results. The method is a truly parallel process and minimizes the need for overhead and buffering that other sequential processes use. Throughout this paper, the parallel method of approach will be investigated and compared to sequential methods.

Parallelizing the RED algorithm filtering can be done simply by doing the obvious – calculating the result of both filters at the same time and computing the single bit representation in the template simultaneously. Unfortunately, while this straightforward approach appears to cut out a large amount of intermediate memory it in fact suffers from a very serious memory bottleneck. The energy data to compute a single convolution result must first be fully loaded into a hardware representation of each filter. Assuming the filters could compute the final template bit in a single clock cycle, each filter must read 81 different 8-bit values of energy data before any filtering can be done. Therefore, a straightforward approach to convolution based digital filtering in hardware will suffer due to the overhead associated with waiting for data to be localized near the computing elements. In this case, since the computing elements are idle until the 81 data values are loaded, the process is fundamentally sequential much like the general purpose processor. Consequently, a straightforward convolution in hardware is inherently not parallel with a single bank of memory holding all the energy data.

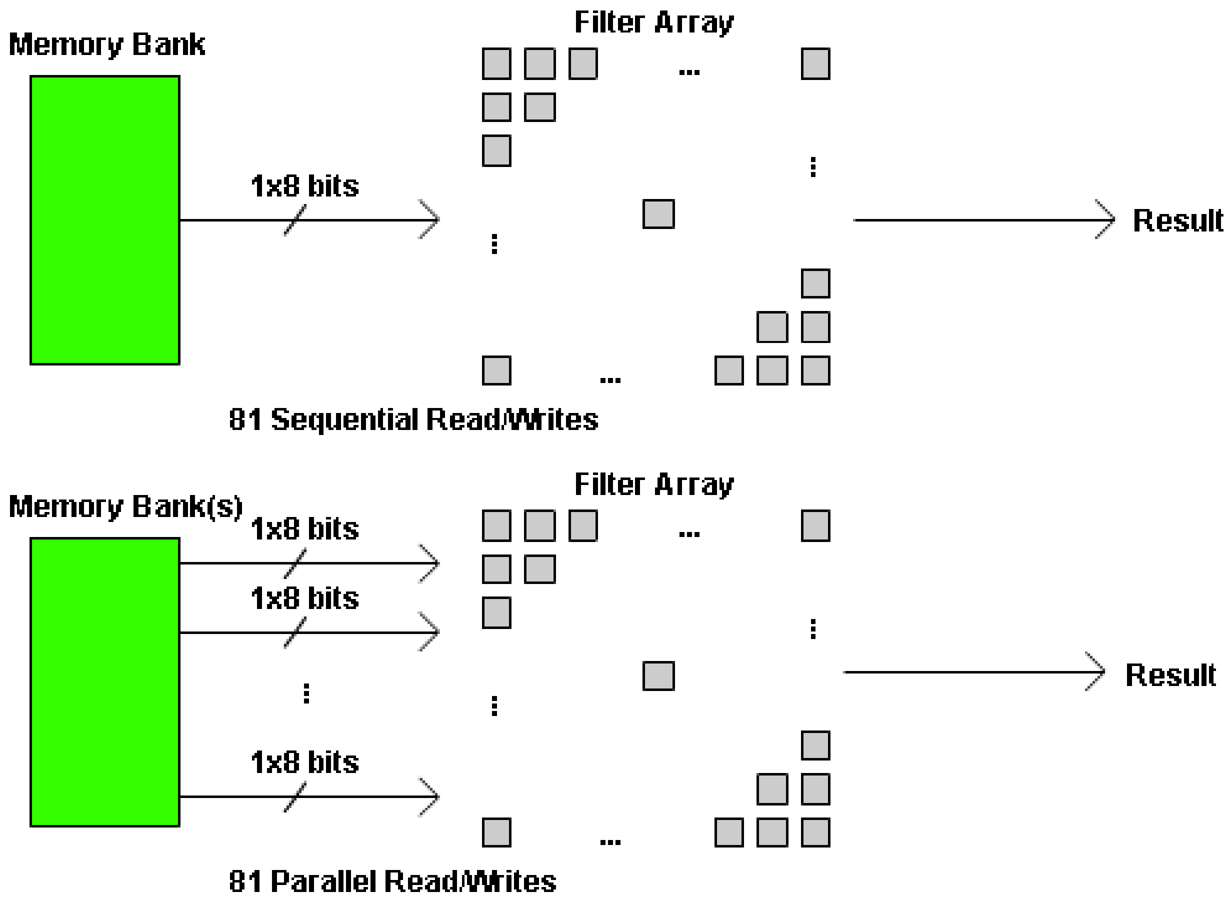


**Figure 4.5: Possible considerations for sequential hardware method**

To parallelize the digital filter convolution, the memory itself must be parallelized. A possible but impractical option to parallelize the straightforward digital filter convolution would be to create an 81 ported block of memory or 81 parallel blocks of memory that could supply all energy data to both filters in a single step as seen in Figure 4.5. Either case would exceed the memory resources of even the most massive FPGAs while also potentially creating interconnect issues. Furthermore, even if the memory issue was inconsequential to the straightforward circular convolution, there is no way around the overhead associated with needing at least 81 energy data values and at least 9 more properly arranged to step in either the horizontal or vertical directions.

## 4.3 Solution

There is an alternative way to accomplish the circular convolution that also offers greater opportunity for parallelization while also slightly more memory friendly in reversing the digital filter. Convolution is a commutative mathematical operation of the form in equations 4.1 and 4.2

$$f(x) = g(x) * h(x) \quad \text{or} \quad f(x) = h(x) * g(x) \qquad (4.1)$$

where

$$y(n) = h(n) * x(n) = \sum_{m} h(n) \cdot x(n-m) \qquad (4.2)$$

where the resulting function “f” is either function “g” convolved with “h” or “h” convolved with “g.” In terms of digital filtering, this is the same as either convolving the filter with the input or convolving the input with the filter. This concept is critical in modifying the digital filtering algorithm so that it can more easily be implemented in hardware. Typically, the filter is convolved “overtop” the input data but for parallelizing purposes the input data will be convolved over the centroid of the filter. The commutative property is important in developing the necessary decomposed algorithm for parallel computations. The decomposed algorithm is a function of the accumulated convolution and the partial contribution of a single sample on the final output as seen in equation 4.3. In the decomposed algorithm equation, h(n) represents the FIR filter coefficients, y is the running accumulation and Y is the next accumulation after summation. Notice that for any m value, the input x(n) sample is shared among all filter coefficients.

$$Y = y + h(n-m) \cdot x(n) \qquad (4.3)$$

This approach is analogous to dropping the energy data onto the centroid of the filter and computing the partial product of all results within scope as seen in Figures 4.6 and 4.7. Commuting the convolution eliminates the overhead associated with buffering all the energy data into the filter to calculate a single result and greatly simplifies the hardware implementation.

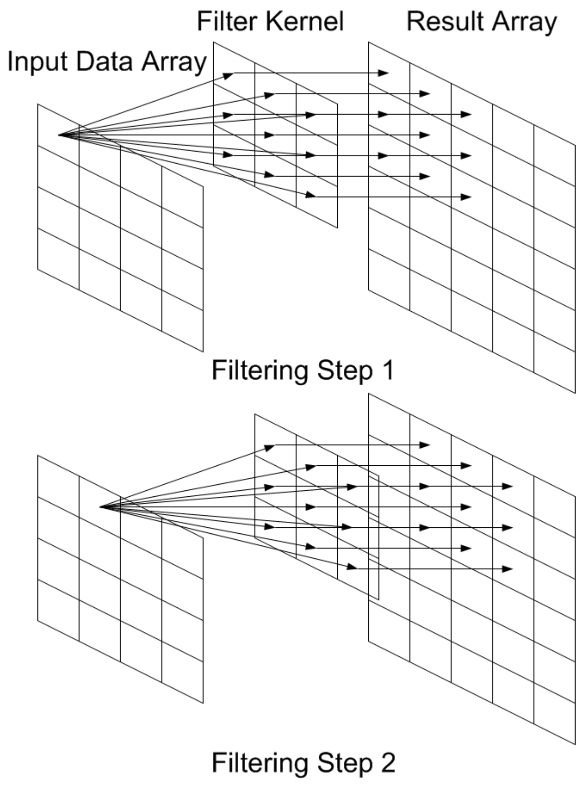
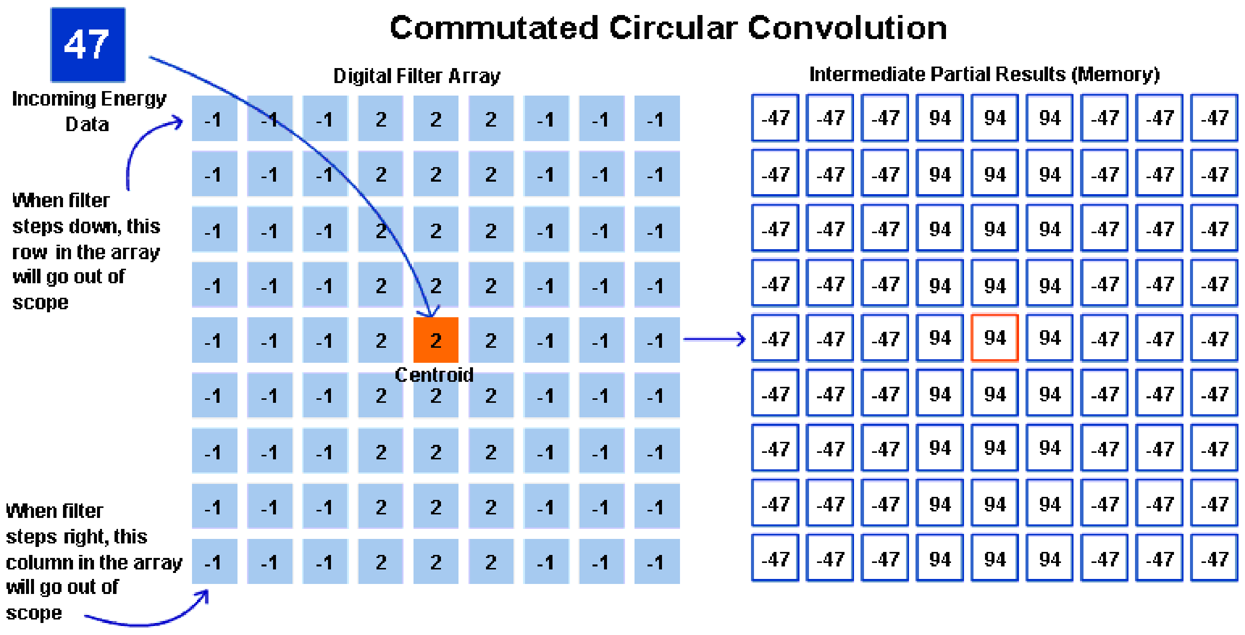


**Figure 4.6: Parallel filtering concept**


47
Commutated Circular Convolution
Digital Filter Array
Intermediate Partial Results (Memory)
Incoming Energy Data
When filter steps down, this row in the array will go out of scope
When filter steps right, this column in the array will go out of scope
Centroid

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |
| -1 | -1 | -1 | 2 | 2 | 2 | -1 | -1 | -1 |

|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |
| -47 | -47 | -47 | 94 | 94 | 94 | -47 | -47 | -47 |

**Figure 4.7: Commutated circular convolution example.**

The parallelized digital filters maximize the computations for any single 8-bits of energy data so that input buffering is kept at an absolute minimum while also offering quite a bit of freedom with regard to the form of the resulting data. Since the RED algorithm calls for a circular convolution, the parallelized digital filter computes the partial products and writes the results to memory into the appropriately wrapped memory locations; therefore, more complex convolutions can be handled simply by directing the partial products to the appropriate memory locations. While outside the scope of this paper, it is likely possible to compute the partial results directly into the finalized template, totally eliminating the need for intermediate memory but that is left to the discretion of the engineer to handle the filters in whatever way best fits the application.

