AN ASSESSMENT OF THE SUITABILITY OF RECONFIGURABLE SYSTEMS FOR DIGITAL SIGNAL PROCESSING

Russell J. Petersen

Department of Electrical and Computer Engineering

M. S. Degree, September 27, 1995

ABSTRACT

Digital signal processing (DSP) is an application area requiring intense amounts of I/O and computation, often being applied in areas that require a real-time response. The real-time nature of many DSP systems combined with their intensive computation and data flow requires performance beyond that which can be achieved with a conventional microprocessor. As a result, many custom hardware systems have been designed and developed to provide the required performance. These custom systems, however, have the disadvantage of inflexibility and a high cost of development. The DSP *processor* attempts to overcome the inflexibility and development costs of custom hardware by providing flexibility through software instruction decoding and execution and high performance through custom arithmetic and I/O components. Reconfigurable hardware using FPGAs has also generated interest for use in implementing digital signal processing systems due to its ability to implement custom hardware solutions while still maintaining flexibility through system reprogramming. Using reconfigurable hardware it is hoped that a significant performance improvement can be achieved over the DSP processor without sacrificing system flexibility. This thesis quantifies the ability of reconfigurable hardware using FPGAs to provide a performance improvement over the DSP processor in the area of digital signal processing. A study of multiplier implementation methods and their performance when implemented on FPGAs has been performed for three separate FPGAs. Three common DSP algorithms have also been chosen and implemented on the three FPGAs with their performance being compared to that of equivalent implementations using DSP processor and custom hardware systems.

COMMITTEE APPROVAL: Brad L. Hutchings
Committee Chairman

Brent E. Nelson
Committee Member

Brian D. Jeffs
Committee Member

Wynn C. Stirling
Graduate Coordinator

AN ASSESSMENT OF THE SUITABILITY OF RECONFIGURABLE SYSTEMS FOR DIGITAL SIGNAL PROCESSING

A Thesis
Submitted to the
Department of Electrical and Computer Engineering
Brigham Young University

In Partial Fulfillment
of the Requirements for the Degree
Master of Science

by
Russell J. Petersen
September 27, 1995

This thesis by Russell J. Petersen is accepted in its present form by the Department of Electrical and Computer Engineering of Brigham Young University as satisfying the thesis requirement for the degree of Master of Science.

Brad L. Hutchings
Committee Chairman

Brent E. Nelson
Committee Member

Brian D. Jeffs
Committee Member

Date

Wynn C. Stirling
Graduate Coordinator

ACKNOWLEDGEMENTS

I would like to thank National Semiconductor for their support of this work under ARPA/CSTO contract number DABT63-94-C-0085. I would also like to thank Harry Holt of National Semiconductor for his help in obtaining performance numbers for the CLAy FPGA and Nick Tredennick of Altera Corporation for his generous donation of the RIPP10 boards and software used for several example applications in this thesis.

I would also like to thank Dr. Brad Hutchings for his very helpful insights and suggestions during the course of this project. Paul Graham also deserves thanks for his tireless help with the tools and equipment needed to carry out this work.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

An Assessment of the Suitability of Reconfigurable Systems for Digital Signal Processing

Russell J. Petersen

September 27, 1995

CHAPTER 1

INTRODUCTION

Digital signal processing (DSP) is an area that has been gaining in popularity and general interest for years. It involves the digital storage and manipulation of real world signals such as those found in radar, audio and video applications, control systems, and all other areas where electrical signals are used. It has become a popular alternative to analog techniques of processing signals since, with the signal in digital form, mathematical operations such as multiplication and division can be more accurately applied to manipulate the signal. Digital manipulation of the signal allows not only increased accuracy (due to the nature of digital versus analog computation) but it also allows greater flexibility in the types of operations that can be applied to the signal. An example of a popular and well known application of digital signal processing is the audio processing circuitry used in the compact disc player.

As a result of the potential benefits to be gained from and the possible widespread application of the techniques of digital signal processing, much effort has gone into the development of specialized digital hardware systems that allow the efficient implementation of digital signal processing algorithms. These specialized hardware systems seek to meet the stringent computational and I/O demands that are often placed on hardware by DSP algorithms. Typically, these specialized hardware systems have taken the form of either a programmable processor with specialized functional units or a single application design with limited or no programmability. The purpose of the first approach is to provide application flexibility while still achieving a modest level of performance. The goal of the second approach is to achieve the highest performance level possible by sacrificing flexibility. These two approaches represent the extremes of implementation possibilities and have conflicting goals. That is, system flexibility and maximum performance are generally not harmonious and represent a tradeoff that designers of digital signal processing systems must make. What is needed is an approach that is able to combine the desirable characteristic of system flexibility with the equally important goal of achieving a high level of system performance. This approach is reconfigurable hardware.

1.1 Reconfigurable hardware

Reconfigurable hardware refers to the use of any electronic hardware system that can be changed in structure either statically (between applications) or dynamically (during an application) without the addition of physical hardware elements. It can be implemented using any technology that allows the structure of the hardware to change using only electrical signals. Early examples of reconfigurable hardware can be seen in the application of programmable logic devices (PLDs) and systems incorporating sets of fixed components whose interconnection can be changed by reconfiguring a crossbar or interconnection device.

In the context of this thesis, reconfigurable hardware refers to the use of the Field Programmable Gate Array (FPGA) as the basic hardware element to be reconfigured. FPGAs generally consist of sets of flexible gates, registers, and memories whose function and interconnection are controlled through the loading of SRAMs. A reconfigurable gate could, for example, be made to behave as either an OR or an AND gate depending upon the status of the SRAM configuration bits attached to the gate. FPGAs are able to support both static (reconfiguration between applications) and dynamic (reconfiguration during application execution) reconfiguration since all that is needed to change the hardware is to reload the controlling SRAMs with the appropriate configuration bits.

The advantage of the reconfigurable hardware approach for digital signal processing system implementation is that application flexibility is provided by allowing the actual hardware design to change rather than by adding software-programmable hardware to a design to provide flexibility. That is, to change the application implemented by a reconfigurable hardware system, the system need only be reconfigured through the reloading of the configuration SRAM bits. This allows a completely new hardware design to be implemented. No extra hardware needs to be added to the initial design to provide this flexibility. The flexibility is provided by the reconfigurability of the FPGA itself and is thus utilized by changing the entire design at the basic hardware level. Thus, the designer need not consider the programmable hardware elements and can hence concentrate on an optimized hardware design. This is an advantage since many designs suffer in performance when extra flexibility is achieved through the addition of software-programmable circuitry. The addition of software-programmable circuitry requires extra area on the chip and often increases the critical path of the design. With reconfigurable hardware no such penalty need be paid and hence greater performance can be achieved.

The above argument represents the basis on which reconfigurable hardware has been selected as a potentially attractive method of implementing digital signal processing systems. However, the argument is flawed in that it fails to mention the penalty that must be paid in order to make a reconfigurable system reconfigurable. That is, in the case of the FPGA extra circuitry must be added to the basic gates in order to allow them to be reconfigured and hence provide system flexibility. In practice, for FPGAs this usually results in approximately a $10x$ increase in the number of transistors needed to implement a gate as compared to a VLSI implementation. This makes the resulting system larger and hence slower than a system with no such additions. The question that this thesis seeks to quantitatively answer is whether or not the area and time penalty imposed by the addition of circuitry to make the hardware reconfigurable is greater or less than that paid by the addition of software re-programmable hardware elements. In specific, this thesis seeks to answer the question of whether or not FPGAs are a useful and justifiable alternative for implementing digital signal processing systems. Do they provide a significant performance increase over software-programmable-based methods (in specific, the dsp-processor) while still maintaining the advantage of flexibility? This thesis will show that they do.

1.2 Previous Work

Previous work has focused mainly on specific signal processing applications implemented on FPGAs rather than on the more general topic of the overall suitability of FPGA-based hardware to DSP system implementation. Examples are papers and one thesis discussing one- and two-dimensional filter implementations on various FPGAs such as in [3], [6], [10], [14],[23], and [13]. Other example papers dealing with various DSP algorithm implementations on FPGAs, including video processing applications, can be found in [5], [34] and [15]. [16] investigates the issues associated with the mapping of DSP algorithms to field-programmable technologies but does not focus on final DSP algorithm performance and DSP processor/FPGA performance comparison issues. [12], [36], and [8] discuss and attempt to illustrate some of the potential and the advantages of FPGA-based DSP systems over ASICs and DSP processors but are each limited in scope. [12] sets forth some of the requirements for high-speed DSP and discusses the architecture of a custom field-programmable multiprocessor chip developed by the authors. Most of the paper is spent discussing the custom chip and its associated software. A rough comparison of their custom architecture to FPGAs and DSP processors is also presented with numbers being given for the FPGA/custom chip comparison. [36] begins with a short discussion of the potential advantages of FPGAs over ASICs and DSP processors and presents results from FIR/IIR filter implementations on Xilinx FPGAs as support. The filter implementations, however, obtain performance only roughly equivalent to DSP processor implementations. [8] presents a rough on-paper comparison of FPGA-based DSP systems to parallel DSP boards using multiple DSP processors and an DEC Alpha workstation. The authors conclude that FPGA-based systems give a modest performance improvement for floating-point DSP when using FPGAs coupled with custom multiplication chips. The comparison is very rough however and performance is based only upon the number of multiply-accumulates that are possible per second with no attempt to include factors such as use of parallelism, I/O effects, and control customization. Use of the FPGA as a DSP coprocessing element has also been examined and is illustrated in [33].

This work builds upon much of the previous work in defining more clearly the potential advantages of the FPGA-based reconfigurable hardware approach to digital signal processing and reasons for its success. It also presents the first known work to date to attempt to quantify the performance of FPGA-based DSP systems relative to DSP processor and ASIC-based systems using more than a single application.

1.3 Thesis outline

Chapter two will present more background on the nature of the computations and I/O that are characteristic of digital signal processing algorithms. The advantages and disadvantages of the DSP programmable processor chip and the ASIC-based approaches to implementing digital signal processing algorithms will also be introduced and discussed. These two approaches represent the current major alternatives for implementing DSP algorithms. The reconfigurable hardware approach to DSP algorithm implementation will also be discussed, highlighting its potential advantages over the other two alternatives. Chapter three will then present a study of multiplier implementations on FPGAs discussing the importance of fast multiplication to DSP algorithms and possible implementation alternatives for multipliers. Results of the implementation of several types of multipliers on different FPGA architectures will be presented and their performance will be compared to that of custom silicon multiplier chips. Chapter four presents a set of three example DSP applications and their performance when implemented on FPGAs with comparisons to the performance of equivalent implementations on DSP processors and ASICs. These three applications are used to show the potential of using reconfigurable hardware in DSP applications. Finally, Chapter five will form conclusions and discuss further work to be done in this area along with some discussion of new advances in FPGA technology and its potential impact on the use of reconfigurable hardware for DSP.

CHAPTER 2

BACKGROUND ON DSP COMPUTATION

Digital signal processing (DSP) algorithms involve the manipulation of intense amounts of data and heavy mathematical computation. Seven criteria that are characteristic of DSP algorithms are listed below [7]. DSP algorithms:

- involve immense amounts of data,
- reuse data,
- may have feed back paths such as with IIR filters,
- require fast multiplications,
- often utilize constants,
- involve intensive intermediate data manipulation and address generation, and
- are mostly amenable for parallel computation.

As an example of a characteristic DSP algorithm, consider the discrete Fourier transform (DFT) computed using a fast Fourier transform (FFT) algorithm [32] [18]. The equation for an N-point DFT of a sequence $x[n]$ is:

$$X(k) = \sum_{n=0}^{N-1} x[n] e^{-jk(2\pi/N)n}$$

Calculation of the DFT of a 1024-point sequence (N=1024) using a radix-4 based FFT algorithm requires a total of 5 stages (corresponding to the 5 columns of the FFT). Each of these stages requires the 1024 output points from the previous stage as inputs with the first stage using the original inputs to the algorithm and the last stage producing the outputs of the algorithm. A total of 5120 complex multiplications and 1024 radix-4 butterfly computations are required. Each butterfly computation requires 4 complex additions. The order of data addressing also changes between each column (stage) of the FFT.

In order to continuously compute a 1024-point FFT on an incoming data stream with a sample rate of 40 MHz, the FFT must be computed in no more than 25.6 μs. *This results in a 200 MHz complex multiplication rate.* If we assume that the incoming data is buffered in an input ram with intermediate calculations also being stored in ram a total of 10,240 complex memory accesses are required (assuming that the coefficients used in the FFT are also stored in memory). *This represents a total memory bandwidth of 800 Megabytes per second using 16-bit complex data.* This would require memories with an access time of 5 ns or less (assuming two separate 16-bit wide memories). With such intense I/O and multiplication rates it can be seen that even the calculation of a fairly short FFT such as this can provide a formidable challenge in terms of both I/O and computation rates. Finally, the FFT algorithm characterizes all of the listed DSP algorithm criteria including a feedback path in the connection of the stages of the FFT.

2.1 Real-time aspects of DSP computation

Digital signal processing algorithms are often applied in areas such as control systems and video signal processing which by nature require a real-time response. As a result, stringent time requirements such as those investigated in the DFT computation above are not unusual. In fact, most digital signal processing systems require I/O bandwidths and math computation rates that far exceed the capabilities of even today's highest performing general purpose CPUs. Because of this, much effort has been expended on the development of special-purpose hardware aimed at providing the real-time computational and I/O rates required by DSP systems.