## 4.4 TECHNICAL DISCUSSION

The RED algorithm digital filtering is designed and implemented into hardware (an FPGA) using Very High Speed Integrated Circuit Hardware Description Language (VHDL) and Altera Quartus II web edition design software. An FPGA can be programmed with a hardware design and logically behave like real hardware. VHDL allows the engineer to describe the design of the system and implement it onto the FPGA as a substrate for real-world testing. The Altera device targeted in the design process is the Stratix III FPGA where the digital filters consume about 6% of the average Stratix III device resources and approximately 4% of the device's available on-chip RAM blocks (also known as M9K blocks).

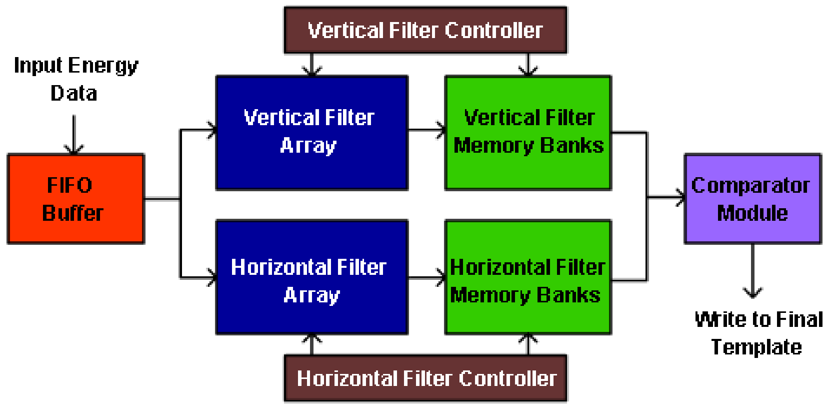

Vertical Filter Controller
Input Energy Data
Vertical Filter Array
Vertical Filter Memory Banks
FIFO Buffer
Comparator Module
Horizontal Filter Array
Horizontal Filter Memory Banks
Write to Final Template
Horizontal Filter Controller

**Figure 4.8: System block diagram**

The parallel hardware filter consists of 81 accumulator cores arranged in a nine by nine array to represent the original filter. The accumulators are grouped within the array by row since the energy data from the segmentation process (or most other common process) emerges left to right, top to bottom, easing the memory access as the filter array steps through the incoming data. Each row is accommodated with a block of simple dual ported ram to support all the simultaneous reading and writing needs of the accumulator core row. Also among each row is a multiplexer to narrow the control of memory to a single accumulator core and a row controller to grant any accumulator within the row memory control and issue the proper control signals to the row’s accumulator cores. Finally, the array is fitted with a first-in-first-out (FIFO) memory buffer to ease incoming data and a comparator module that judges which filter had the highest output and issues the appropriate bit to the finalized template for matching. The full FIR filtering system is shown in Figure 4.8.

The accumulator cores are the meat of the parallelized filter and are responsible for applying the filter coefficients to the incoming data and tracking the accumulated partial results. Each accumulator core is a finite state machine (a basic type of logic device) that is driven by both outside control signals and internal control signals. As 8-bit energy data becomes available from the FIFO buffer, an external “go” signal instructs the accumulator core to apply one of two coefficients to the data and add it to the contents of a 16-bit wide accumulation register. The accumulator cores also track where the partial results are stored in the local bank of memory and as the energy data and filter steps right 1 byte at a time, one accumulator core (the left-most in the filter scope) will go out of scope and need to exchange its accumulation register with memory to come back into scope. The address pointers for reading and writing to memory are updated at the end of the last accumulation cycle that was in scope and new data is read in and accumulated at the start of the next accumulation cycle. Since data memory must stabilize before being read, the accumulator cores cycle between an accumulation state and an idle state and thus requiring only two cycles to perform all partial results of the filter. However, a special case exists when the accumulator core array must step down and return as the segmented energy data moves onto the next row (much like the carriage return of a typewriter). During a step down operation, all accumulators in all rows must exchange data with local row memory which requires 3 cycles per accumulator (21 cycles total) which halts filter processing briefly. A brief block diagram of the accumulator state machine is shown in Figure 4.9.

.

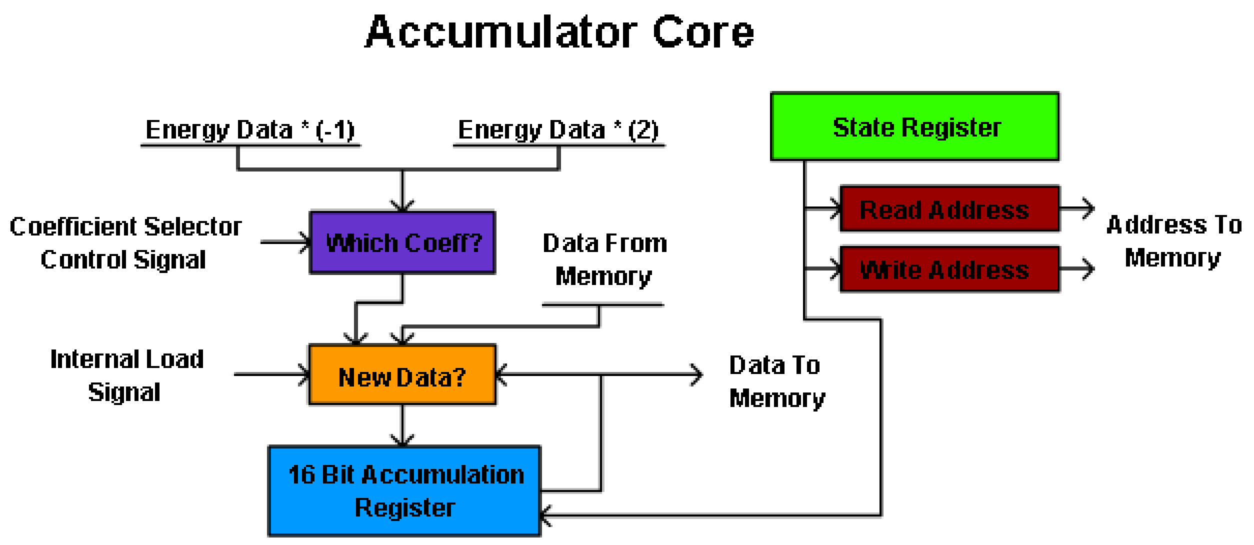


**Figure 4.9: Accumulator core state machine**

The accumulator cores are managed by a simple driver driven by logic lookup tables (LUTs) and arranged into rows to maximize efficiency and minimize the amount of memory used by the filters. In each step right operation, only the left most accumulator in the scope (not always the physical left most) will save the value in its accumulation register and address a read on the data value located in memory of the next column to come into scope. The row controller selects the appropriate data and address busses so that the accumulator reading and writing data has control of the memory. The row controller also handles the go signals for the accumulator cores (informing the accumulator core that energy data is stable coming from the FIFO buffer) and the coefficient select signals in order to make modifying and customizing the filter operation simpler and centralized.

Once the filtering is complete, the comparator module evaluates the result of each filter and compiles the final template ready for matching. The module subtracts the horizontal filter results from the vertical filter at each corresponding memory location in all row bank memories simultaneously and detects overflow. Depending on which filter's output was highest, the comparator module assigns the appropriate bit to the template.

# Chapter 5

# Results

## 5.1 Binary Morphology Hardware Implementation Performance

The functions for computing the binary dilation and erosion are isolated and the start and end times are recorded to compute the time in milliseconds the functions took to execute. The data is tabulated in Table 5.1. The images tested are standard 640 by 480 VGA images from the BATH database.

**Table 5.1: Timing comparisons of software and hardware implementations. 1000 iterations of the functions comprise these timing characteristics.**

| Software Functions | | Hardware Functions | |
|---|---|---|---|
| Mean | 1.341s | Mean | 6.144ms |
| Dilation | 611.7ms | | |
| Erosion | 729.3ms | Standard Deviation | 313µs |
| Standard Deviation | 72.476ms | Range | 5.943ms to 6.846ms |
| Range | 1.090s to 1.470s | | |

## 5.2 FIR Filtering Hardware Implementation Performance

The FIR filtering hardware successfully eliminated the filter kernel size impact on execution time of the template generation process in the algorithm. As stated before, the filter kernel's computations are loaded, accumulated and saved in parallel. Thus, the FIR filtering is dropped from a fourth order growth function of four nested loops to only second order. With the hardware FIR filtering, only the image size affects total processing time and each pixel only requires between two and three cycles to process. The resulting timing analysis is shown in Figure 5.1.

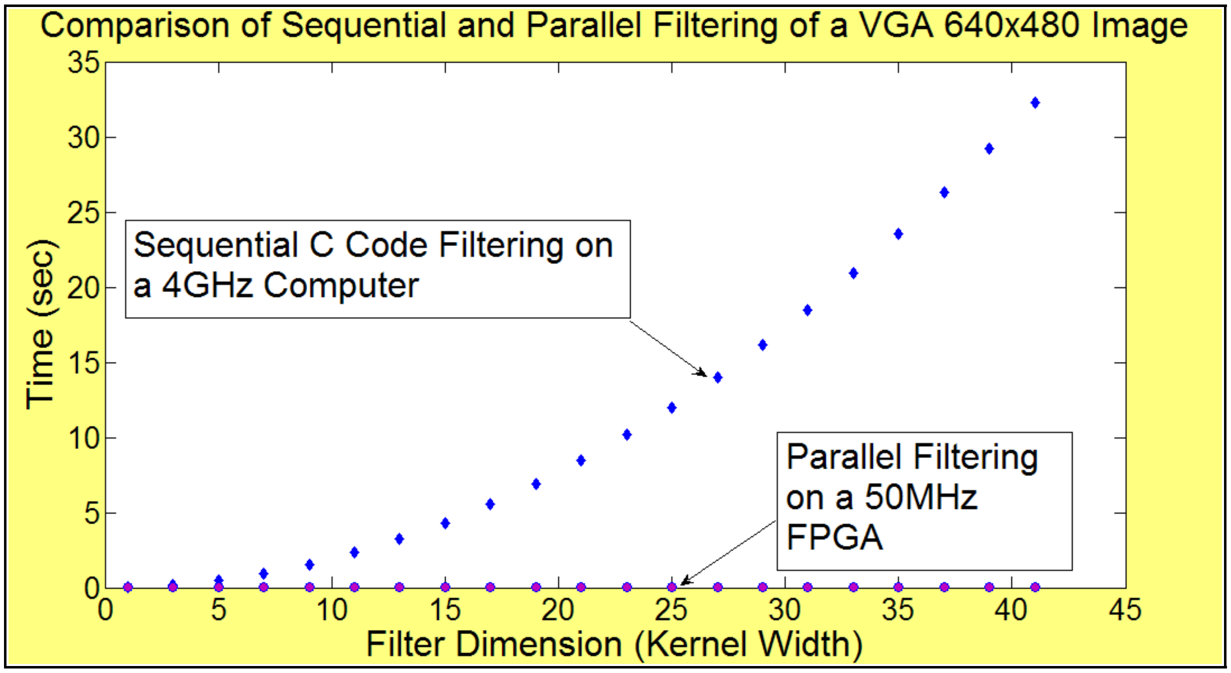


**Figure 5.1: FPGA vs GPP on VGA input and differing kernel sizes**

Figure 5.1 illustrates that the sequential C code filtering is in fact two orders of magnitude greater growth function than the parallel FPGA hardware implementation. Also, since the execution time on the FPGA is in the microseconds and the graph is time vs. the filter dimension (a 2nd order variable), the FPGA timing appears as a flat line across the bottom of the graph. This evaluation is independent of the actual process performed by the RED algorithm which uses at least 16 rows by 128 columns to generate the 2048 bit template.

# Chapter 6

# Conclusion

Application specific hardware design can be a complex process that may or may not be worth the cost of design for just small enhancements over software. Theoretically, an algorithm's execution could be nearly totally implemented with hardware functions and the processor would merely need to move data from one peripheral hardware device to another, assuming sufficient space on the programmable logic device or integrated circuit. This design is certainly not the only part of the RED iris recognition algorithm that can be implemented using hardware functions. However, some things are simply easier to do in software even though they may be slower.

## *6.1 Binary Morphology*

The hardware function outperforms the software in speed of execution. The software timing shows that the binary morphology dilation and erosion functions together take approximately 218 times longer to execute. In large processing applications that must process many images at a time this can be a significant increase. Furthermore, the increase makes the segmentation of the pupil from the image nearly negligible and increases total segmentation time by about 37%.

There are tradeoffs, however, and the hardware takes up additional space within the FPGA which also includes the microprocessor, memory controller and other peripherals. With today's synthesizing software that effectively optimizes the design for fit and power, the logic consumption takes up less than 1% of the logic elements and registers in the Stratix III EP3SL150F1152C3ES which has approximately 150,000 logic elements and 1 Mbit of on-chip memory. Subsequently, the power consumption in the overall system is reduced since the hardware implementation uses substantially less logic than the software loops used.

## *6.2 FIR Filtering*

The parallelized digital filtering has many advantages over the sequential and straightforward approach to the circular convolution filter. The parallel filter can be easily inserted into a pipelined dataflow where maximum processing for a single byte of data is desired. By modifying the convolution process slightly, the hardware is freed of most overhead associated with the sequential circular convolution. Also, the memory usage is only slightly greater in the parallelized convolution approach than the basic software or basic hardware convolution while significantly less than the parallelized memory needed to handle the hefty reading requirements of front loaded digital filtering of the basic hardware convolution. Furthermore, the partial approach to circular convolution presents many more opportunities for optimization and customization since the data flow can be channeled to meet the needs of the application.

Depending on the size of the energy data to be loaded, the time needed to complete the filtering can be easily calculated. The basic formula for how many cycles a complete convolution requires is shown below in equation 7.1:

$$t(x,y) = 2xy + 27y \tag{7.1}$$

Where x is the number of columns and y is the number of rows in the energy data. The equation is asymmetric because the parallelism is oriented with respect to the rows (x direction) and thus there is added latency when the filter array shifts up or down in the y direction. Also note that the size of the filter kernel has no effect on the execution time and thus a filter as large as infinity by infinity could be calculated in the same time as a filter with only a single coefficient (1x1). For the RED algorithm, the number of clock cycles needed to complete the template filtering is shown in equation 7.2 which is 90.56 microseconds at a 50 MHz clock frequency. An iris size of 128 radial lines and 16 concentric circles is chosen because this is the minimum resolution of the iris to compute the 2048 bit template.

$$t(128,16) = 4528 \text{ cycles} \tag{7.2}$$

In comparison, simulated tests of filtering were performed on a general purpose processor running at 4 GHz and multiple cores. This machine took an average of 32 milliseconds to compute a 9x9 kernel convolution on a large VGA (640x480) input. The same convolution was tested on the parallel filtering array and took only 1.3milliseconds. This is illustrated in Figure 5.1.

## 6.3 Wrapping Up

The final system could never fully be tested because of the design issues identified the DE3 board. Therefore, the most interesting comparison of how the overall embedded system performed next to the original general purpose machine could not be evaluated in the available time. Yet, the embedded system may one day prove to be preferable over the general purpose system for many reasons. An embedded system is independent of operating systems and environments designed for unrelated computing. Thus, this research developed an embedded system unique to iris recognition which is a feat in itself. The implementation in this research is a proof-of-concept to show that an algorithm, iris recognition or other that has value being portable, can be redesigned as a stand-alone device. Lower power operation and even more efficient and faster operation than anything ever before are likely possible given enough design resources, money and time.

# Bibliography

[1] Zhang, D. D., [Automated Biometrics: Technologies and Systems], Kluwer Academic Publishers, 1-18 (2000).

[2] Sims, D., "Biometrics Recognition: Our Hands, Eyes and Faces Give Us Away," IEEE Computer graphics and Applications, 0272-17-16/94 (1994).

[3] Williams, G. O., “Iris Recognition Technology,” IEEE, 0-7803-3537-6-9/96 (1996).

[4] Boles, W. W., “A Security System Based on Human Iris Identification Using Wavelet Transform,” First International Conference on Knowledge-Based Intelligent Electronic Systems, Adelaide, 21-23 (1997).

[5] Kennell, L. R., Ives, R. W., Gaunt, R. M., “Binary morphology and local statistics applied to iris segmentation for recognition,” Image Processing, Proc. of the 13th Annual International Conference on, (2006).

[6] Daugman, J., "Statistical richness of visual phase information." International Journal of Computer Vision 45(1), 25-38 (2001).

[7] Ives, R. W., Broussard, R. P., Kennell, L. R., Rakvic, R. N., Etter, D. M., “Recognition using the Ridge Energy Direction Algorithm” Signals, Systems and Computers, presented at the 42nd Asilomar Conference on, Pacific Grove, California, (2008).

[8] Daugman, J., "Probing the uniqueness and randomness of IrisCodes: Results from 200 billion iris pair comparisons." IEEE, Proc. of the, 94(11), 1927-1935 (2006).

[9] Daugman, J., “High Confidence Visual Recognition of a Persons By a Test of Statistical Independence,” PAMI, IEEE Trans. on, 15, (1993).

[10] Altera Inc., “Stratix III Datasheet.” [Online datasheet], Available at http://altera.com/literature/hb/stx3/stx3_siii52001.pdf, (2008).

[11] Altera Inc., “Stratix III FPGAs.” [Online datasheet], Available at http://www.eetchina.com/ARTICLES/2006NOV/PDF/br-stratixIII.pdf, (2008).

[12] Kung, H. T., “Why Systolic Architectures?” Computer, 15(1), 37-46 (1982).

# Appendix A: Embedded System Top Level VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
library altera;
use altera.altera_syn_attributes.all;

entity RED_System is
        port
        (
                -- Buttons
                Button : in std_logic_vector(3 downto 0);

                -- Clock Out
                CLK_OUT : out std_logic;

                -- DDR2
                mem_addr : out std_logic_vector(15 downto 0);
                mem_ba : out std_logic_vector(2 downto 0);
                mem_cas_n : out std_logic;
                mem_cke : out std_logic_vector(1 downto 0);
                mem_clk_n : inout std_logic_vector(1 downto 0);
                mem_clk : inout std_logic_vector(1 downto 0);
                mem_cs_n : out std_logic_vector(1 downto 0);
                mem_dm : out std_logic_vector(7 downto 0);
                mem_dq : inout std_logic_vector(63 downto 0);
                mem_dqs : inout std_logic_vector(7 downto 0);
                mem_dqsn : inout std_logic_vector(7 downto 0);
                mem_odt : out std_logic_vector(1 downto 0);
                mem_ras_n : out std_logic;
                mem_we_n : out std_logic;

                -- DIP Switches
                DIP_SW : in std_logic_vector(7 downto 0);

                -- Clocks
                EXT_CLK : in std_logic;
                OSC1_50 : in std_logic;
                OSC2_50 : in std_logic;
                OSC_BA : in std_logic;
                OSC_BB : in std_logic;
                OSC_BC : in std_logic;
                OSC_BD : in std_logic;

                -- Seven Segment
                HEX0 : out std_logic_vector(6 downto 0);
                HEX0_DP : out std_logic;
                HEX1 : out std_logic_vector(6 downto 0);
                HEX1_DP : out std_logic;

                -- JTAG
                JVC_CLK : out std_logic;
                JVC_CS : out std_logic;
                JVC_DATAIN : in std_logic;
                JVC_DATAOUT : out std_logic;

                -- LEDs
                LEDB : out std_logic_vector(7 downto 0);
                LEDG : out std_logic_vector(7 downto 0);
                LEDR : out std_logic_vector(7 downto 0);

                -- USB
                OTG_A : out std_logic_vector(17 downto 1);
                OTG_CS_n : out std_logic;
                OTG_D : inout std_logic_vector(31 downto 0);
```

```
               OTG_DC_DACK : out std_logic;
               OTG_DC_DREQ : in std_logic;
               OTG_DC_IRQ : in std_logic;
               OTG_HC_DACK : out std_logic;
               OTG_HC_DREQ : in std_logic;
               OTG_HC_IRQ : in std_logic;
               OTG_OE_n : out std_logic;
               OTG_RESET_n : out std_logic;
               OTG_WE_n : out std_logic;