There are many variants of these special purpose hardware systems that have been developed to try and meet the requirements imposed by the above DSP algorithm criteria. All of the variants differ in their implementation but several common features of these architectures can be listed. All current DSP hardware systems employ one or more of the following [7]:

- fast arithmetic units,
- multiple functional units,

- parallel computation,
- pipelined functional units,
- proximity of storage,
- localized data communication, and
- high bandwidth buses.

In this chapter a discussion of the two most popular approaches to these DSP hardware systems: the DSP processor and the ASIC will be given along with an introduction to a third alternative, reconfigurable hardware systems. The architecture, advantages, and disadvantages of each approach will be discussed in order to highlight the benefits of the third alternative of reconfigurable hardware.

2.2 Implementations of digital signal processing systems

2.2.1 The DSP ASIC

This use of an Application Specific Integrated Circuit (ASIC) for a DSP system represents the maximum level of hardware customization that can be applied to an algorithm. It provides the advantage of allowing the designer to customize the solution to the specific algorithm at hand. The system word lengths, bus widths, functional units, levels of pipelining (hence system latency), degrees of parallelism, data storage, and clock rate can all be customized to meet the needs of the application. This allows the designer to eliminate the overhead that is imposed by the inclusion of circuitry to provide increased functionality. The designer may choose to provide less functionality (hence a more application-specific solution) by increasing the level of customization. An example of one such optimization can be seen in the design of an ASIC to perform Finite Impulse Response (FIR) filtering. If the designer has prior knowledge of the types of data to be filtered the system word lengths can be fixed to those required by the given application and the maximum tolerable error. This may allow a shorter word-length and hence less circuitry than an implementation that is required to filter all types of data. Further optimizations may also be able to be made in the clock rate and structure of the functional units if the maximum sample rate of the system is known. The designer may be able to choose a smaller but slower multiply-accumulate (MAC) unit for example so long as the maximum sample rate for the system is still met. This makes the design more limited in application but saves silicon area and cost. Thus, the first advantage of the ASIC-based approach is the ability to include in the design only that which is necessary to implement the desired algorithm.

The second advantage of the ASIC-based approach relates to the first in that the elimination of the excess circuitry needed for increased functionality leads to an increase in performance. Extra circuitry takes up extra area in silicon and often results in an increase in the clock period of a synchronous system or the critical delay path in an asynchronous system. The moral is that the more application-specific the designer can make the system, the better the performance is likely to be. The ASIC allows the designer full flexibility in this area since the uncommitted silicon can be used to implement any structures that the designer desires within the constraints of the available area.

The disadvantages of the ASIC-based approach to DSP system implementation include the high manufacturing cost of the ASIC, the lengthened development cycle, and the resulting inflexibility of the circuit. Custom chips are expensive to fabricate and can generally only be justified when the associated production volumes will be large enough to amortize the initial fabrication costs for the masks over thousands of chips. The development cycle is also longer and costlier due to the increased complexity associated with customizing a complete chip. Some attempts have been made to offset these costs through the use of design libraries containing common components such as multipliers, control cores, etc., but the process still remains prohibitive for applications requiring only a few tens of chips or less. The application must be common enough to allow large production volumes and fixed in its basic structure so as to allow customized, inflexible silicon resources to be used. Despite these efforts, the development times are longer for ASICs than for other alternatives (see DSP processor and reconfigurable hardware discussions below), a penalty which is incurred several times more frequently than for other methods due to the resulting inflexibility of the circuit. Still, the ASIC is the logical choice for applications requiring the maximum in performance and large production volumes.

2.2.2 The DSP Processor

The DSP processor represents nearly the opposite end of the spectrum from the aforementioned ASIC-based approach. The DSP processor sacrifices performance in favor of the flexibility provided by software programmability. It has become popular in the past ten years due to its ability to provide a performance improvement for DSP algorithms over a standard microprocessor while still maintaining the software programmability. The performance improvement is achieved by streamlining the design of the processor to include only those elements specifically needed by DSP algorithms and by optimizing certain elements of the processor for increased performance [26] [27] [9]. In other words, the DSP processor takes the basic architecture of a truly general purpose processor and customizes it to better suit the efficient implementation of DSP algorithms. For example, real-time DSP algorithm implementations have no need of a processor with extensive memory management capabilities such as those needed for virtual memory management. Hence, this circuitry can be optimized out of the design. In a similar way, DSP systems must always carefully balance I/O with computation and therefore many DSP processors have multiple data buses to allow fetching of several operands at once. For example, consider the following multiply-accumulate loop: $Y = Y + A * B$. Both operands A and B could be fetched simultaneously on most DSP processors that allow the operation to proceed in a single clock cycle. A more general purpose CPU such as those found in PC's or workstations would require several clock cycles to fetch the data operands and instructions and to perform the addition and the multiplication.

Many other enhancements have been made to DSP processor architectures that increase their performance on DSP algorithms. The following is a list of some of the most common features of DSP processors with examples of their application:

- single-cycle multiply-accumulate (MAC) units - allows fast computation of inner MAC loops such as in FIR and IIR filters,
- separate instruction and data memories - allows simultaneous fetching of instructions and data operands,

- on-chip multi-ported data and instruction memories to allow not only simultaneous instruction and data access but also quick access,
- high-speed buses and DMA controllers to allow speedy on- and off-chip data movement,
- special purpose addressing modes such as modulo-n addressing for implementation of delay lines and bit-reversed addressing as needed for fast Fourier transforms, and
- zero software overhead looping and single instruction repetition hardware to allow fast computation of tight inner loops such as in FIR and IIR filters.

All of these features perform two basic tasks: (1) reduce the overhead imposed by software-based control and (2) increase performance through specialized functional units. The separate instruction and data memories minimize control overhead, for example, by allowing both an instruction and up to two data operands to be fetched in parallel to minimize the cost of fetching the instruction. The single-cycle MAC units, zero overhead looping, and single-instruction repetition hardware increase performance and reduce control overhead by increasing the speed of critical inner MAC loops. Once the loop instruction and bounds have been set up in the appropriate hardware registers the loop can be performed with no additional software intervention. In other words, the loop is continually executed in hardware rather than through the fetch-decode-execute cycle needed for software execution. Finally, the high speed buses, the on-chip separated data and instruction memories, and the direct memory access (DMA) controllers serve the purpose of helping to balance the computational and I/O demands of DSP algorithms.

These optimizations make the DSP processor perform significantly better than general purpose processors in DSP system implementations. DSP processors are able to achieve as much as an order of magnitude in performance improvement while still maintaining the flexibility of software-based control. However, the penalty paid by the implementation of the software-based control is still great enough that the application of DSP processors have typically been limited to DSP applications involving data sample rates in the audio-rate range. This implies data sample rates of around 44 kHz and below. While a great many popular applications fit into this category (digital audio systems for example) there are many that do not. Video compression and decompression represent just one area that DSP processors are not yet powerful enough to tackle. Even when these processors do evolve to the point that they are capable of the faster sample rates (such as for video) the extensive parallelism that will need to be employed to meet the performance goals will result in large and expensive systems. This is because the hardware to allow the software control must also be replicated in parallel in addition to the functional units. Also, the complexity of the control hardware increases as the options for parallelism increase. Finally, the overhead of the fetch-decode-execute cycle of software remains even as the degree of parallelism increases. What is needed is an approach that can combine the flexibility of the software programmable DSP processor with the performance of the DSP ASIC. This approach is reconfigurable hardware and is discussed next.

2.2.3 Reconfigurable Hardware

As stated in the introduction reconfigurable hardware refers to any electronic hardware system that can be changed in structure either statically or dynamically without adding additional physical hardware elements. Again, in the context of this thesis this implies the use of Field Programmable Gate Arrays (FPGAs). The FPGA allows an electronic hardware system to be implemented which can change its structure through the reprogramming of the FPGA. Use of FPGAs generally imply digital hardware systems although some analog circuit FPGAs have appeared and are finding use. FPGA programming methods include anti-fuse and SRAM-based technologies although only the latter allow true reconfigurability since anti-fuse FPGAs can only be programmed once. For the benefit of readers not familiar with FPGAs the basic architecture of the FPGAs used in the study presented in this thesis will now be discussed. Following this section the potential advantages and disadvantages of FPGAs as reconfigurable hardware will be discussed in relation to their use in digital signal processing systems.

FPGA Architecture

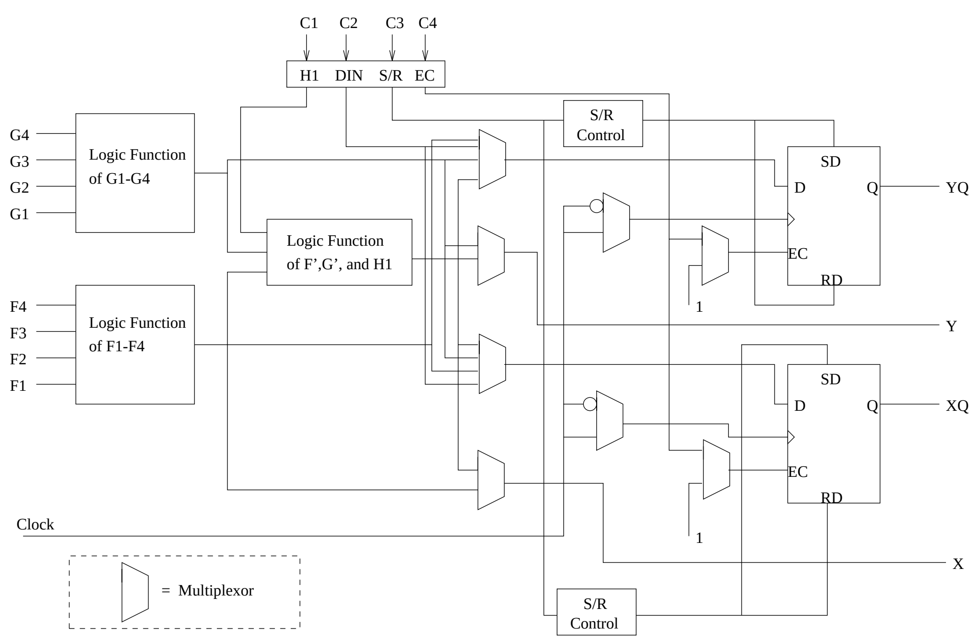
A very basic definition of a digital FPGA would be an integrated circuit consisting of a set of gates such as ANDs, ORs, XORs, etc., latches, and registers (D-flipflops), whose interconnection and function can be altered through the reprogramming of configuration bits in the FPGA. The main distinction to be made in the architecture is whether it is coarse-grained (ie: large basic functional units) or fine-grained (small basic functional units). Coarse-grained architectures have the potential advantage of reducing the amount of interconnection needed since the basic functional units to be interconnected are quite large and powerful. However, they can also result in poor device utilization since the architecture of the application may not match well with the functionality provided by the basic functional units, hence necessitating the use of a larger number of the basic units.

The three FPGAs used in this thesis were the Xilinx 4010, the Altera Flex8000 81188, and the National Semiconductor CLAy31 [37] [1] [31]. The first two are characterized as medium-grained architectures while the third FPGA is a fine-grained architecture. The basic functional units of the Xilinx and Altera FPGAs consist of look-up tables (LUTs), flipflops, fast-carry logic, and control circuitry. The Altera FPGAs also contain cascade logic to allow several basic functional units to be cascaded together to form larger LUTs as would be needed in a large fan-in AND gate, for example. The Xilinx FPGAs provide a similar feature by adding wide edge decoders along the edges of the device. The LUTs in both of the FPGAs can be used to produce any function of 4-input variables. The fast carry logic is used to form high speed arithmetic elements such as fast ripple-carry adders while the cascade logic allows large fan-in input gates such as 8-input AND gates. The main difference between the basic Xilinx logic element (a configurable logic block (CLB)) and the basic Altera logic element (LE) is that the Xilinx CLB contains two 4-input LUTs, two flipflops, and two outputs, while each Altera LE contains only 1 4-input LUT, 1 flipflop, and 1 output. The RAM in the LUTs of the Xilinx parts can also be used to form distributed on-chip RAM instead of being used as function generators. The individual LUT RAMs can be combined together to form larger on-chip memories which proves to be a powerful tool useful in distributed-arithmetic multiplication as will be seen in Chapter 3.

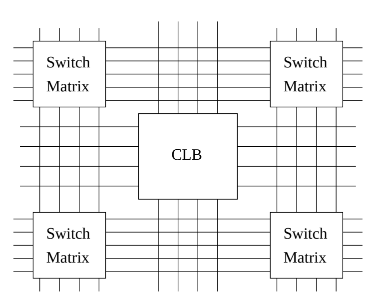
The interconnection of the Xilinx and Altera CLBs also differ in that the Xilinx CLBs are connected to each other through an x-y grid of wires and switch matrices while the Altera LEs are first grouped into sets of 8 LEs (LABs) which are then connected to each other through an x-y grid of row and column interconnects. A Xilinx 4010 has 400 of the CLBs while an Altera 81188 has 1008 LEs. Thus, in terms of logic these Xilinx and Altera parts are nearly equivalent since the Altera LE is roughly equal to $\frac{1}{2}$ of a Xilinx CLB. Figures 2.1 and 2.2 show the basic logic elements and their basic interconnections for the Xilinx and Altera parts respectively.