               -- SD Card
               SD_CLK : out std_logic;
               SD_CMD : inout std_logic;
               SD_DAT : inout std_logic;
               SD_WPn : in std_logic;

               -- Slide Switches
               SW : in std_logic_vector(3 downto 0);

               -- Temp Sensor
               TEMP_CLK : out std_logic;
               TEMP_DATA : inout std_logic;
               TEMP_INTn : in std_logic

       );

end RED_System;

architecture ppl_type of RED_System is

       signal usb_addr       : std_logic_vector(16 downto 0);

       --==============================================
       --  IO Group Voltage Configuration (Do not modify it)
       --==============================================
       component IOV_A3V3_B1V8_C3V3_D3V3 is
               port (

                       signal iCLK    : IN STD_LOGIC;
                       signal iRST_n  : IN STD_LOGIC;
                       signal iENABLE : IN STD_LOGIC;
                       signal oREADY  : OUT STD_LOGIC;
                       signal oERR           : OUT STD_LOGIC;
                       signal oERRCODE       : OUT STD_LOGIC;
                       signal oJVC_CLK       : OUT STD_LOGIC;
                       signal oJVC_CS : OUT STD_LOGIC;
                       signal oJVC_DATAOUT   : OUT STD_LOGIC;
                       signal iJVC_DATAIN    : IN STD_LOGIC
               );
       end component;


       component RED_SOPC is
        port (
             -- 1) global signals:
                signal altmemddr_aux_full_rate_clk_out : OUT STD_LOGIC;
                signal altmemddr_aux_half_rate_clk_out : OUT STD_LOGIC;
                signal altmemddr_phy_clk_out : OUT STD_LOGIC;
                signal clk : IN STD_LOGIC;
                signal reset_n : IN STD_LOGIC;

             -- the_ISP1761
                signal A_from_the_ISP1761 : OUT STD_LOGIC_VECTOR (17 DOWNTO 1);
                signal CS_N_from_the_ISP1761 : OUT STD_LOGIC;
                signal DC_DACK_from_the_ISP1761 : OUT STD_LOGIC;
                signal DC_DREQ_to_the_ISP1761 : IN STD_LOGIC;
                signal DC_IRQ_to_the_ISP1761 : IN STD_LOGIC;
                signal D_to_and_from_the_ISP1761 : INOUT STD_LOGIC_VECTOR (31 DOWNTO 0);
```

```
              signal HC_DACK_from_the_ISP1761 : OUT STD_LOGIC;
              signal HC_DREQ_to_the_ISP1761 : IN STD_LOGIC;
              signal HC_IRQ_to_the_ISP1761 : IN STD_LOGIC;
              signal RD_N_from_the_ISP1761 : OUT STD_LOGIC;
              signal WR_N_from_the_ISP1761 : OUT STD_LOGIC;

           -- the_SEG7
              signal avs_s1_export_seg7_from_the_SEG7 : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);

           -- the_altmemddr
              signal global_reset_n_to_the_altmemddr : IN STD_LOGIC;
              signal local_init_done_from_the_altmemddr : OUT STD_LOGIC;
              signal local_refresh_ack_from_the_altmemddr : OUT STD_LOGIC;
              signal local_wdata_req_from_the_altmemddr : OUT STD_LOGIC;
              signal mem_addr_from_the_altmemddr : OUT STD_LOGIC_VECTOR (12 DOWNTO 0);
              signal mem_ba_from_the_altmemddr : OUT STD_LOGIC_VECTOR (1 DOWNTO 0);
              signal mem_cas_n_from_the_altmemddr : OUT STD_LOGIC;
              signal mem_cke_from_the_altmemddr : OUT STD_LOGIC;
              signal mem_clk_n_to_and_from_the_altmemddr : INOUT STD_LOGIC_VECTOR (1 DOWNTO
0);
              signal mem_clk_to_and_from_the_altmemddr : INOUT STD_LOGIC_VECTOR (1 DOWNTO 0);
              signal mem_cs_n_from_the_altmemddr : OUT STD_LOGIC;
              signal mem_dm_from_the_altmemddr : OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
              signal mem_dq_to_and_from_the_altmemddr : INOUT STD_LOGIC_VECTOR (63 DOWNTO 0);
              signal mem_dqs_to_and_from_the_altmemddr : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
              signal mem_dqsn_to_and_from_the_altmemddr : INOUT STD_LOGIC_VECTOR (7 DOWNTO 0);
              signal mem_odt_from_the_altmemddr : OUT STD_LOGIC;
              signal mem_ras_n_from_the_altmemddr : OUT STD_LOGIC;
              signal mem_we_n_from_the_altmemddr : OUT STD_LOGIC;
              signal oct_ctl_rs_value_to_the_altmemddr : IN STD_LOGIC_VECTOR (13 DOWNTO 0);
              signal oct_ctl_rt_value_to_the_altmemddr : IN STD_LOGIC_VECTOR (13 DOWNTO 0);
              signal reset_phy_clk_n_from_the_altmemddr : OUT STD_LOGIC;

           -- the_pio_button
              signal in_port_to_the_pio_button : IN STD_LOGIC_VECTOR (3 DOWNTO 0);

           -- the_pio_led
              signal out_port_from_the_pio_led : OUT STD_LOGIC_VECTOR (23 DOWNTO 0);

           -- the_usb_reset_n
              signal out_port_from_the_usb_reset_n : OUT STD_LOGIC
           );
        end component;

begin

        IOV_Inst: IOV_A3V3_B1V8_C3V3_D3V3
        port map (

                     iCLK    => OSC2_50,
                     iRST_n  => '1',
                     iENABLE => '1',
                     --READY => ,
                     --ERR => ,
                     --ERRCODE => ,
                     oJVC_CLK => JVC_CLK,
                     oJVC_CS => JVC_CS,
                     oJVC_DATAOUT => JVC_DATAOUT,
                     iJVC_DATAIN => JVC_DATAIN
        );


        RED_inst: RED_SOPC
        port map
        (


          -- 1) global signals:
            -- altmemddr_aux_full_rate_clk_out => <signal>,     -- Unused
            -- altmemddr_aux_half_rate_clk_out => <signal>,     -- Unused
            -- altmemddr_phy_clk_out => <signal>,                              -- Unused
            clk => OSC1_50,
            reset_n => '1',


      -- the USB
            A_from_the_ISP1761(17 downto 1) => OTG_A(17 downto 1),
            CS_N_from_the_ISP1761 => OTG_CS_n,
            DC_DACK_from_the_ISP1761 => OTG_DC_DACK,
            DC_DREQ_to_the_ISP1761 => OTG_DC_DREQ,
            DC_IRQ_to_the_ISP1761 => OTG_DC_IRQ,
            D_to_and_from_the_ISP1761 => OTG_D,
            HC_DACK_from_the_ISP1761 => OTG_HC_DACK,
            HC_DREQ_to_the_ISP1761 => OTG_HC_DREQ,
            HC_IRQ_to_the_ISP1761 => OTG_HC_IRQ,
            RD_N_from_the_ISP1761 => OTG_OE_n,
            WR_N_from_the_ISP1761 => OTG_WE_n,
            out_port_from_the_usb_reset_n => OTG_RESET_n,

      -- SEG7
            avs_s1_export_seg7_from_the_SEG7(15) => HEX1_DP,
            avs_s1_export_seg7_from_the_SEG7(14 downto 8) => HEX1,
            avs_s1_export_seg7_from_the_SEG7(7) => HEX0_DP,
            avs_s1_export_seg7_from_the_SEG7(6 downto 0) => HEX0,


      -- Buttons
            in_port_to_the_pio_button => Button,


      -- LEDs
            out_port_from_the_pio_led(23 downto 16) => LEDR,
            out_port_from_the_pio_led(15 downto 8) => LEDG,
            out_port_from_the_pio_led(7 downto 0) => LEDB,


      -- the_altmemddr
            global_reset_n_to_the_altmemddr => '1',
            mem_addr_from_the_altmemddr => mem_addr(12 downto 0),
            mem_ba_from_the_altmemddr => mem_ba(1 downto 0),
            mem_cas_n_from_the_altmemddr => mem_cas_n,
            mem_cke_from_the_altmemddr => mem_cke(0),
            mem_clk_n_to_and_from_the_altmemddr => mem_clk_n,
            mem_clk_to_and_from_the_altmemddr => mem_clk,
            mem_cs_n_from_the_altmemddr => mem_cs_n(0),
            mem_dm_from_the_altmemddr => mem_dm,
            mem_dq_to_and_from_the_altmemddr => mem_dq,
            mem_dqs_to_and_from_the_altmemddr => mem_dqs,
            mem_dqsn_to_and_from_the_altmemddr => mem_dqsn,
            mem_odt_from_the_altmemddr => mem_odt(0),
            mem_ras_n_from_the_altmemddr => mem_ras_n,
            mem_we_n_from_the_altmemddr => mem_we_n,

            oct_ctl_rs_value_to_the_altmemddr => "00000000000000",
            oct_ctl_rt_value_to_the_altmemddr => "00000000000000"


    );


end;
```

# Appendix B: Binary Morphology VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;

entity BM_Dilate_Erode is

        generic
        (
                ROWS : natural := 4;
                COLS : natural := 4
        );

        port
        (

                avs_s1_clk : in std_logic;
                --avs_s1_reset_n : in std_logic := '1';
                avs_s1_read : in std_logic;
                avs_s1_write : in std_logic;
                avs_s1_chipselect : in std_logic;
                avs_s1_address : in std_logic;
                avs_s1_readdata : out std_logic_vector(7 downto 0);
                avs_s1_writedata : in std_logic_vector(7 downto 0)

        );

end BM_Dilate_Erode;

architecture rtl of BM_Dilate_Erode is

        signal loop_mux, loop_data    :       std_logic := '0';
        signal local_data      :       std_logic_vector(ROWS*COLS downto 0);
        signal local_interconnect     :       std_logic_vector(ROWS*COLS downto 0);
        signal clk_e   :       std_logic := '0';
        signal shift_load      :       std_logic := '0';
        signal done            :       std_logic := '0';
        signal operation       :       std_logic := '0';

        component BM_Element is


                port(
                        -- synchronous
                        clk                   :       in      std_logic;
                        clk_enable     :       in      std_logic;

                        -- asynchronous
                        --reset        :       in      std_logic;
                        aload          :       in      std_logic;
                        adatain        :       in      std_logic;
                        adataout       :       out     std_logic;

                        -- input
                        --topleft             :       in      std_logic;
                        topmid         :       in      std_logic;
                        --topright     :       in      std_logic;
                        midleft        :       in      std_logic;
                        midright       :       in      std_logic;
                        --bottomleft   :       in      std_logic;
                        bottommid      :       in      std_logic;
                        --bottomright  :       in      std_logic;
                        operation      :       in      std_logic;

                        -- output
                        output          :       out     std_logic
                );

        end component;

begin


        controller: process(avs_s1_clk, avs_s1_chipselect, avs_s1_write)
                variable dilate_erode : natural := 0; -- 30 Dilate/Erode
                variable count : natural := 0;                       -- Track number of pixels
loaded and unloaded
        begin
                --if(avs_s1_reset_n = '0') then       -- Asynchronous reset
                        --dilate_erode := 0;
                        --count := 0;
                --elsif(rising_edge(avs_s1_clk)) then
                if(rising_edge(avs_s1_clk)) then
                        if(avs_s1_chipselect = '1' and avs_s1_write = '1') then             -- We
are being written to
                                if(count < ROWS*COLS) then
                                -- Only allow rows*cols number of pixels in
                                        local_data(0) <= avs_s1_writedata(0);
                        -- Place write data into first shift BM_Element
                                        shift_load <= '1';
                                        -- Shift loading = TRUE
                                        count := count + 1;
                                        -- Increment pixels count
                                else
                                        -- Written to?
                                        count := ROWS*COLS;
                                        -- Error: Hold at same number of pixels in image
                                        shift_load <= '0';
                                        -- Do not shift
                                end if;
                        elsif(avs_s1_chipselect = '1' and avs_s1_read = '1') then   -- We are
being read from
                                if(avs_s1_address = '0') then
                        -- Reading..
                                        if(count > 0 and dilate_erode = 30) then
                        -- Still data?
                                                avs_s1_readdata(0) <= local_data(ROWS*COLS);
                        -- Place data out
                                                shift_load <= '1';
                                                -- Shift data out
                                                count := count - 1;
                                                -- Decremenet pixels count
                                        else
                                                avs_s1_readdata(0) <= '0';
                                                -- Don't output
                                                shift_load <= '0';
                                                -- Don't shift
                                                count := 0;
                                                        -- Keep count zero
                                        end if;
                                elsif(avs_s1_address = '1') then
                        -- Polling...
                                        if(dilate_erode = 30) then
                                        -- Done?
                                                avs_s1_readdata(0) <= '1';
                                                -- Yes
                                        else
                                                avs_s1_readdata(0) <= '0';
                                                -- No
                                        end if;
                                end if;

                        end if;
                        if(dilate_erode > 14) then
                                operation <= '1';
                        else
                                operation <= '0';
                        end if;

                        if(count = ROWS*COLS and dilate_erode < 30) then                    --
Processing
                                clk_e <= '1';
                                dilate_erode := dilate_erode + 1;
                        elsif(count = ROWS*COLS and dilate_erode = 30) then
                                clk_e <= '0';
                                dilate_erode := 30;
                        end if;
                end if;
        end process;


        BM_rows: for j in 0 to ROWS - 1 generate

        begin

                BM_cols: for i in 0 to COLS - 1 generate

                begin
                        --loop_mux <= local_data(j*COLS + i) when i+j > 0 else loop_data;

                        U1: BM_Element


                        port map(
                                clk     => avs_s1_clk,
                                clk_enable => clk_e,
                                operation => operation,

                                -- asynchronous
                                --reset => avs_s1_reset_n,
                                aload => shift_load,
                                adatain => local_data(j*COLS + i),
                                adataout => local_data(j*COLS + i + 1),



                                -- top
                                --topleft => local_interconnect( ((j + ROWS - 1) mod ROWS)*COLS +
((i + COLS - 1) mod COLS) ),
                                topmid => local_interconnect( ((j + ROWS - 1) mod ROWS)*COLS + ((i
+ COLS) mod COLS) ),
                                --topright => local_interconnect( ((j + ROWS - 1) mod ROWS)*COLS +
((i + COLS + 1) mod COLS) ),

                                -- mid
                                midleft => local_interconnect( (j*COLS) + ((i+(COLS-1)) mod
COLS) ),
                                midright => local_interconnect( (j*COLS) + ((i+(COLS+1)) mod
COLS) ),

                                -- bottom
                                --bottomleft => local_interconnect( ((j + ROWS + 1) mod ROWS)*COLS
+ ((i + COLS - 1) mod COLS) ),
                                bottommid => local_interconnect( ((j + ROWS + 1) mod ROWS)*COLS +
((i + COLS) mod COLS) ),
                                --bottomright => local_interconnect( ((j + ROWS + 1) mod ROWS)*COLS
+ ((i + COLS + 1) mod COLS) ),

                                -- output
```

```
                              output => local_interconnect( j*COLS + i) -- point to current
element
                      );
               end generate;

       end generate;

       --out_memory <= local_interconnect;

end rtl;


library ieee;
use ieee.std_logic_1164.all;

entity BM_Element is


       port(
              -- synchronous
              clk                    :      in     std_logic;
              clk_enable      :      in     std_logic;

              -- asynchronous
              --reset         :      in     std_logic;
              aload           :      in     std_logic;
              adatain         :      in     std_logic;
              adataout        :      out    std_logic;

              -- input
              --topleft              :      in     std_logic;
              topmid          :      in     std_logic;
              --topright      :      in     std_logic;
              midleft         :      in     std_logic;
              midright        :      in     std_logic;
              --bottomleft    :      in     std_logic;
              bottommid       :      in     std_logic;
              --bottomright   :      in     std_logic;
              operation       :      in     std_logic;

              -- output
              output          :      out    std_logic
       );

end entity;

architecture rtl of BM_Element is

       signal  pixel, pixel_step     :      std_logic;
       -- signal      pixel_step     :      std_logic;
begin


process(clk, pixel, pixel_step)
begin
       pixel <= pixel_step;
end process;

-- In Altera devices, register signals have a set priority.
-- The HDL design should reflect this priority.
process(aload, adatain, clk)
begin
       -- The asynchronous reset signal has the highest priority
       --if (reset = '0') then
              --pixel_step <= '0';
       -- Asynchronous load has next-highest priority
       if (aload = '1') then
              adataout <= pixel;
```

```
               pixel_step <= adatain;
        else
               -- At a clock edge, if asynchronous signals have not taken priority,
               -- respond to the appropriate synchronous signal.
               -- Check for synchronous reset, then synchronous load.
               -- If none of these takes precedence, update the register output
               -- to be the register input.
               if (rising_edge(clk)) then
                      if(clk_enable = '1') then
                             if(operation = '0') then              -- Dilate
                                    --pixel_step <= topleft or topmid or topright or midleft or
midright or bottomleft or bottommid or bottomright;
                                    pixel_step <=topmid or midleft or midright or bottommid;
                             else
                                    pixel_step <= not(topmid or midleft or midright or
bottommid);
                             end if;
                      end if;
               end if;
        end if;
end process;

process(clk)
begin
        if(rising_edge(clk)) then
               if(operation = '0') then
                      output <= pixel;
               else
                      output <= not(pixel);
               end if;
        end if;
end process;

end rtl;
```