The National Semiconductor Configurable Logic Array (CLAy) CLAy31 architecture was left out of the discussion above since its architecture differs significantly from that of the Xilinx and Altera FPGAs. The CLAy31 consists of an array of 3136 logic cells arranged in a 56x56 grid. Each cell, rather than containing a LUT as with the Xilinx and Altera parts, contains a set of logic which can be reconfigured to one of 44 different logic and routing functions. The concept behind the cell organization is to provide a smaller cell with the most commonly occuring logic combinations rather than allowing any function of n variables as with the LUT-based approach. In addition to the logic configurations, each cell can be configured to be used as routing resources. This is necessary since the CLAy architecture contains significantly less global routing resources than the Xilinx and Altera parts. The concept is that local routing is the most common type of interconnect and hence it is the most important to support. Figure 2.3 shows the cell structure and interconnections for the CLAy architecture.

In practice, even though the CLAy architecture has many more cells than the Xilinx and Altera parts, cell utilization for logic implementation is generally much less due to the need to use many cells for routing. As much as 50-70% of the cells can end up being used for routing in a CLAy design. An advantage of the CLAy architecture, however, that is not possible with the Xilinx and Altera parts is its ability to be partially reconfigured. The CLAy FPGA can have part of its array reconfigured while the rest of the array remains intact. This provides the possibility of hiding reconfiguration since part of the array may remain operational while the rest of the array is being reconfigured. This is not possible with the Altera and Xilinx parts since the entire array must be reconfigured at once.



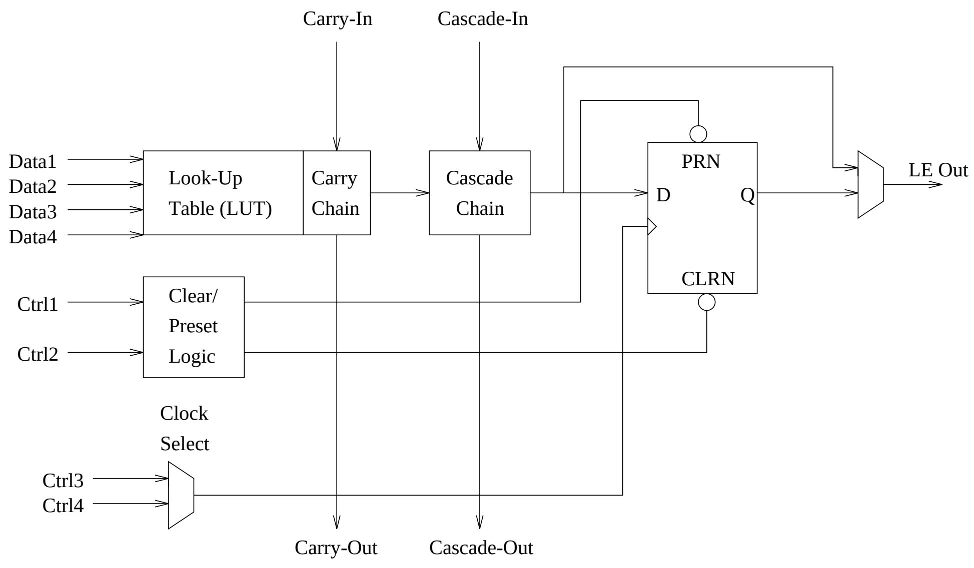
Basic XC4000 CLB Block Diagram



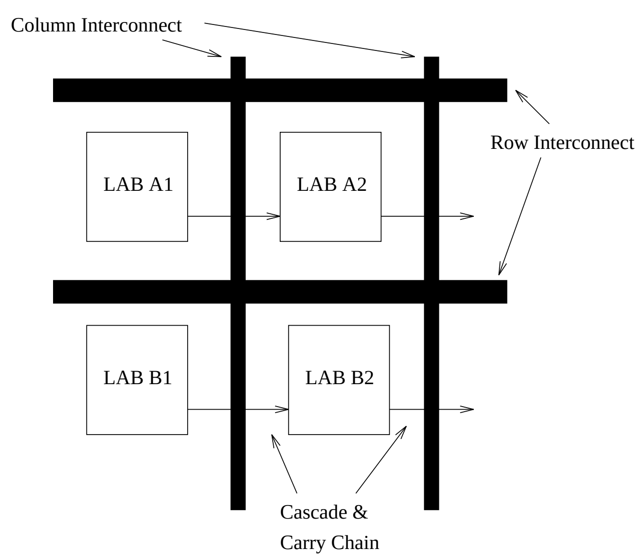
Basic XC4000 Interconnection Block Diagram

Note: Adapted from figures 1 and 9 of Xilinx 1994 Programmable Logic Data Book

Figure 2.1: Xilinx 4000 Series Architecture



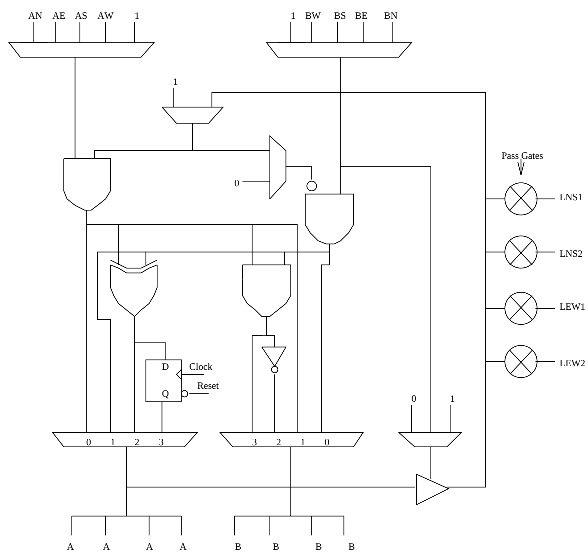
Flex 8000 Logic Element (LE)



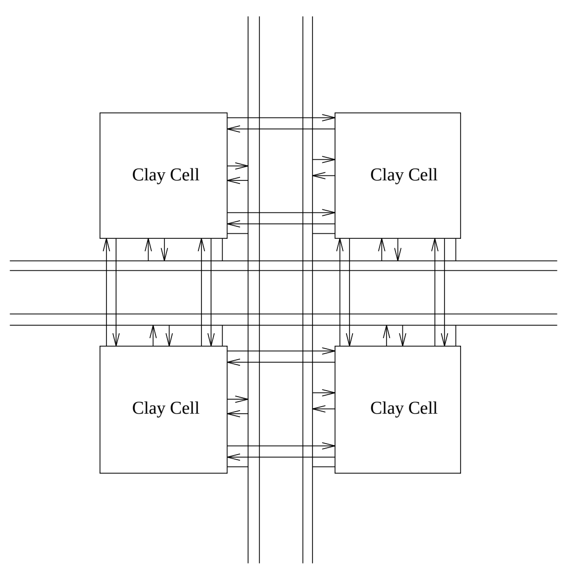
Flex 8000 LAB Interconnections

Note: Adapted from figures 2 and 9 of App Note 40 of May 1994 Altera Flex 8000 Handbook

Figure 2.2: Altera Flex8000 Architecture



CLAy Cell Structure



CLAy Cell Interconnections

Figure 2.3: National Semiconductor CLAy Architecture

In addition to the basic functional units and routing mentioned above all of the FPGAs have additional cells along their perimeter which differ slightly to provide I/O for the FPGA. All of the FPGAs support both registered inputs and outputs with tri-state buffers available on the outputs.

2.2.4 Advantages/Disadvantages of FPGAs and Reconfigurable Hardware

As stated above, the main advantage FPGAs provide for implementing digital signal processing systems is the flexibility that is maintained while obtaining a significant performance improvement over DSP processor-based systems. The FPGA is able to implement in actual hardware what the DSP processor would implement in software. This is an advantage since the instruction fetch-decode-execute cycle is bypassed and hence the time and circuitry needed to implement it can be used to advantage elsewhere. The control is simply hardwired into the design on the FPGA rather than using complex software programmable control. One disadvantage, however, of the FPGA/reconfigurable hardware approach is the time to develop and implement the algorithm. It takes more time to design and test hardware than it does to write a piece of software. This is because hardware generally requires the designer to deal with the algorithm at a lower level and it also requires the designer to consider the timing of the algorithm and its hardware implementation. Software is generally written at a higher level (although most DSP processors are still programmed in assembly language to obtain reasonable performance levels) and is inherently sequential in nature. The designer need not worry as much about *what* happens *when* because the code is executed sequentially. This last advantage of DSP processors is negated however when working with multiple DSP processors connected in parallel. In this case the designer again has to worry about possible timing and synchronization problems between the separate processors.

An additional advantage of reconfigurable hardware is the ability to change the hardware dynamically. In other words, the system to be implemented can be larger than the available reconfigurable hardware if the algorithm is broken up into several time-independent stages. The stages can then be successively implemented by reconfiguring the hardware in between the stages. This is an advantage over an ASIC-based approach where the silicon cannot be reused but must remain fixed. This ability to reconfigure the hardware can also allow the hardware to be even more application specific than a corresponding ASIC. The hardware can be configured to implement exactly what is needed at a particular time by a particular algorithm. As an example, in an FFT there are several addressing sequences that must be generated for the successive columns of the FFT. If one ASIC chip is to be used to compute all the columns of the FFT it must have addressing circuitry that is flexible enough to produce all of the required address sequences for the successive columns. This flexibility takes extra space on the chip and can result in a slower address generator. With the reconfigurable hardware approach the address generator can be reconfigured between columns to suit the changing addressing needs of the successive columns of the FFT. This can thus result in a smaller and faster set of addressing circuitry. This approach is applicable to the FFT implementation discussed in Chapter 4.

A point of performance central to the acceptance of the FPGA as a DSP system target technology, however, is the ability of the FPGA to perform mathematical computations quickly. Multiplication is central to the implementation of nearly all DSP algorithms. As seen with the FFT example at the beginning of this chapter the multiplication rates required are often extremely high. Hence, in order for the FPGA to be successful in DSP it must be able to implement fast and fairly compact multipliers. The ability to implement a reasonably fast multiplier in a compact space will allow extensive use of parallelism to achieve the high multiplication performance levels required by DSP algorithms. The following chapter is an investigation of the ability of the FPGA to implement multipliers. Several types of multipliers have been implemented on the three FPGAs discussed above and their performance has been tabulated for comparison.

CHAPTER 3

FPGA-BASED MULTIPLICATION

In this chapter an evaluation of the implementation of multipliers on the three FPGAs chosen for this thesis will be discussed. Several different types of multipliers have been implemented (both signed and unsigned) which fit into the basic categories of parallel and bit-serial multipliers. Many other types of multiplier architectures are possible but those considered here represent the most common forms and can serve as a reference from which the possible multiplication performance of FPGAs can be judged. 8- and 16-bits of word-length precision were chosen for implementing each of the multipliers since 8-bit multipliers are the most common in video DSP algorithms while 16-bit multipliers are common in most fixed-point DSP processors. Only fixed-point multipliers have been considered since floating-point multiplier implementations on FPGAs are generally considered to be too large in area and too slow. The fixed-point multiplier represents the current best possibility for fast multiplication on the FPGA. Work showing possible implementations of floating-point arithmetic on Xilinx FPGAs can be found in [35]. All of the implemented multipliers have been tested for speed and area with their results tabulated for comparison. In addition, the performance of the FPGA-based multipliers will be compared to that of custom multiplication chips.

3.1 Multiplication architectures

When choosing an architecture for a hardware digital multiplier two basic alternatives are available: parallel and bit-serial [22] [29] [25]. In between these two choices there exists a full range of combinations of parallel and serial architectures [30] that can be considered neither fully parallel nor fully serial but rather are a combination of the two techniques such as with a nibble-serial multiplier. Only the two extremes of fully parallel and fully bit-serial multipliers have been implemented and tested in this thesis in order to obtain performance numbers for the two extremes. However, many combination parallel and serial multipliers could also be designed as a compromise between the two techniques, hence achieving some of the advantages of both.

3.1.1 Parallel multipliers

Figure 3.1 shows a basic block diagram for a parallel multiplication array with its associated computations and multiplication time. The multiplication time is dependent upon whether or not the multiplication array is fully combinational or pipelined with registers to hold intermediate results. The key feature that distinguishes the architecture as being fully parallel is that all of the NxM (N-bit multiplier, M-bit multiplicand) partial-product terms are computed simultaneously even though their addition may occur in pipelined stages. The size of the array is generally dependent to a slight degree upon whether the multiplier is a signed (fully two's complement for the purposes of this thesis) or unsigned multiplier. A signed parallel multiplier will require a slight amount of extra circuitry to handle the signs of the multiplicand and multiplier. The overall size of the array is, however, primarily determined by the word sizes of the multiplier and multiplicand. A fully parallel 8x8 multiplier will, for example, take only approximately $\frac{1}{4}$ of the area of a 16x16 equivalent multiplier since there are 64 partial-products in the 8x8 and 256 partial-products in the 16x16 multiplier. The main advantage of the fully parallel-array is speed. Since all of the partial-products are computed and added in parallel, the parallel multiplier generally gives the best multiplication rates. The disadvantage, however, is the large amount of circuitry required to implement the multiplier.

3.1.2 Bit-serial multipliers

Figure 3.2 shows the architecture of a basic bit-serial-parallel multiplier. This multiplier is not completely serial since the multiplicand (B_{in}) is brought in in parallel. However, simple modifications can be made to the multiplier if a serial multiplicand is desired [17]. These modifications result in only a slight increase in the overall multiplier size.