# Appendix C: FIR Filtering VHDL Code

```
library ieee;
use ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.ALL;
use ieee.numeric_std.all;

entity TemplateTest is
	generic
	(
		ADDRESS_WIDTH  : natural  :=   8;
		DATA_WIDTH     : natural  :=  8
	);


	port
	(
		-- Input ports
		clock          : in std_logic;
		go_input       : in std_logic := '0';
		reset_in       : in std_logic;
		data_input     : in std_logic_vector((DATA_WIDTH-1) downto 0);


		-- Output ports
		wide_data_out : out std_logic_vector((9*2*DATA_WIDTH-1) downto 0);

		wide_wr_out : out std_logic_vector(8 downto 0)

	);
end TemplateTest;



architecture rtl of TemplateTest is

	-- 16 bit memory
	signal a_data : std_logic_vector((DATA_WIDTH-1) downto 0);
	signal b_data : std_logic_vector((DATA_WIDTH-1) downto 0);
	signal data_write_bus : std_logic_vector((9*2*DATA_WIDTH-1) downto 0);
	signal data_read_bus : std_logic_vector((9*2*DATA_WIDTH-1) downto 0);


	signal wr : std_logic_vector(8 downto 0) := "000000000";
	signal sel_col : natural := 0;

	signal addr_write_bus : std_logic_vector((9*ADDRESS_WIDTH-1) downto 0);
	signal addr_read_bus : std_logic_vector((9*ADDRESS_WIDTH-1) downto 0);
	signal square_data : std_logic_vector((2*DATA_WIDTH-1) downto 0);


	component TemplateElement

	generic
	(
		DATA_WIDTH : natural := 8;
		ADDRESS_WIDTH : natural := 8;
		INITIAL_COL : natural;
		INITIAL_ROW : natural;
		ARRAY_COL      : natural;
		ARRAY_ROW      : natural
	);

	port
	(
        go                      : in std_logic := '0';
        coeff           : in std_logic := '0';
        new_row_in      : in std_logic := '0';
        reset           : in std_logic := '0';
        clk                     : in std_logic;

        -- 16 bit memory
        a                       : in std_logic_vector  ((DATA_WIDTH-1) downto 0);
        b                       : in std_logic_vector  ((DATA_WIDTH-1) downto 0);
        data_in         : in std_logic_vector ((2*DATA_WIDTH-1) downto 0);
        data_out                : out std_logic_vector ((2*DATA_WIDTH-1) downto 0);


        read_write      : out std_logic_vector(0 downto 0);
        addr_in         : out std_logic_vector((ADDRESS_WIDTH-1) downto 0);
        addr_out        : out std_logic_vector((ADDRESS_WIDTH-1) downto 0)
);

end component;

component RowBankMemory is

        generic
        (
                DATA_WIDTH : natural := 8;
                ADDR_WIDTH : natural := 8
        );

        port
        (
                clk             : in std_logic;
                raddr   : in natural range 0 to 2**ADDR_WIDTH - 1;
                waddr   : in natural range 0 to 2**ADDR_WIDTH - 1;
                -- 16 bit memory
                data    : in std_logic_vector((2*DATA_WIDTH-1) downto 0);
                q       : out std_logic_vector((2*DATA_WIDTH -1) downto 0);


                we              : in std_logic_vector(0 downto 0) := "0"
        );

end component;

component RowMUX is
        generic
        (
                DATA_WIDTH      : natural  :=  8;
                ADDRESS_WIDTH   : natural  :=  8
        );

        port
        (
                -- Input ports
                data_write_bus0        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus1        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus2        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus3        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus4        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus5        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus6        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus7        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
                data_write_bus8        : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);

                addr_write_bus0 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                addr_write_bus1 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                addr_write_bus2 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                addr_write_bus3 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                addr_write_bus4 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
```

```
                  addr_write_bus5 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_write_bus6 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_write_bus7 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_write_bus8 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);

                  addr_read_bus0 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus1 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus2 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus3 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus4 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus5 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus6 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus7 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus8 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);

                  wr0 : in std_logic_vector(0 downto 0);
                  wr1 : in std_logic_vector(0 downto 0);
                  wr2 : in std_logic_vector(0 downto 0);
                  wr3 : in std_logic_vector(0 downto 0);
                  wr4 : in std_logic_vector(0 downto 0);
                  wr5 : in std_logic_vector(0 downto 0);
                  wr6 : in std_logic_vector(0 downto 0);
                  wr7 : in std_logic_vector(0 downto 0);
                  wr8 : in std_logic_vector(0 downto 0);

                  col_sel        : natural;

                  -- Output ports
                  data_write_bus_out     : out std_logic_vector((2*DATA_WIDTH-1) downto 0);
                  addr_write_bus_out     : out std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  addr_read_bus_out      : out std_logic_vector((ADDRESS_WIDTH-1) downto 0);
                  wr_out  : out std_logic_vector(0 downto 0)
            );
      end component;

      component FilterRowController is

            generic
            (
                  ROW_INIT       : natural := 0
            );

            port
            (
                  -- Input ports
                  go                     : in std_logic;
                  -- reset               : in std_logic;
                  clk                    : in std_logic;

                  -- Output ports
                  go_cmd  : out std_logic;
                  oper    : out std_logic_vector(8 downto 0);
                  -- new_col_out : out std_logic_vector(8 downto 0);
                  new_row_out    : out std_logic_vector(8 downto 0);
                  col_sel : out std_logic_vector(3 downto 0)
                  -- col_count_out : out std_logic_vector(7 downto 0)
            );

      end component;



begin
      process(data_input, square_data, go_input)
      begin
            -- square_data <= data_input*data_input;

            -- 16 bit memory
```

```
		-- Coefficient of 1
		a_data <= data_input((DATA_WIDTH-1) downto 0);

		-- Coefficient of -0.5
		b_data(DATA_WIDTH-1) <= '0'; -- not(square_data(2*DATA_WIDTH-1));
		b_data((DATA_WIDTH-2) downto 0) <= data_input((DATA_WIDTH-1) downto 1); --
not(square_data((2*DATA_WIDTH-1) downto DATA_WIDTH+1))+1;


	end process;



	filter_rows: for j in 0 to 0 generate -- 12 28

		-- Local Row Signals
		signal local_wr		: std_logic_vector(8 downto 0);
		signal local_wr_data	: std_logic_vector((9*2*DATA_WIDTH-1) downto 0);
		signal local_wr_addr	: std_logic_vector((9*ADDRESS_WIDTH-1) downto 0);
		signal local_rd_addr	: std_logic_vector((9*ADDRESS_WIDTH-1) downto 0);
		signal col_control	: std_logic_vector(8 downto 0);
		signal row_control	: std_logic_vector(8 downto 0);
		signal oper_control	: std_logic_vector(8 downto 0);
		signal go_control	: std_logic;
		signal col_selects	: std_logic_vector(3 downto 0);

	begin

		filter_cols: for i in 0 to 8 generate -- 125 252

		begin
			U1: TemplateElement2
			generic map(
				INITIAL_COL => ((i + 124) mod 128),
				INITIAL_ROW => ((j + 12) mod 16),
				ARRAY_COL => i,
				ARRAY_ROW => j
			)

			port map(
				go => go_control,
				coeff => oper_control(i),
				-- new_col => col_control(i),
				new_row_in => row_control(i),
				clk => clock,
				a => a_data,
				b => b_data,
				reset => reset_in,
				data_in => data_read_bus((j*16 + (2*DATA_WIDTH-1)) downto j*16),

				read_write => local_wr(i downto i),
				data_out => local_wr_data(((i+1)*2*DATA_WIDTH-1) downto
(i*2*DATA_WIDTH)),
				addr_out => local_wr_addr(((i+1)*ADDRESS_WIDTH-1) downto
(i*ADDRESS_WIDTH)),
				addr_in => local_rd_addr(((i+1)*ADDRESS_WIDTH-1) downto
(i*ADDRESS_WIDTH))
			);

--			wr(j downto j) <= local_wr when sel_col = i else "Z";
--			data_write_bus((j*2*DATA_WIDTH + (2*DATA_WIDTH-1)) downto j*2*DATA_WIDTH)
<= local_wr_data when sel_col = i else "ZZZZZZZZZZZZZZZZ";
--			addr_write_bus((j*ADDRESS_WIDTH + (ADDRESS_WIDTH-1)) downto
(j*ADDRESS_WIDTH)) <= local_wr_addr when sel_col = i else "ZZZZZZZZ";
--			addr_read_bus((j*ADDRESS_WIDTH + (ADDRESS_WIDTH-1)) downto
(j*ADDRESS_WIDTH)) <= local_rd_addr when sel_col = i else "ZZZZZZZZ";
		end generate;
		U4: VertFilterRowController
		generic map(
			ROW_INIT => j
		)

		port map(
			go => go_input,
			-- reset => reset_in,
			clk    => clock,

			-- Output ports
			go_cmd => go_control,
			oper => oper_control,
			-- new_col_out => col_control,
			new_row_out => row_control,
			col_sel => col_selects
		);

		U3: RowMUX
		port map(
			data_write_bus0       => local_wr_data((1*2*DATA_WIDTH-1) downto
(0*2*DATA_WIDTH)),
			data_write_bus1       => local_wr_data((2*2*DATA_WIDTH-1) downto
(1*2*DATA_WIDTH)),
			data_write_bus2       => local_wr_data((3*2*DATA_WIDTH-1) downto
(2*2*DATA_WIDTH)),
			data_write_bus3       => local_wr_data((4*2*DATA_WIDTH-1) downto
(3*2*DATA_WIDTH)),
			data_write_bus4       => local_wr_data((5*2*DATA_WIDTH-1) downto
(4*2*DATA_WIDTH)),
			data_write_bus5       => local_wr_data((6*2*DATA_WIDTH-1) downto
(5*2*DATA_WIDTH)),
			data_write_bus6       => local_wr_data((7*2*DATA_WIDTH-1) downto
(6*2*DATA_WIDTH)),
			data_write_bus7       => local_wr_data((8*2*DATA_WIDTH-1) downto
(7*2*DATA_WIDTH)),
			data_write_bus8       => local_wr_data((9*2*DATA_WIDTH-1) downto
(8*2*DATA_WIDTH)),

			addr_write_bus0 => local_wr_addr((1*ADDRESS_WIDTH-1) downto
(0*ADDRESS_WIDTH)),
			addr_write_bus1 => local_wr_addr((2*ADDRESS_WIDTH-1) downto
(1*ADDRESS_WIDTH)),
			addr_write_bus2 => local_wr_addr((3*ADDRESS_WIDTH-1) downto
(2*ADDRESS_WIDTH)),
			addr_write_bus3 => local_wr_addr((4*ADDRESS_WIDTH-1) downto
(3*ADDRESS_WIDTH)),
			addr_write_bus4 => local_wr_addr((5*ADDRESS_WIDTH-1) downto
(4*ADDRESS_WIDTH)),
			addr_write_bus5 => local_wr_addr((6*ADDRESS_WIDTH-1) downto
(5*ADDRESS_WIDTH)),
			addr_write_bus6 => local_wr_addr((7*ADDRESS_WIDTH-1) downto
(6*ADDRESS_WIDTH)),
			addr_write_bus7 => local_wr_addr((8*ADDRESS_WIDTH-1) downto
(7*ADDRESS_WIDTH)),
			addr_write_bus8 => local_wr_addr((9*ADDRESS_WIDTH-1) downto
(8*ADDRESS_WIDTH)),

			addr_read_bus0 => local_rd_addr((1*ADDRESS_WIDTH-1) downto
(0*ADDRESS_WIDTH)),
			addr_read_bus1 => local_rd_addr((2*ADDRESS_WIDTH-1) downto
(1*ADDRESS_WIDTH)),
			addr_read_bus2 => local_rd_addr((3*ADDRESS_WIDTH-1) downto
(2*ADDRESS_WIDTH)),
			addr_read_bus3 => local_rd_addr((4*ADDRESS_WIDTH-1) downto
(3*ADDRESS_WIDTH)),
			addr_read_bus4 => local_rd_addr((5*ADDRESS_WIDTH-1) downto
(4*ADDRESS_WIDTH)),
                        addr_read_bus5 => local_rd_addr((6*ADDRESS_WIDTH-1) downto
(5*ADDRESS_WIDTH)),
                        addr_read_bus6 => local_rd_addr((7*ADDRESS_WIDTH-1) downto
(6*ADDRESS_WIDTH)),
                        addr_read_bus7 => local_rd_addr((8*ADDRESS_WIDTH-1) downto
(7*ADDRESS_WIDTH)),
                        addr_read_bus8 => local_rd_addr((9*ADDRESS_WIDTH-1) downto
(8*ADDRESS_WIDTH)),

                        wr0 => local_wr(0 downto 0),
                        wr1 => local_wr(1 downto 1),
                        wr2 => local_wr(2 downto 2),
                        wr3 => local_wr(3 downto 3),
                        wr4 => local_wr(4 downto 4),
                        wr5 => local_wr(5 downto 5),
                        wr6 => local_wr(6 downto 6),
                        wr7 => local_wr(7 downto 7),
                        wr8 => local_wr(8 downto 8),

                        col_sel => conv_integer(col_selects),

                        -- Output ports
                        data_write_bus_out     => data_write_bus((j*2*DATA_WIDTH + (2*DATA_WIDTH-
1)) downto j*2*DATA_WIDTH),
                        addr_write_bus_out     => addr_write_bus((j*ADDRESS_WIDTH + (ADDRESS_WIDTH-
1)) downto (j*ADDRESS_WIDTH)),
                        addr_read_bus_out      => addr_read_bus((j*ADDRESS_WIDTH + (ADDRESS_WIDTH-
1)) downto (j*ADDRESS_WIDTH)),
                        wr_out  => wr(j downto j)
                );

                U2: RowBankMemory
                port map(
                        clk => clock,
                        raddr => conv_integer(addr_read_bus((j*ADDRESS_WIDTH + (ADDRESS_WIDTH-1))
downto (j*ADDRESS_WIDTH))),
                        waddr => conv_integer(addr_write_bus((j*ADDRESS_WIDTH + (ADDRESS_WIDTH-1))
downto (j*ADDRESS_WIDTH))),
                        data => data_write_bus((j*2*DATA_WIDTH + (2*DATA_WIDTH-1)) downto
j*2*DATA_WIDTH),
                        we => wr(j downto j),
                        q => data_read_bus((j*2*DATA_WIDTH + (2*DATA_WIDTH-1)) downto
j*2*DATA_WIDTH)
                );

                wide_addr_write_out <= local_wr_addr;
                wide_addr_read_out <= local_rd_addr;
                wide_data_out <= local_wr_data;
                wide_wr_out <= local_wr;

                go_control_out <= go_control;
                col_selects_out <= col_selects;


        end generate;




end rtl;



library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RowBankMemory is
```

```
	generic
	(
		DATA_WIDTH : natural := 8;
		ADDR_WIDTH : natural := 8
	);

	port
	(
		clk		: in std_logic;
		raddr	: in natural range 0 to 2**ADDR_WIDTH - 1;
		waddr	: in natural range 0 to 2**ADDR_WIDTH - 1;
		-- 16 bit memory
		data	: in std_logic_vector((2*DATA_WIDTH-1) downto 0);
		q		: out std_logic_vector((2*DATA_WIDTH-1) downto 0);

		we		: in std_logic := '1'

	);

end RowBankMemory;

architecture rtl of RowBankMemory is

	-- Build a 2-D array type for the RAM
	-- 16 bit memory
	subtype word_t is std_logic_vector((2*DATA_WIDTH-1) downto 0);

	type memory_t is array(raddr'high downto 0) of word_t;

	-- Declare the RAM signal.
	signal ram : memory_t;

begin

	process(clk)
	begin
	if(rising_edge(clk)) then
		if(we = '1') then
			ram(waddr) <= data;
		end if;

		-- On a read during a write to the same address, the read will
		-- return the OLD data at the address
		q <= ram(raddr);
	end if;
	end process;

end rtl;




library ieee;
use ieee.std_logic_1164.all;

entity RowMUX is
	generic
	(
		DATA_WIDTH	: natural  :=  8;
		ADDRESS_WIDTH  : natural  :=  8
	);


	port
	(
		-- Input ports
		data_write_bus0	: in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
```

```
               data_write_bus1       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
               data_write_bus2       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
               data_write_bus3       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
               data_write_bus4       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
               data_write_bus5       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
               data_write_bus6       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
               data_write_bus7       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);
               data_write_bus8       : in  std_logic_vector((2*DATA_WIDTH-1) downto 0);

               addr_write_bus0 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus1 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus2 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus3 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus4 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus5 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus6 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus7 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_write_bus8 : in  std_logic_vector((ADDRESS_WIDTH-1) downto 0);

               addr_read_bus0 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus1 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus2 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus3 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus4 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus5 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus6 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus7 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus8 : in   std_logic_vector((ADDRESS_WIDTH-1) downto 0);

               wr0 : in std_logic_vector(0 downto 0);
               wr1 : in std_logic_vector(0 downto 0);
               wr2 : in std_logic_vector(0 downto 0);
               wr3 : in std_logic_vector(0 downto 0);
               wr4 : in std_logic_vector(0 downto 0);
               wr5 : in std_logic_vector(0 downto 0);
               wr6 : in std_logic_vector(0 downto 0);
               wr7 : in std_logic_vector(0 downto 0);
               wr8 : in std_logic_vector(0 downto 0);

               col_sel        : natural;

               -- Output ports
               data_write_bus_out    : out std_logic_vector((2*DATA_WIDTH-1) downto 0);
               addr_write_bus_out    : out std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               addr_read_bus_out     : out std_logic_vector((ADDRESS_WIDTH-1) downto 0);
               wr_out  : out std_logic_vector(0 downto 0)
       );
end RowMUX;


architecture rtl of RowMUX is

begin

       SEL_PROCESS: process (col_sel, data_write_bus0, data_write_bus1, data_write_bus2,
data_write_bus3, data_write_bus4,
                                    data_write_bus5, data_write_bus6, data_write_bus7,
data_write_bus8, addr_write_bus0, addr_write_bus1,
                                    addr_write_bus2, addr_write_bus3, addr_write_bus4,
addr_write_bus5, addr_write_bus6, addr_write_bus7,
                                    addr_write_bus8, addr_read_bus0, addr_read_bus1,
addr_read_bus2, addr_read_bus3, addr_read_bus4,
                                    addr_read_bus5, addr_read_bus6, addr_read_bus7,
addr_read_bus8, wr0, wr1, wr2, wr3, wr4, wr5, wr6, wr7, wr8)
       begin
               case col_sel is
                      when 0  =>
                             data_write_bus_out <= data_write_bus0;
```

```
                              addr_write_bus_out <= addr_write_bus0;
                              addr_read_bus_out <= addr_read_bus0;
                              wr_out <= wr0;
                       when 1  =>
                              data_write_bus_out <= data_write_bus1;
                              addr_write_bus_out <= addr_write_bus1;
                              addr_read_bus_out <= addr_read_bus1;
                              wr_out <= wr1;
                       when 2  =>
                              data_write_bus_out <= data_write_bus2;
                              addr_write_bus_out <= addr_write_bus2;
                              addr_read_bus_out <= addr_read_bus2;
                              wr_out <= wr2;
                       when 3  =>
                              data_write_bus_out <= data_write_bus3;
                              addr_write_bus_out <= addr_write_bus3;
                              addr_read_bus_out <= addr_read_bus3;
                              wr_out <= wr3;
                       when 4  =>
                              data_write_bus_out <= data_write_bus4;
                              addr_write_bus_out <= addr_write_bus4;
                              addr_read_bus_out <= addr_read_bus4;
                              wr_out <= wr4;
                       when 5  =>
                              data_write_bus_out <= data_write_bus5;
                              addr_write_bus_out <= addr_write_bus5;
                              addr_read_bus_out <= addr_read_bus5;
                              wr_out <= wr5;
                       when 6  =>
                              data_write_bus_out <= data_write_bus6;
                              addr_write_bus_out <= addr_write_bus6;
                              addr_read_bus_out <= addr_read_bus6;
                              wr_out <= wr6;
                       when 7  =>
                              data_write_bus_out <= data_write_bus7;
                              addr_write_bus_out <= addr_write_bus7;
                              addr_read_bus_out <= addr_read_bus7;
                              wr_out <= wr7;
                       when 8  =>
                              data_write_bus_out <= data_write_bus8;
                              addr_write_bus_out <= addr_write_bus8;
                              addr_read_bus_out <= addr_read_bus8;
                              wr_out <= wr8;
                       when others  =>
                              data_write_bus_out <= data_write_bus0;
                              addr_write_bus_out <= addr_write_bus0;
                              addr_read_bus_out <= addr_read_bus0;
                              wr_out <= wr0;
               end case;
        end process SEL_PROCESS;


end rtl;


library ieee;
use ieee.std_logic_1164.all;
-- use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;
use ieee.numeric_std.all;


entity TemplateElement is

        generic
        (
               DATA_WIDTH : natural := 8;
               ADDRESS_WIDTH : natural := 8;
```

```
		INITIAL_COL : natural;
		INITIAL_ROW : natural;
		ARRAY_COL	: natural;
		ARRAY_ROW	: natural
	);

	port
	(
		go			: in std_logic := '0';
		coeff		: in std_logic := '0';
		new_row_in	: in std_logic := '0';
		reset		: in std_logic := '0';
		clk			: in std_logic;

		-- 16 bit memory
		a			: in std_logic_vector  ((DATA_WIDTH-1) downto 0);
		b			: in std_logic_vector  ((DATA_WIDTH-1) downto 0);
		data_in		: in std_logic_vector ((2*DATA_WIDTH-1) downto 0);
		data_out		: out std_logic_vector ((2*DATA_WIDTH-1) downto 0);


		addr_in		: out std_logic_vector((ADDRESS_WIDTH-1) downto 0);
		addr_out	: out std_logic_vector((ADDRESS_WIDTH-1) downto 0)
	);

end entity;

architecture rtl of TemplateElement is
	-- Build the state machine
	type state_types is (IDLE, ACC, STEP_DOWN, STEP_DOWN_IDLE, STEP_DOWN_LOAD);

	-- Registers to hold the current state and the next state
	signal present_state, next_state	: state_types := IDLE;
	signal col_counter, prev_col, step_col, step_prev_col	: std_logic_vector(6 downto 
0) := std_logic_vector(to_unsigned(INITIAL_COL, 7));
	signal row_counter, prev_row, step_row, step_prev_row	: std_logic_vector(0 downto 
0) := std_logic_vector(to_unsigned(INITIAL_ROW / 9, 1)); -- conv_std_logic_vector(INITIAL_ROW / 
9, 1);

	-- 16 bit memory
	signal accumulation, step_accum, step_data_out		: 
std_logic_vector((2*DATA_WIDTH-1) downto 0);

	signal go_bit, go_done, step_go_done, load_data, step_load_data	: std_logic;
	signal shift_bit, shift_done, step_shift_done	: std_logic;
	signal step_read_write : std_logic_vector(0 downto 0);

	signal unsigned_a : std_logic_vector((DATA_WIDTH) downto 0);

	signal col_shift_counter, step_col_shift_counter	: std_logic_vector(3 downto 0) := 
"0000";
	-- signal row_shift_counter, next_row_shift_counter  : std_logic_vector(4 downto 0) := 
"0000";

	signal new_col, step_new_col  : std_logic := '0';
	signal down_bit, new_row, step_new_row	: std_logic := '0';


begin

	-- Force positive input to avoid signed operations
	unsigned_a(DATA_WIDTH) <= '0';
	unsigned_a(DATA_WIDTH-1 downto 0) <= a;

	-- Determine what the next state will be, and set the output bits
	process (present_state, new_col, new_row, reset, go_bit, down_bit)
	begin
		case present_state is
```

```
                        when IDLE =>

                                if (go_bit = '1') then
                                        next_state <= ACC;
                                elsif(down_bit = '1') then
                                        next_state <= STEP_DOWN;
                                else
                                        next_state <= IDLE;
                                end if;


                        when ACC =>
                                next_state <= IDLE;



                        when STEP_DOWN =>
                                next_state <= STEP_DOWN_IDLE;


                        when STEP_DOWN_IDLE =>
                                next_state <= STEP_DOWN_LOAD;


                        when STEP_DOWN_LOAD =>
                                next_state <= IDLE;

                end case;

end process;

-- Move to the next state
process(clk, reset)
begin
        if(reset = '1') then
                present_state <= IDLE;
                col_counter <= std_logic_vector(to_unsigned(INITIAL_COL, 7));
                row_counter <= std_logic_vector(to_unsigned(INITIAL_ROW / 9, 1));
                accumulation <= "0000000000000000";
                prev_col <= std_logic_vector(to_unsigned(INITIAL_COL, 7));
                prev_row <= std_logic_vector(to_unsigned(INITIAL_ROW / 9, 1));
                go_done <= '0';
                read_write <= "0";
                load_data <= '0';
                data_out <= "0000000000000000";
                col_shift_counter <= "0000";
                new_row <= '0';
                new_col <= '0';
        elsif(clk'event and clk = '1') then
                present_state <= next_state;
                col_counter <= step_col;
                row_counter <= step_row;
                prev_col <= step_prev_col;
                prev_row <= step_prev_row;
                go_done <= step_go_done;
                read_write <= step_read_write;
                load_data <= step_load_data;
                data_out <= step_accum; --step_data_out;
                accumulation <= step_accum;
                col_shift_counter <= step_col_shift_counter;
                new_row <= step_new_row;
                new_col <= step_new_col;
        end if;
end process;
```

```
        process(present_state, data_in, col_counter, row_counter, coeff, accumulation, b,
unsigned_a, load_data, step_accum, prev_col, prev_row, go_done, col_shift_counter, new_row_in,
new_col, down_bit)
                variable stepping_row : natural range 0 to 8 := 1;
        begin
                case present_state is


                        when IDLE =>
                                step_accum <= accumulation;   -- no accumulation or data to load
                                step_col <= col_counter;      -- keep column counter constant
                                step_row <= row_counter;      -- keep row counter constant
                                step_prev_col <= prev_col;
                                step_prev_row <= prev_row;