The bit-serial adder is also shown in Figure 3.2 which is the basic building block of bit-serial arithmetic elements, and hence the bit-serial multiplier. The

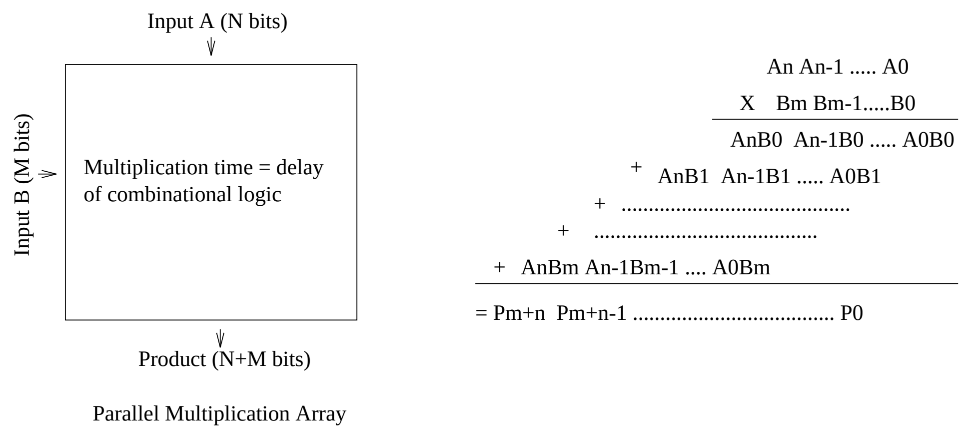


Figure 3.1: Parallel Multiplier Architecture

multiplication time for a bit-serial multiplier is $N + M$ bit times to produce an $(N + M)$-bit product where N is the number of bits in the multiplier and M is the number of bits in the multiplicand. The symbolic multiplication of two 3-bit numbers (A and B) is also shown in Figure 3.2 along with the first few product bits as would be produced by a 3-bit bit-serial multiplier multiplying A and B. As can be seen the bit-serial multiplier implements a shift-and-add method of multiplication using a string of bit-serial adders.

The main advantage of the bit-serial approach is the reduction in size of the multiplier. For an NxN-bit multiplication a bit-serial multiplier can be $\frac{1}{N}$th the size or less of an equivalent parallel implementation. This allows either more multiplication parallelism to be implemented in a DSP system or more space to be devoted to other circuitry. The disadvantage of the bit-serial approach is the time it takes to perform the multiply. To obtain the entire $2N$-bit product, $2N$ clock cycles or bit-times are needed. This often leads one to believe that the bit-serial approach is thus $2N$ times slower than an equivalent NxN-bit parallel approach but this is not true. The bit-times (clock cycles for synchronous multipliers) for a bit-serial multiplier are often much shorter than for a parallel multiplier due to the reduced size of the multiplier and hence reduced propagation path of the signals. This results in a bit-serial approach commonly achieving multiplication rates two to four times smaller than an equivalent parallel approach with the bit-serial approach

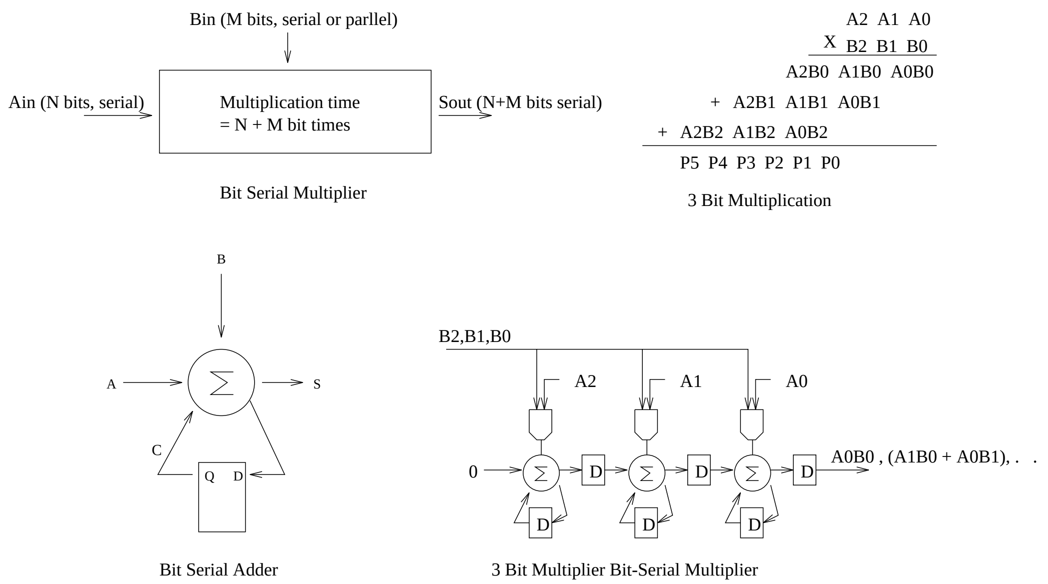


Figure 3.2: Bit-Serial Multiplier Architecture

even sometimes exceeding the parallel approach in some circumstances.

An additional disadvantage of the bit-serial approach is the difference between a signed (fully two's complement) and unsigned multiplier. The signed bit-serial multiplier takes extra control circuitry to handle the sign extension of the multiplicand and intermediate partial-product results which on average results in a doubling of the multiplier size. The actual size difference depends on the multiplier implementation method but it is nearly always significant.

3.2 FPGA multiplier implementation results

Table 3.1 lists results for several different multipliers implemented on the three FPGAs. The table is grouped according to the FPGA used and the type of multiplier implemented. Most multipliers have results listed for both signed and unsigned versions of the multiplier.

The table clearly indicates that parallel multipliers are faster than equivalent bit-serial versions. The speed numbers listed for the bit-serial multipliers are the clock-speed (bit-rate) of the multiplier and the effective multiplication rate (clock-speed/$2N$ where N = precision of the multiplier). The parallel versions of

Table 3.1: FPGA Multiplier Performance Results

Type of Multiplier	# CLB/LC's	% of FPGA	Mult. Speed
Altera 81188 Parallel Multipliers			
8-bit unsigned fast-adder	133	13	14.8 MHz
8-bit signed fast-adder	150	14	12.8 MHz
8-bit unsigned synthesis	129	12	7 MHz
8-bit signed synthesis	135	13	6.84 MHz
8-bit signed complex synthesis	584	57	5.86 MHz
16-bit unsigned fast-adder	645	63	3.34 MHz
16-bit unsigned synthesis	519	51	3.66 MHz
16-bit signed synthesis	535	53	3.4 MHz
Altera 81188 Bit-Serial Multipliers			
8-bit unsigned	29	3	84.03/5.25 MHz
8-bit signed	91	9	69/4.6 MHz
16-bit unsigned	61	7	68.49/2.14 MHz
16-bit signed	186	18	64/2 MHz
National Semiconductor CLAy31 Parallel Multipliers			
8-bit unsigned	329	11	7.9 MHz
8-bit signed	338	11	7.2 MHz
16-bit unsigned	1425	45	3.6 MHz
16-bit signed	1446	46	3.53 MHz
National Semiconductor CLAy31 Bit-Serial Multipliers			
8-bit unsigned	48	1.5	32.2/2.01 MHz
8-bit signed	48	1.5	32.2/2.01 MHz
16-bit unsigned	96	3	29.2/.91 MHz
16-bit signed	96	3	29.2/.91 MHz
National Semiconductor CLAy31 Parallel Constant Multipliers			
8-bit unsigned ROM	846	27	12.1 MHz
Xilinx 4010 Parallel Multipliers			
8-bit unsigned	64	16	8.54 MHz
16-bit signed	259	65	4.35 MHz
8-bit unsigned synthesis	61	15	9 MHz
8-bit signed synthesis	61	15	8 Mhz
8-bit signed complex synthesis	266	66	7.3 MHz
16-bit unsigned synthesis	242	60	3.8 MHz
16-bit signed synthesis	250	63	3.7 MHz
Xilinx 4010 Bit-Serial Multipliers			
8-bit unsigned	17	4	73.1/4.6 MHz
8-bit signed	32	8	52/3.3 MHz
16-bit unsigned	33	8	62/1.9 MHz
16-bit signed	64	16	50/1.6 MHz
Xilinx 4010 Parallel Constant Multipliers			
8-bit unsigned ROM	22	5.5	21.7 MHz
16-bit unsigned ROM	84	21	11.36 MHz
8-bit unsigned RAM	39	9.75	17.86 MHz
16-bit unsigned RAM	117	29.3	10.4 MHz

the multipliers obtain an average speed increase of 2.5 over equivalent bit-serial versions but do so with an average area (size) increase factor of 6.5. The 16-bit versions experienced a larger size increase with an average increase factor of 9 compared to 4 for the 8-bit multipliers. This is due to the fact that the size of a parallel multiplier increases with the square of the increase in word precision whereas bit-serial multipliers increase linearly in size with increasing word-length.

Also illustrated by the table results is the large difference in signed and unsigned bit-serial multiplier sizes as compared to the size difference between signed and unsigned parallel multipliers. Parallel multipliers differ with an average size increase of 5% from unsigned to signed versions while bit-serial multipliers differ by an average increase of 150% in going from an unsigned to a signed multiplier. This is due to the difference in the control needed to implement a bit-serial signed multiplier as opposed to a bit-serial unsigned multiplier. A near doubling of the control registers is needed in a bit-serial multiplier which for an FPGA generally results in a near doubling of the size of the multiplier. This is because the basic structure of an unsigned bit-serial multiplier maps neatly to the functionality provided by the Altera and Xilinx basic functional blocks (CLBs/LEs) while a signed version does not. This effect is not experienced by the CLAy architecture whose bit-serial multipliers remain the same size due to the use of compact pre-complementers on the inputs and outputs of the multipliers to achieve signed operation.

3.2.1 Multipliers listed in the table

The architectures used for the multipliers implemented on the FPGAs were common parallel multiplier arrays such as the Baugh-Wooley two's complement array multiplier [22] [25] and straightforward implementations of pipelined versions of the bit-serial multiplier shown in Figure 3.2 [29]. In addition, a few versions of parallel multipliers that take advantage of the special logic features available on the Altera and Xilinx FPGAs were created. These are intended to represent near the absolute maximum possible multiplier performance that can be achieved with these current FPGAs. In specific, constant versions of parallel multipliers and multipliers utilizing the fast carry-logic were produced. These specific customizations will be discussed below. Equivalent multipliers were not implemented using bit-serial techniques due to the restricted structure of a bit-serial multiplier. That is, use of a constant (multiplicand or multiplier is held constant) in a bit-serial multiplier results in almost no difference in the structure of the multiplier and hence its performance. [1] In addition, the fast carry-logic of the Xilinx and Altera FPGAs is not able to be used by the bit-serial multipliers since its purpose is to speed up operations such as ripple-carry parallel adders. Thus, there is considerably more freedom in the architectural changes that can be easily applied to a parallel multiplier than to a bit-serial multiplier. This section discusses in more detail the types of multipliers implemented, what they illustrate, and why they were chosen.

Synthesized multipliers

Several of the multipliers listed in the table have the label synthesis attached. This label indicates that the multipliers were created by synthesizing simple high-level hardware language (VHDL) design statements such as

$$z <= a * b;$$

These multipliers were included so as to allow a comparison between hand-placed multipliers using schematics and high-level language designed multipliers. This comparison is important since it is desired that much FPGA-based design (including DSP system design) take place at a higher level than schematic capture so as to greatly decrease the development time. Thus, it is important to understand how a multiplier synthesized with a high-level language compares in performance to one designed by hand. The table results show that the synthesized multipliers performed very favorably as shown in the Xilinx 4010 parallel multiplier table section. The 8- and 16-bit unsigned and signed array multipliers listed first were designed with schematics and were hand placed onto the Xilinx FPGA. However,

[1] This is not entirely true in that the size of the multiplier is reduced by the appearance of zeros in the constant. However, the speed of the multiplier remains the same for a synchronous bit-serial multiplier since the clock speed is determined by the slowest path. Hence, the slight decrease in size is not considered important enough to warrant the use of special constant bit-serial multipliers when mapping to FPGAs.

their performance was nearly identical in terms of both speed and area required to the multipliers synthesized from VHDL.

In addition to the standard multipliers a complex 8-bit multiplier was also synthesized and routed on the Altera and Xilinx FPGAs to show the possibility of implementing complex-valued logic. No bit-serial multipliers were designed using VHDL since the only way to specify the use of a bit-serial approach in VHDL is to design the multiplier structurally which is equivalent to a multiplier designed using schematic capture.

Fast carry-logic based parallel multipliers

The Altera 81188-based multipliers labeled *fast-adder* refer to the use of the fast carry-logic available on the Altera FPGAs to make fast ripple-carry adders. These adders are then used to build fast multipliers by using the adders to add the successive partial-product rows. Figure 3.3 shows the block diagram of the design for an 8x8 bit parallel multiplier using this technique. Each of the adders shown is implemented using fast carry-logic. This technique results in multipliers that are approximately twice as fast on the FPGAs as those multipliers not implemented with special logic. The disadvantage of this approach is the resulting difficulty that arises with the placement of the multiplier onto the FPGA. The multipliers make extensive use of the special carry-logic of the FPGA which adds an additional constraint to the FPGA router. This results in the router only being able to place three of the 8-bit multipliers on a 81188 FPGA even though they only utilize 13% of the total FPGA resources each. Up to 5 8-bit multipliers will fit on an 81188 FPGA when not using the special-logic. A 16-bit unsigned version of the fast-adder multiplier was also constructed but not all of the fast carry-logic chains could be routed which resulted in performance less than that of equivalent multipliers synthesized from VHDL.

Equivalent designs were not implemented on the Xilinx parts even though they possess fast carry-logic due to the lack of readily accessible macros for arbitrarily-sized fast adders in the employed schematic capture library. Thus, the decision to not implement the fast-adder based multipliers on Xilinx parts was due to design time considerations and not the ability of the FPGA to implement the multipliers.