                                -- 16 bit memory
                                step_load_data <= load_data;
                                step_read_write <= "0";



                                if(col_shift_counter = std_logic_vector(to_unsigned(ARRAY_COL, 4)))
then
                                        step_new_col <= '1';
                                else
                                        step_new_col <= '0';
                                end if;


                                step_col_shift_counter <= col_shift_counter;




                        when ACC =>

                                if(down_bit = '1') then
                                        if(load_data = '1') then
                                                step_accum <= data_in;
                                                step_load_data <= '0';
                                        else
                                                step_accum <= accumulation;

                                        end if;
                                else


                                        if(load_data = '1') then
                                                if(coeff = '1') then
                                                        step_accum <= data_in - b;
                                                else
                                                        step_accum <= data_in + unsigned_a;
                                                end if;
                                                step_load_data <= '0';
                                        else
                                                if(coeff = '1') then
                                                        step_accum <= accumulation - b;
                                                else
                                                        step_accum <= accumulation + unsigned_a;
                                                end if;

                                        end if;
                                end if;
```

```
                                        if(new_col = '1') then

                                                step_col <= col_counter + 9;

                                                step_row <= row_counter;      -- keep row counter constant
                                                step_prev_col <= col_counter;
                                                step_prev_row <= row_counter;
                                                step_read_write <= "1";
                                                step_load_data <= '1';
                                                if(col_shift_counter = "1000") then
                                                        step_col_shift_counter <= "0000";
                                                else
                                                        step_col_shift_counter <= col_shift_counter + 1;
                                                end if;


                                        else
                                                step_col <= col_counter;
                                                step_row <= row_counter;      -- keep row counter constant
                                                step_prev_col <= prev_col;
                                                step_prev_row <= prev_row;
                                                step_read_write <= "0";
                                                step_load_data <= '0';
                                                if(col_shift_counter = "1000") then
                                                        step_col_shift_counter <= "0000";
                                                else
                                                        step_col_shift_counter <= col_shift_counter + 1;
                                                end if;
                                        end if;

                                        step_new_col <= '0';


                                when STEP_DOWN =>
                                        step_accum <= accumulation;
                                        step_col <= std_logic_vector(to_unsigned(INITIAL_COL,
step_col'length));              -- return column counter to initial position
                                        if(ARRAY_ROW = stepping_row) then
                                                step_row <= row_counter + 1;  -- increment row counter
                                        else
                                                step_row <= row_counter;
                                        end if;
                                        if(stepping_row = 8) then
                                                stepping_row := 0;
                                        else
                                                stepping_row := stepping_row + 1;
                                        end if;
                                        step_prev_col <= col_counter;
                                        step_prev_row <= row_counter;
                                        step_read_write <= "1";
                                        step_load_data <= '1';
                                        step_col_shift_counter <= "0000"; -- col_shift_counter;
                                        step_new_col <= '0';

                                when STEP_DOWN_IDLE =>

                                        step_accum <= accumulation;
                                        step_load_data <= '0';
                                        step_col_shift_counter <= col_shift_counter;
                                        step_col <= col_counter;      -- keep column counter constant
                                        step_row <= row_counter;      -- keep row counter constant
                                        step_prev_col <= prev_col;
                                        step_prev_row <= prev_row;
                                        step_read_write <= "0";
                                        step_load_data <= '0';
                                        step_new_col <= '0';
```

```
                when STEP_DOWN_LOAD =>

                        step_accum <= data_in;
                        step_load_data <= '0';
                        step_col_shift_counter <= col_shift_counter;
                        step_col <= col_counter;      -- keep column counter constant
                        step_row <= row_counter;      -- keep row counter constant
                        step_prev_col <= prev_col;
                        step_prev_row <= prev_row;
                        step_read_write <= "0";
                        step_load_data <= '0';
                        step_new_col <= '0';

        end case;
    end process;

    -------------------------------------------------------------
    -- Internal Control Process
    --
    -- This process modifies the go command from the processor so that
    -- only a single increment can occur per toggle of the go signal
    -- with the goal of eliminating timing issues or gliches where
    -- no accumulation or multiple accumulations of the same data
    -- occurs.
    --
    --
    -- This process drives the following signals:
    --   step_go_done: this sets the go_done signal to the value of
    --                               the go input so that the internal go_bit signal
    --                               is set when an accumulation has not yet occured
    --                       and cleared when the accumulation state is reached
    --                               once.
    --
    -------------------------------------------------------------
    process(present_state, go, go_done, new_row, new_row_in)
    begin
        if(present_state = ACC) then  -- Accumulation state?
                step_go_done <= go;    -- Set go_done to the go bit on next clock cycle
                step_new_row <= new_row;
        elsif(present_state = STEP_DOWN) then
                step_new_row <= new_row_in;
                step_go_done <= go_done;
        else
                step_new_row <= new_row;
                step_go_done <= go_done;      -- Any other state keeps go_done constant
        end if;
    end process;

    down_bit <= new_row xor new_row_in;
    go_bit <= go_done xor go;
    addr_in(7 downto 7) <= row_counter;
    addr_in(6 downto 0) <= col_counter;
    addr_out(7 downto 7) <= prev_row;
    addr_out(6 downto 0) <= prev_col;


end rtl;


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_signed.all;

entity FilterRowController is

    generic
    (
        ROW_INIT : natural := 0
    );

    port
    (
        -- Input ports
        go      : in  std_logic;
        clk     : in  std_logic;

        -- Output ports
        go_cmd  : out std_logic;
        oper    : out std_logic_vector(8 downto 0) := "111000111";
        new_col_out    : out std_logic_vector(8 downto 0);
        new_row_out    : out std_logic_vector(8 downto 0);
        col_sel : out std_logic_vector(3 downto 0)
    );
end FilterRowController;



architecture rtl of FilterRowController is


    subtype Array_Control is std_logic_vector(8 downto 0);
    subtype Col_Sel_Control is std_logic_vector(3 downto 0);

    type Software_Array_Type is array(9 downto 1) of Array_Control;
    type Col_Sel_Software_Type is array(8 downto 0) of Col_Sel_Control;

    signal Oper_Control_LUT : Software_Array_Type;
    signal Col_Control_LUT : Software_Array_Type;
    signal Row_Control_LUT : Software_Array_Type;
    -- signal Col_Sel_Control_LUT : Col_Sel_Software_Type;


begin

    --Oper_Control_LUT(0) <= "000000000";
    Oper_Control_LUT(1) <= "111000111";
    Oper_Control_LUT(2) <= "110001111";
    Oper_Control_LUT(3) <= "100011111";
    Oper_Control_LUT(4) <= "000111111";
    Oper_Control_LUT(5) <= "001111110";
    Oper_Control_LUT(6) <= "011111100";
    Oper_Control_LUT(7) <= "111111000";
    Oper_Control_LUT(8) <= "111110001";
    Oper_Control_LUT(9) <= "111100011";

    --Row_Control_LUT(0) <= "000000000";
    Row_Control_LUT(1) <= "000000001";
    Row_Control_LUT(2) <= "000000011";
    Row_Control_LUT(3) <= "000000111";
    Row_Control_LUT(4) <= "000001111";
    Row_Control_LUT(5) <= "000011111";
    Row_Control_LUT(6) <= "000111111";
    Row_Control_LUT(7) <= "001111111";
    Row_Control_LUT(8) <= "011111111";
    Row_Control_LUT(9) <= "111111111";

    --Col_Control_LUT(0) <= "000000000";
    Col_Control_LUT(1) <= "000000001";
    Col_Control_LUT(2) <= "000000010";
    Col_Control_LUT(3) <= "000000100";
    Col_Control_LUT(4) <= "000001000";
    Col_Control_LUT(5) <= "000010000";
    Col_Control_LUT(6) <= "000100000";
    Col_Control_LUT(7) <= "001000000";
    Col_Control_LUT(8) <= "010000000";
Col_Control_LUT(9) <= "100000000";



process(clk, go)

       variable step_counter : integer range 1 to 9 := 1;
       variable step_down_counter : integer range 0 to 127 := 0;
       variable row_step_counter: integer range 0 to 9 := 0;
       variable col_sel_counter : integer range 0 to 8 := 0;
       variable three_clk : integer range 0 to 2 := 0;
       variable two_clk : integer range 0 to 1 := 1;
       variable go_toggle : std_logic := '0';

begin


       if(clk'event and clk = '1') then

              if(step_down_counter = 127) then
                     oper <= Oper_Control_LUT(1);
                     new_col_out <= Col_Control_LUT(1);
                     new_row_out <= Row_Control_LUT(row_step_counter);
                     col_sel <= conv_std_logic_vector(row_step_counter, 4);
                     if(three_clk = 2) then
                            if(row_step_counter = 9) then
                                   step_down_counter := 0;
                                   row_step_counter := 9;
                                   step_counter := 1;
                                   col_sel_counter := 0;
                            else
                                   row_step_counter := row_step_counter + 1;
                                   three_clk := 0;
                            end if;
                     else
                            three_clk := three_clk + 1;

                     end if;

              else
                     oper <= Oper_Control_LUT(step_counter);
                     new_col_out <= Col_Control_LUT(1);
                     col_sel <= conv_std_logic_vector(col_sel_counter, 4);
                     new_row_out <= Row_Control_LUT(row_step_counter);

                     if(go = '1') then
                            go_toggle := not(go_toggle);
                            go_cmd <= go_toggle;


                            step_down_counter := step_down_counter + 1;

                            if(step_counter = 9) then
                                   step_counter := 1;
                            else
                                   step_counter := step_counter + 1;
                            end if;

                            if(col_sel_counter = 9) then
                                   col_sel_counter := 1;
                            else
                                   col_sel_counter := col_sel_counter + 1;
                            end if;
                            two_clk := 0;

                     else
                            go_cmd <= go_toggle;
                     end if;
```

```
                              end if;

                        end if;

                  end process;

            end rtl;
```