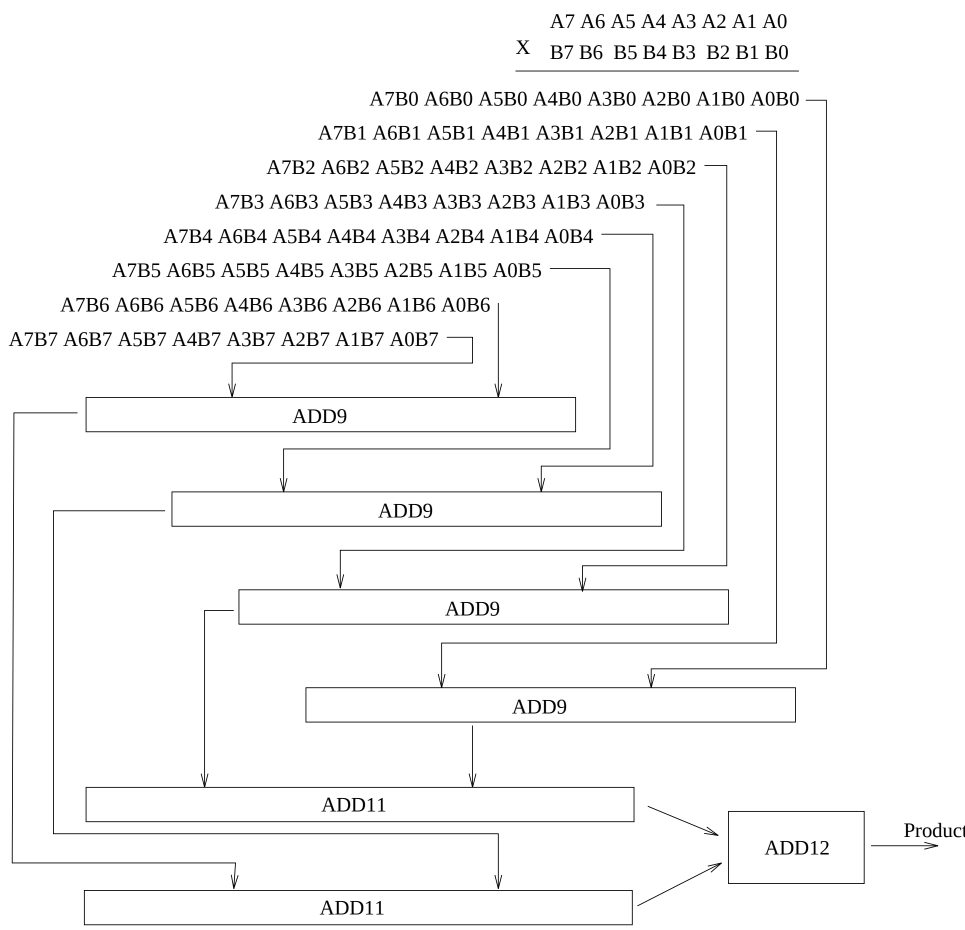


Figure 3.3: 8-Bit Multiplier Using Successive Add Stages

Constant multipliers

The use of constants (constant multiplicand) in multiplication can significantly reduce the size of a parallel multiplier array. This is because the presence of zeros in the constant can result in the elimination of many partial-product terms in the multiplication array. This technique is especially useful in DSP systems since many of the multiplications to be performed can be specified in terms of constant multipliers. For example, with an FIR filter each tap of the filter can be implemented using a multiplier with a constant tap-coefficient. The exact reduction in size and hence speed increase that results through the use of a constant depends upon the value of the constant (ie: number of zeros present in its binary representation). In Chapter 4 the discussion of the video convolution algorithm illustrates example reductions that occur when the multiplicand is held constant in 8x8 bit fully parallel multiplier arrays.

Distributed-arithmetic approach to constant multiplication

The use of constants in multiplication also makes available another technique that can result in a significant multiplier performance increase. This technique is called the *distributed-arithmetic* approach to multiplication and can be best implemented by the Xilinx FPGAs due to their ability to provide small blocks of distributed RAM/ROM to be used as partial-product lookup tables. The RAMs and ROMs are created using the RAM of the lookup tables in the CLB function generators. The Altera Flex8000 parts are ideally also capable of implementing ROMs efficiently since their LE architecture is quite similar to the Xilinx CLB architecture. However, in order to create a ROM with the Altera parts an individual logic equation must be written for each LE in order to store a bit. The outputs of these equations can then be combined to create the ROM. The result is a difficult and time-consuming process not attempted in this thesis. Finally, the CLAy architecture and software allows the formation of ROMs but with less efficiency than with the Xilinx parts as is shown by the constant multiplier results for the Xilinx and CLAy parts in Table 3.1.

The distributed-arithmetic approach to multiplication relies upon the ability to easily pre-compute all of the possible products of a multiplication when one of the values is held constant. This is also possible when neither of the values is held constant but the resulting number of possible products becomes quickly far too large to be able to efficiently handle. For example, an 8x8 bit multiplier has 65,536 possible 16-bit products. The number of products then quadruples with each additional bit of precision in the inputs. Thus, it is only with constant multipliers that the number of products can be limited to a more reasonable number. The question becomes how to store and then output the appropriate product for a given input.

One possible solution for an 8x8 bit multiplier with one input held constant is to simply store all of the 256 possible 16-bit products in a RAM- or ROM-based lookup table and then use the other 8-bit input as the address applied to the table. The output of the table is the proper 16-bit product. This requires 512 bytes of memory storage plus the decoding circuitry. Another possibility is to break up the input into two nibbles (4 bits) and then use each nibble as the address applied to two separate 12-bit wide, 16-location lookup tables. Two separate 16x12-bit tables are required since each of the nibbles produces 16 possible 12-bit partial products. The partial product outputs of each table are then weighted appropriately and added to produce the product. This method requires 48 bytes of memory storage, decoding circuitry, and a 12-bit adder. Thus, the memory is reduced from 512 bytes to 48 bytes (a tenfold reduction) with the addition of only a 12-bit adder. The method is illustrated in Figure 3.4. The partial-product produced by the high-order nibble of the input word is shifted by 4 bits to the left (a weighting factor of 16) and added to the partial-product produced by the lower-order nibble of the input word to produce the 16-bit output.

Although there are many possible ways of breaking up the non-constant input and applying it to lookup tables (LUTs), the architecture of the target technology influences the choice greatly. The Xilinx CLBs each provide a LUT which can be used to form two 16x1 bit memories or a single 32x1 bit memory. These individual RAMs can then be combined with the LUT RAM from other CLBs to form larger RAMs/LUTs. A convenient split of the memories is to use the 16x1 bit memory configuration. Then, the input to a constant multiplier is simply split into multiple nibble-wide sections which are applied to separate 16-location LUTs. The width of each LUT is determined by the width of the

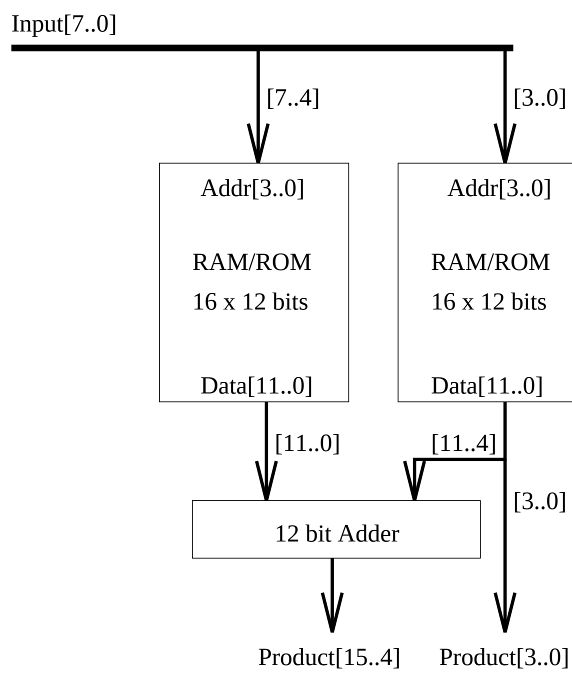


Figure 3.4: 8-Bit Constant Unsigned Multiplier Using Distributed-Arithmetic

multiplication constant. For example, an 8-bit constant produces LUTs with 12-bit entries while a 12-bit constant produces LUTs with 16-bit entries. This is because the multiplication of a N-bit multiplicand by an M-bit multiplier produces an N+M bit product. This division of the CLB RAMs works well for multipliers with inputs which are an even number of nibbles wide. Hence, 8- and 16-bit multipliers are a good fit for this division.

Implementing an 8x8 multiplier (8-bit input with 8-bit constant value) using the above technique in a Xilinx FPGA [11] a total of 384 bits of storage are required along with a 12-bit adder using the fast carry-logic available in the Xilinx FPGAs. This results in a minimum of 12 CLBs for the data storage and approximately 12 CLBs for the adder for a total of 24 CLBs. The actual number of CLBs (area) required is dependent upon optimizations that the place-and-route software is able to make and can be seen to be slightly less (22 CLBs) for the ROM-based 8x8 multiplier in Table 3.1. Adders using the fast carry-logic of the device are used since 90% of the multiplication time is spent on the partial-product addition. The read-time of the RAMs/ROMs on the Xilinx FPGA is on the order of 6 ns and is less significant. If adders not based on the fast carry-logic are used the multiplication time will nearly triple and the adder size will double since only 1 adder bit can fit in a CLB without the use of the carry-logic. Only 1 bit can be placed in a CLB since only two normal (non special-logic) outputs are available per CLB which must both be used to produce the sum and carry outputs whereas before they could both be used as sum outputs. The difference in size and speed between the RAM- and ROM-based versions listed in the table is due to the elimination of the additional inputs used for loading the RAM on the ROM version and the associated optimizations that the place-and-route software can make. The advantage of the RAM-based versions is the ability to change the constant value of the multiplier without reconfiguring the device by reloading the RAMs with the proper partial-products. Such a feature could be useful for an application such as an adaptive filter since it is much faster to reload the RAMs than it is to reprogram the FPGAs. Only unsigned constant Xilinx multipliers are listed in the table but signed versions of the multipliers can also easily be built by sign-extending the partial products and the input to the multiplier. Further performance increases can be achieved through pipelining of the distributed-arithmetic multipliers by placing registers on the inputs and outputs of the memories as well as the adders. Table 3.2 lists the performance possible using this optimization. The pipelined speed is determined by the speed of the slowest element, which is the largest adder in the system [11].

Results for an 8x8-bit constant multiplier based on distributed arithmetic and implemented on the CLAy FPGA also appear in Table 3.1. The design software reported a cell utilization of 27% but actual layout suggests that only two of the 8x8-bit multipliers can be placed onto the FPGA simultaneously as compared to over ten for the Xilinx FPGA. In addition, the Xilinx-based 8x8-bit ROM multiplier performed 80% faster than the CLAy based ROM multiplier.

3.2.2 Comparisons to custom multiplier chips

In order to understand the performance level achieved by the FPGA-based multipliers it is important to provide a reference to measure from. This

Table 3.2: Pipelined Xilinx 4010 Parallel Constant Multiplier Performance

Type of Multiplier	**# CLB/LC's**	**% of FPGA**	**Mult. Speed**
8-bit unsigned ROM	22	5.5	41.3 MHz
16-bit unsigned ROM	84	21	25.5 MHz
8-bit unsigned RAM	39	9.75	39.9 MHz
16-bit unsigned RAM	117	29.3	25 MHz

Table 3.3: Custom Multiplier Chip Performance

Part #	**Precision**	**Mult. Speed**
Logic Devices LMU08/LMU8U	8x8-16 bit signed/unsigned	28.6 MHz
Logic Devices LMU18	16x16-32 bit signed/unsigned	28.6 MHz
Harris HMU16/HMU17	16x16-32 bit signed/unsigned	28.6 MHz
Cypress CY7516/517	16x16-32 bit signed/unsigned	26.3 MHz
GEC Plessey PDSP16116/A	16-64 bit signed/unsigned complex	20 MHz

reference is best provided by full-custom solutions to multiplication (ie: multiplication ASICs). Table 3.3 lists several fixed-point parallel multiplication chips available from various manufacturers along with their performance. Only parallel custom multipliers are available for comparison as no known manufacturer produces custom multipliers that employ bit-serial multiplication.

As can be seen from Tables 3.1 and 3.3 most FPGA-based parallel multipliers obtain approximately $\frac{1}{4}$ to $\frac{1}{3}$ of the performance of the custom multipliers for the 8-bit versions while the 16-bit multipliers obtain only about $\frac{1}{10}$ the performance. The only FPGA-based multipliers that come close to matching the custom multiplier performance are the Xilinx-based constant multipliers based on the distributed-arithmetic approach. The non-pipelined versions of these multipliers achieve 75% and 33% of the performance of the custom multipliers for the 8- and 16-bit versions respectively. The pipelined versions exceed the performance of the custom multiplier for the 8-bit multipliers and nearly equal the performance for the 16-bit multipliers. The pipelined multipliers, however, have increased latency due to the pipelining registers. Thus, it is clear that for an FPGA-based system to obtain multiplication performance levels equal to that of ASIC systems using custom multipliers, extensive parallelism must generally be employed. Fortunately, many FPGA-based multipliers take only a small fraction of the device to implement and hence the use of several multipliers per device is practical.

Using custom multipliers with FPGAs

The results of Table 3.3 suggest the possibility of using custom multiplier chips in conjunction with an FPGA, with the FPGA providing the necessary control. This allows the use of multipliers designed in VLSI that are faster, smaller, and less expensive than equivalent implementations on FPGAs. Disadvantages of using external multipliers include the on-/off-chip time required for signals between the FPGA and the multiplier and the high I/O pin requirement when interfacing to a multiplication chip. For example, the 16-bit complex multiplier listed in Table 3.3 requires 128 pins just for data transfer. Some of the I/O constraints are eased with the 16-bit multipliers by multiplexing the inputs with the output data word but this requires extra control and adds latency to the multiplier.

3.2.3 FPGA performance comparisons

Comparing the multiplication performance of the various FPGAs to one another we can see that Xilinx FPGAs are able to provide the greatest multiplication performance due to their ability to implement the distributed-arithmetic approach to constant multiplication. The CLAy part is also able to implement distributed-arithmetic-based multipliers but with less efficiency. For other types of multipliers, however, the Altera, Xilinx, and CLAy parts perform nearly equivalently. One difference that is apparent by looking at Table 3.1 is that the bit-serial multipliers on the CLAy FPGA were about $\frac{1}{2}$ as fast but 3-8 times smaller than equivalent multipliers on the Altera and Xilinx parts. Thus, their area-time product for bit-serial multiplication is superior (ie: smaller) suggesting the use of large numbers of bit-serial multipliers on CLAy parts to achieve high multiplication rates.

The overall performance of a particular FPGA in a DSP algorithm cannot, however, be determined solely by its multiplication performance. Each of the three FPGAs discussed has its own particular weaknesses and strengths such as partial reconfigurability for the CLAy FPGA. Additional factors such as the ability to implement data buses and state machines affect the suitability of a particular FPGA to DSP algorithm implementation. In addition, the results listed in Table 3.1 must be adjusted for differences in process technology among the various FPGAs for a complete comparison. Each particular DSP algorithm also has a separate set of requirements that it places on the FPGA. Hence, the final analysis as to which FPGA is the best will thus be determined by the needs of the particular application.

3.3 Conclusions

The results presented in this chapter have shown that FPGAs are capable of providing multiplication performance that ranges from $\frac{1}{10}$ to equal the performance possible using custom multiplier chips. The ability of the FPGA to approach the performance level of ASIC-based DSP systems while exceeding that of DSP processor-based systems can thus not be determined solely on the multiplication results. Additional factors such as the degrees of parallelism that can be achieved and specific FPGA optimizations will play a large role in determining overall performance. The next chapter presents three DSP applications that will illustrate the ability of FPGA-based reconfigurable hardware systems to achieve performance superior to DSP processor-based systems while approaching the performance of an ASIC-based system. It also illustrates an attempt to combine external custom multiplier chips along with FPGAs to improve multiplication performance.

CHAPTER 4

EXAMPLE APPLICATIONS

In this chapter three distinct digital signal processing applications using FPGAs will be presented. The performance of these systems will be discussed and compared to equivalent or similar implementations using DSP processors and ASICs. The three applications to be discussed are a 20-tap FIR filter, a 64-point, radix-4 FFT, and a 3x3/5x5 image convolution filter.

4.1 FPGA-based FIR filter

A popular digital signal processing algorithm that is required in most signal processing systems and that is often used as a benchmark for rating the performance of signal processing systems is the finite impulse response (FIR) filter. The equation for the computation of an N-tap FIR filter is:

$$y[n] = \sum_{k=0}^{N-1} x[n-k]w[n]$$

This is simply the convolution of the input sequence (x[n]) with the filter coefficients/taps (w[n]). A signal flow graph for the computation of an N-tap FIR filter appears in Figure 4.1. As can be seen in the flow graph the computation of an FIR filter requires a series of multiplications which are summed together to form the final output. The computation can be performed sequentially using a multiply-accumulate loop, in full parallel using N multipliers and N-1 adders, or as a combination of sequential-parallel computation.

Several example FIR filters have been designed and implemented on the three FPGAs discussed in this thesis. The highest performing example was a fully parallel 10-tap FIR filter implemented using the Xilinx FPGAs with 8 bits of precision for both the data words and the coefficients. It uses non-pipelined constant multipliers for the coefficient multiplications since the coefficients for a non-adaptive FIR filter remain fixed in time. The filter was implemented using a single Xilinx 4010 FPGA by employing the distributed arithmetic approach to

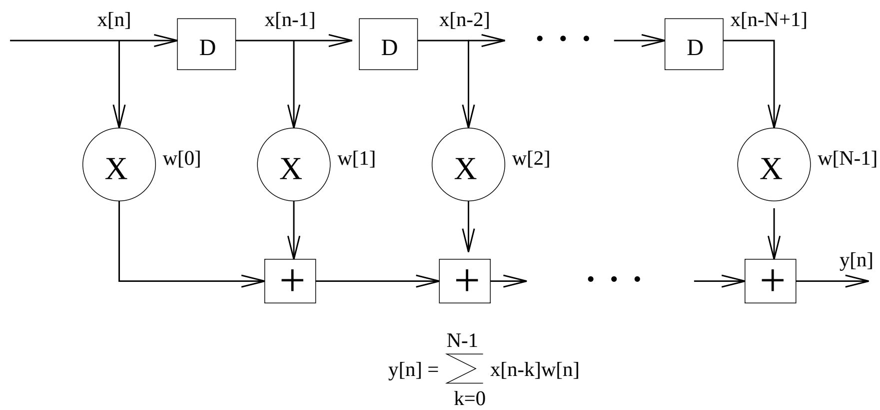


Figure 4.1: Signal flow graph for an N-tap FIR filter

constant multiplication (see Chapter 3). A Xilinx 4010 FPGA provides 400 configurable logic blocks (CLBs) of which 308 were used for an achieved 77% CLB utilization. This is close to the upper limit on utilization that is normally achieved due to the constraints imposed by the routing within the FPGA. The filter is fully pipelined (except within the multipliers themselves) and allows a new 8-bit data input each clock cycle. The filter is capable of a sustained operational rate of 26 MHz minimum. Multiple FPGAs can also be cascaded to achieve longer filters. Hence, to implement a 20-tap 8-bit FIR filter, two FPGAs are required. Using multiple FPGAs adds delay and an on-off chip time of 10 ns maximum is assumed which results in a lowering of the sustainable clock rate to 20.5 MHz. Figure 4.2 shows a block diagram for the architecture of a single 10-tap chip. Figure 4.3 shows the implementation of the constant multipliers on the FPGA and Figure 4.4 shows the schematic for the 10-tap filter. Finally, Figure 4.5 shows the placed and routed 10-tap filter on the Xilinx 4010 FPGA.

A 16-bit version of the filter requires constant multipliers that are four times as large as the 8-bit multipliers (see constant multiplier specs in Chapter 3) and 32-bit adders which are double the size of the 16-bit adders. This results in only being able to place four 16-bit filter taps per Xilinx 4010 FPGA. The operational speed is also reduced due to the size of the multiplications and additions to a sustainable 10 MHz including the 10 ns on-off chip delay time. Figure 4.6 shows the schematic for the 16-bit constant multiplier used to build these 16-bit FIR

Table 4.1: 20-Tap FIR Filter Performance

System	Precision	# of Chips	Computation Time	Data rate
DSP Processor Implementation				
TI TMS320C5X	16-bit	1	1.0 μs	.7389 MHz
Altera 81188 FPGA				
Altera 81188 U-Bit-Serial	8-bit	1	.190 μs	5.25 MHz
Altera 81188 U-Bit-Serial	16-bit	2	.477 μs	2.1 MHz
Altera 81188 S-Bit-Serial	8-bit	3	.227 μs	4.4 MHz
Altera 81188 S-Bit-Serial	16-bit	5	.51 μs	1.96 MHz
Altera 81188 Parallel	8-bit	5	.156 μs	6.4 MHz
Altera 81188 Parallel	16-bit	14	.304 μs	3.28 MHz
Altera 81188 Fast Parallel	8-bit	1	1.76 μs	.567 MHz
FPGA and Custom Multiplier Chips				
LD LMU08	8-bit	2	.9 μs	1.11 MHz
LD LMU18	16-bit	2	.9 μs	1.11 MHz
Xilinx 4010 FPGA				
Xilinx 4010 Fast Parallel	16-bit	2	4.8 μs	.208 MHz
Xilinx 4010 Constant ROM	8-bit	2	.049 μs	20.5 MHz
Xilinx 4010 Constant ROM	16-bit	5	.1 μs	10.0 MHz
CLAy31 FPGA				
CLAy31 S-Bit-Serial	8-bit	1	.421 μs	2.4 MHz
CLAy31 S-Bit-Serial	16-bit	1	.84 μs	1.2 MHz
ASIC Implementations				
LD LF43881	8-bit	3	.033 μs	30 MHz
PDSP16256/A	16-bit	2	.08 μs	12.5 MHz

filters on the Xilinx FPGAs.

To understand the performance of the 8- and 16-bit Xilinx-based filters relative to other implementations, comparisons have been made between the performance of a 20-tap version of these filters and the performance of 20-tap FIR filters implemented using other FPGAs, a DSP processor, and ASICs. Table 4.1 lists these performance comparisons.

4.1.1 Description of table entries

The table entry labeled *TMS320C5X* refers to the popular 16-bit fixed-point C5X DSP processors manufactured by Texas Instruments [19].The benchmark listed is for a C5X with a 35 ns cycle time and a 57 MHz external clock rate. The data throughput rate is less than the inverse of the computation time (1.0 μs) due to the overhead of executing instructions to set up the filter operation and

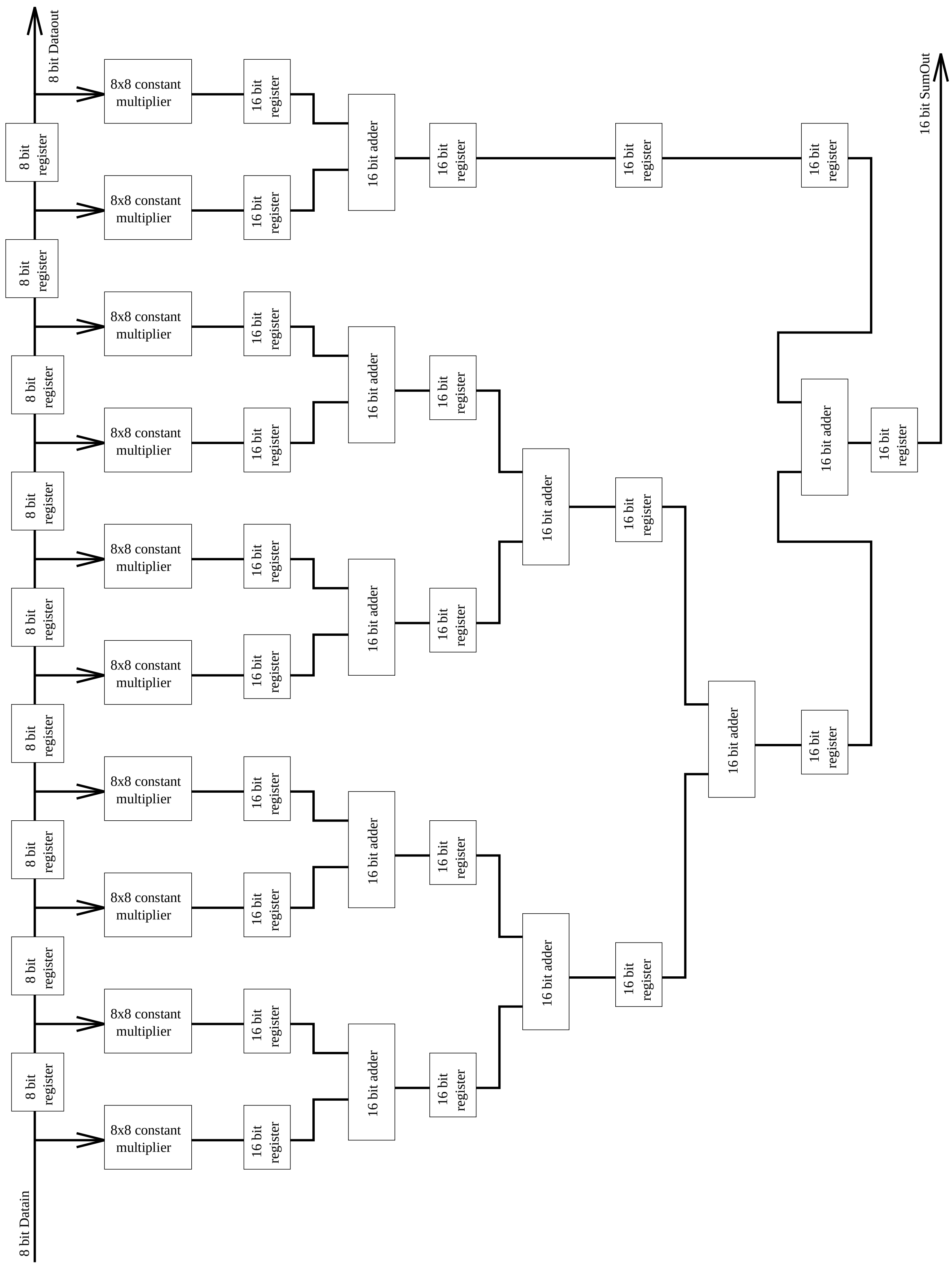


Figure 4.2: Block diagram for a 10-tap FIR filter chip

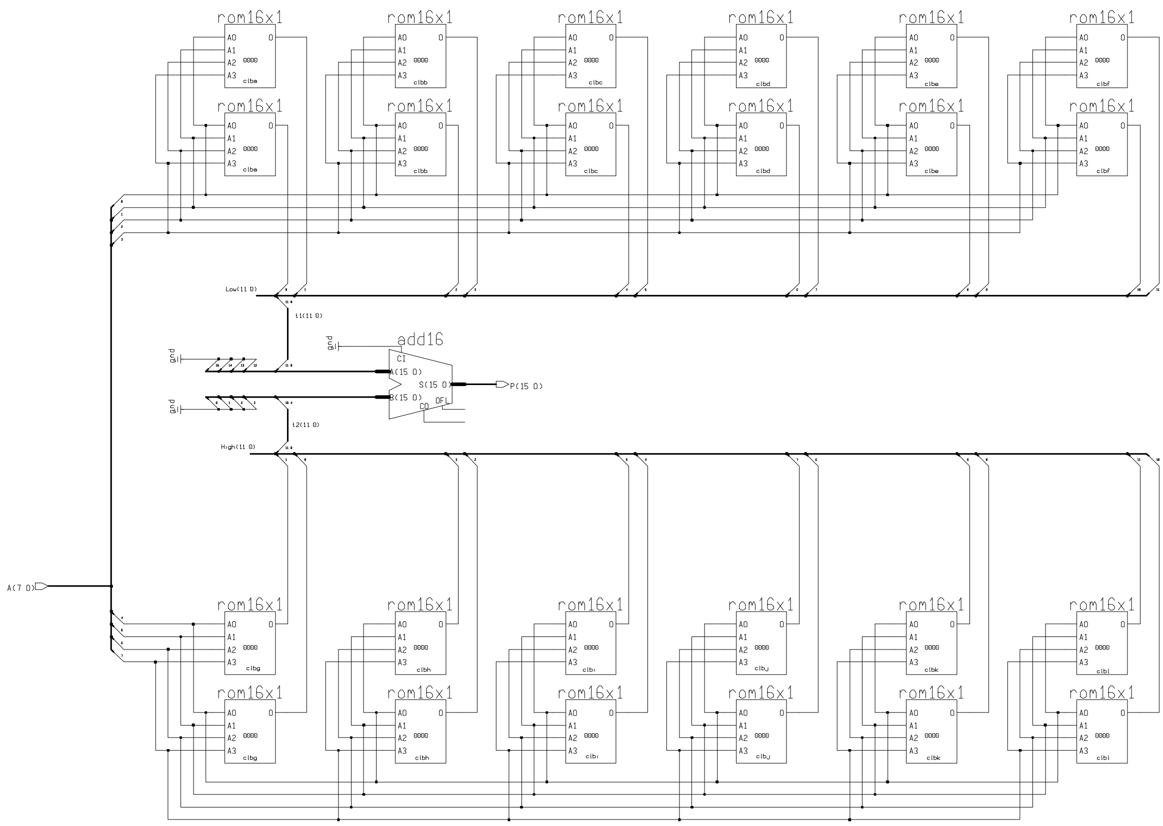


Figure 4.3: 8-bit constant multiplier used in the 10-tap FIR filter

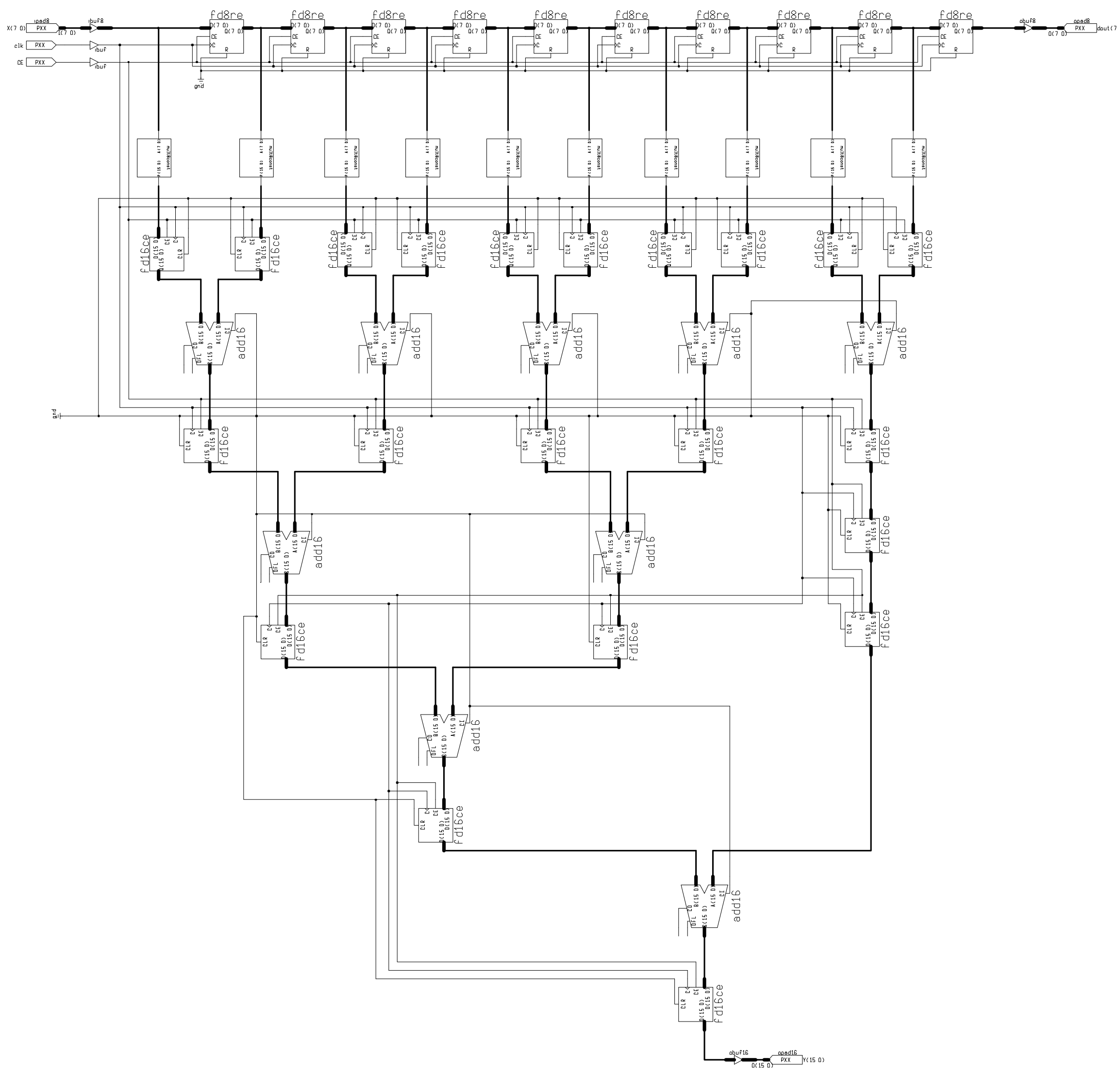
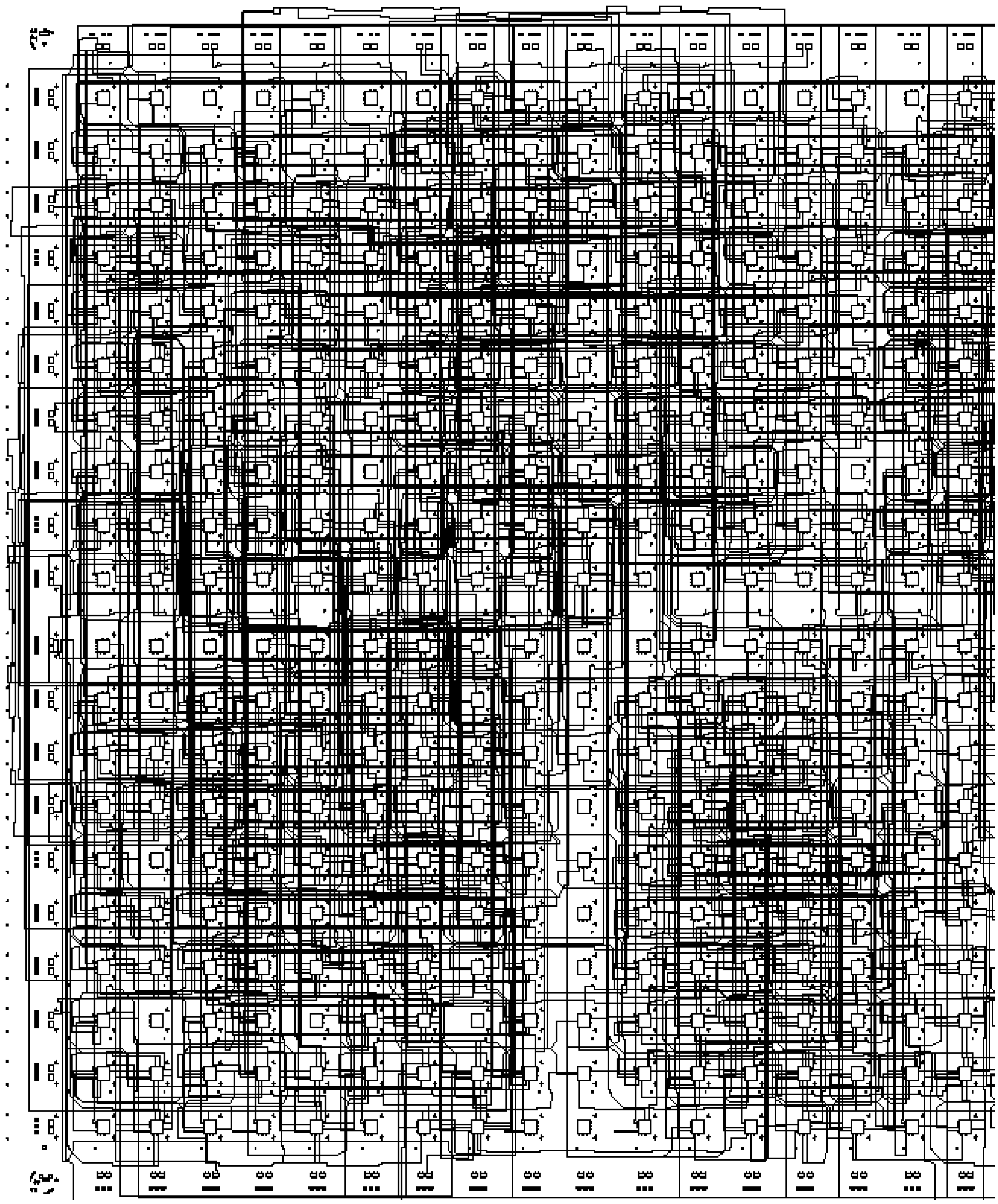


Figure 4.4: 10-tap 8-bit FIR filter schematic



Print Display: filter8x8.lca (4010PC84-4), xact 5.1.0, Fri May 5 13:06:56 1995

Figure 4.5: 10-tap 8-bit FIR filter on Xilinx 4010 FPGA

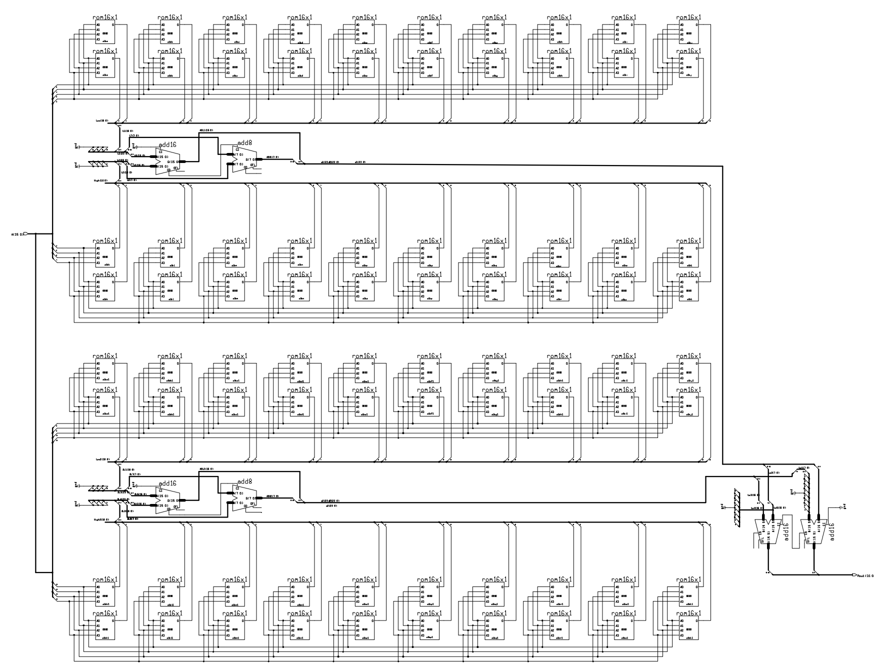


Figure 4.6: 16-bit Xilinx-based constant multiplier

moving data on and off chip.

The entries labeled *Altera U-Bit-Serial* refer to the use of unsigned bit-serial multipliers to build the 20-tap filters while those labeled *Altera S-Bit-Serial* refer to the use of signed bit-serial multipliers. The Altera 81188 FPGA, which has 1008 logic elements (LEs), was used for these implementations. Both signed and unsigned versions of the bit-serial implementations were included to illustrate the possible inefficiencies in the mapping of signed bit-serial multipliers to most FPGA architectures as compared to the mapping of unsigned bit-serial multipliers. In this case the difference in the mapping efficiencies results in an increase in system chip count by factors of 3 and 2.5 respectively for the 8- and 16-bit, 20-tap FIR filters.

The entries labeled *Altera 81188 Parallel* refer to the use of signed multipliers synthesized from VHDL. These were chosen to implement the filter rather than the fast-adder versions (see Chapter 3) since the fast-adder versions create routing difficulties when it is attempted to place multiple multipliers on a single chip due to their extensive use of the device carry logic.

The *LD LMU08* and *LD LMU18* entries refer to the use of 8- and 16-bit custom multiplier chips respectively from Logic Devices in conjunction with an FPGA to implement the filter. The FPGA is used to implement the necessary data delays, data path, multiplier chip control, and the product accumulation required for the multiply-accumulate loop of the FIR filter. The filter is calculated sequentially in time rather than in parallel through the use of a multiply-accumulate loop. Again, a 10 ns on-off chip delay time was assumed between the multiplier chip and the FPGA. For comparison to equivalent implementations using only up to two FPGAs with one FPGA being possibly dedicated to implementing the multiplier (16-bit version only) the entries labeled *Altera Fast Parallel* and *Xilinx Parallel* were included. The 8-bit FPGA-based filter was implemented using an Altera part with a fast-adder parallel multiplier based on the extensive use of carry logic within the FPGA. The 16-bit FPGA filter was implemented on a Xilinx part using a hand-routed parallel array multiplier.

The next entries in the table labeled *Xilinx Constant ROM* present the results for the FIR filters implemented using the Xilinx constant-coefficient multipliers as discussed above. The *CLAy31* entries refer to results extrapolated from a signed bit-serial FIR filter design on the CLAy architecture proposed by design engineer Raymond Andraka [3]. The final entries list results for two custom FIR filter ASICs, the Logic Devices **LF43881** 8x8 bit Digital Filter and the Gec Plessey **PDSP16256/A** Programmable FIR filter.

4.1.2 Comparisons and conclusions

Comparing all of the listed filter implementations it can be seen that the ASIC-based filters have the highest performance. Their performance, however, is nearly matched by the Xilinx-based implementations. This clearly indicates the advantage of the use of the distributed arithmetic approach to multiplication. Using this approach the 8-bit and 16-bit versions of the filter obtain speedup factors of 28 and 13 respectively over the DSP processor. The disadvantage of this approach is the need to implement all of the multiplications in parallel since each multiplier is a constant multiplier and is hence dedicated to a particular filter tap. This results in a larger chip count for the 16-bit filter (5 compared to 2 for the ASIC). The larger chip count for the 16-bit filter may be insignificant however since the ASIC-based filter chip counts listed do not include the inevitable support chips that must be included to set up and control the ASICs.

The only systems that performed worse than the DSP processor were those using only a single FPGA-based multiplier to perform the entire filter loop (entries labeled *fast parallel*). In these systems a single multiplier is used to compute the entire 20-iteration multiply-accumulate loop of the filter. This method most closely approximates the method used in the DSP processor for performing the filter but results in inferior performance due to the difference in speed of the FPGA-based multipliers and the VLSI-based multiplier on the DSP processor. Hence, when a custom VLSI multiplier chip is used in conjunction with an FPGA (table entries labeled *LD LMU08* and *LD LMU18*), this architecture again surpasses the performance of the DSP processor.

The results of this section thus clearly indicate the ability of FPGA-based filters to exceed the performance possible using a DSP processor. They exceed DSP processor performance by using hardwired control and increased parallelism by computing several multiplications at once, rather than one at a time as with the DSP processor. It is possible to increase the performance of a DSP processor-based system by placing several processors in parallel to work on the same problem. However, the speedup obtained using this technique is much less than the degree of parallelism introduced due to the need for complex interprocessor communication and synchronization. Hence, FPGA-based filtering is more efficient and is capable of achieving a performance level that approaches that of ASICs and far exceeds that of the DSP processor.

4.2 FPGA-based FFT processor

The fast Fourier transform is another popular digital signal processing algorithm. It is used in computing the discrete Fourier transform (DFT) of a digital input sequence. It is commonly used as a quick method of computing the DFT due to its inherent reduction of the number of multiplications required to compute the DFT. The equation for the DFT of an N-sample input sequence is:

$$X(k) = \sum_{n=0}^{N-1} x[n]e^{-jk(2\pi/N)n}$$

The straightforward computation of the DFT will require N^2 complex multiplications while a radix-4 FFT algorithm takes only $Nlog_4(N)$ complex multiplications. For a 1024-point FFT this implies a reduction factor of over 200 in the number of multiplications required which results in a considerable time savings.

A block diagram showing the flow graph for the computation of a 64-point, radix-4 decimation in frequency FFT [21] appears in Figure 4.7. The butterfly element used in this flow graph is shown in Figure 4.8. As can be seen by looking at the flow graph for the butterfly element, the radix-4 butterfly only requires additions and subtractions and no multiplications. The only multiplications needed to compute the radix-4 FFT are those for the twiddle-factor multiplications between the columns of the FFT as shown in Figure 4.7. This has the advantage of making the butterfly uniform, allowing the same butterfly structure to be used throughout the entire FFT. To compute the 64-point, radix-4 FFT a total of 192 complex multiplications and 3 stages are required (the three columns of the FFT shown in Figure 4.7)

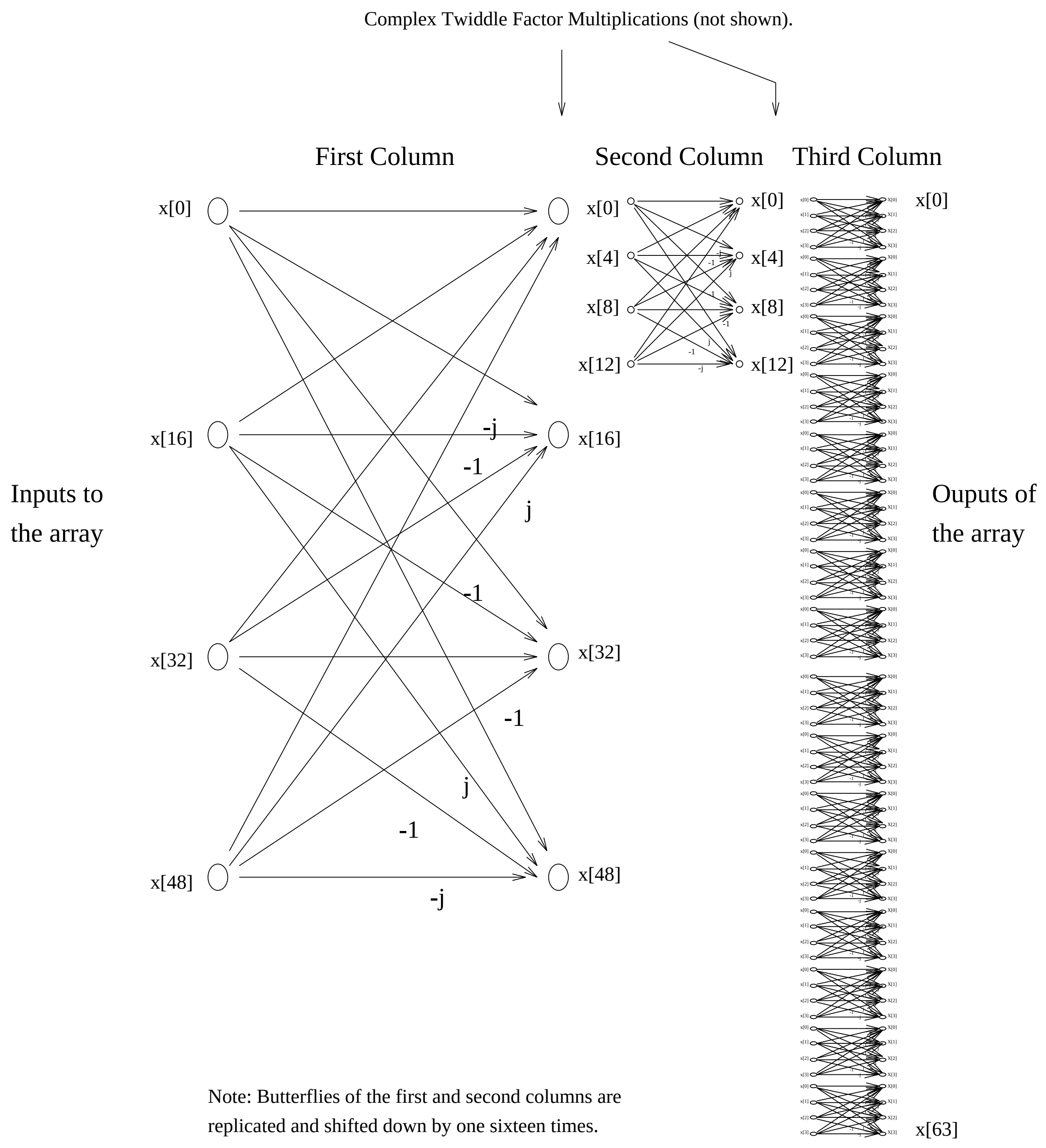


Figure 4.7: Flow graph of radix-4 64-point decimation-in-frequency FFT.

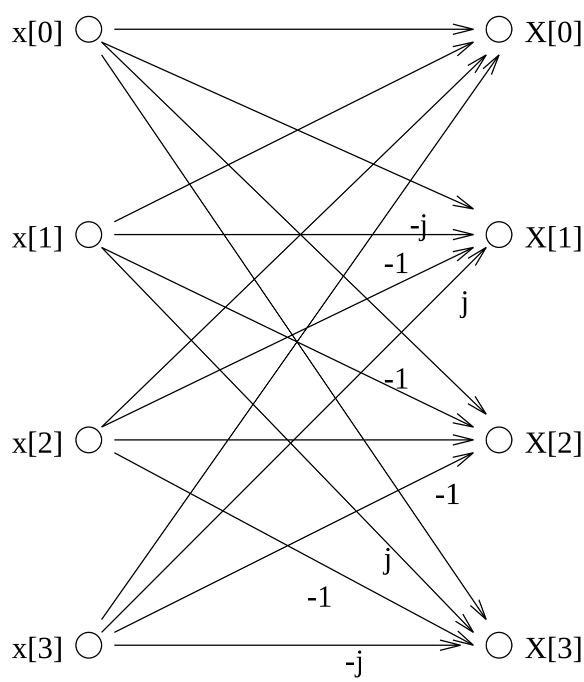


Figure 4.8: Radix 4 butterfly element

4.2.1 FPGA FFT hardware implementation

The radix-4 FFT FPGA hardware was built to compute a 64-point FFT but it is easily modified to handle an FFT of any length. The radix-4 FFT algorithm was chosen since its butterfly element requires no multiplications and is hence easily reused and simple to compute. 8 bits of precision for both the real and imaginary portions of the complex data were used for a total of 16 bits per data word. The FFT was implemented in reconfigurable hardware using two Altera 81188 FPGAs. The first FPGA was used to implement the complex multiplier required for the twiddle-factor multiplications while the second FPGA was used to implement the complex butterfly element, the control logic, the datapath, and the addressing logic. A single 8-bit SRAM was also required to store the FFT data and the complex coefficients used in the twiddle-factor multiplications. A block diagram of the hardware elements appears in Figure 4.9.

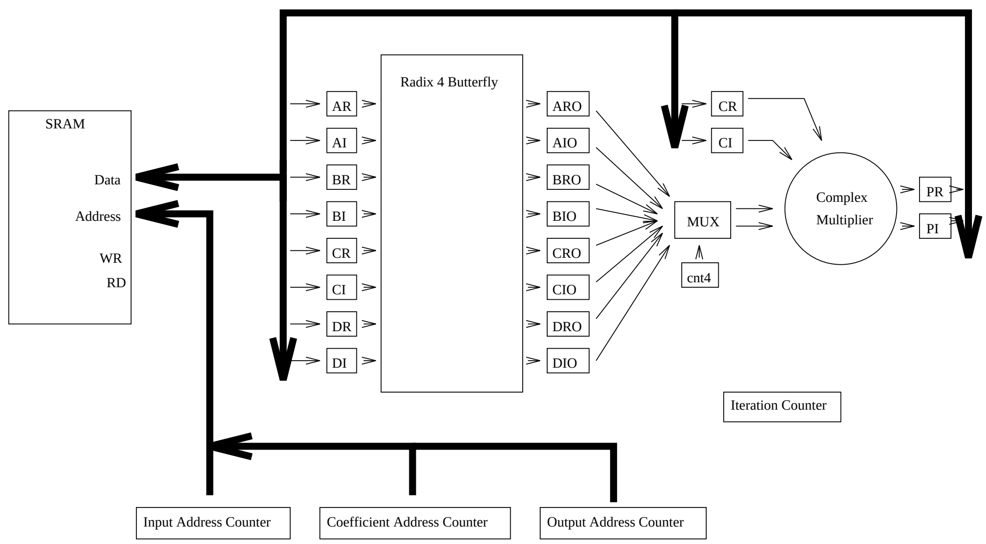


Figure 4.9: Block diagram of the FFT hardware elements

Computational hardware

The main computational components used in the FFT are the complex multiplier, the radix-4 butterfly element, and the address generation units. The complex multiplier was implemented using behavioral VHDL mixed with a small amount of structural code to handle the overflow case. Eight bits of precision were chosen to implement the multiplier and hence the FFT due to the size of the RAMS on the Altera 81188-based RIPP10 FPGA board and the physical size of a synthesized complex multiplier. The complex multiplier consists of four fully parallel array multipliers and two addition units and performs the function of computing the complex twiddle-factor multiplications that are needed between the columns of the FFT (see Figure 4.10).

The butterfly element was also implemented using mostly behavioral VHDL. The butterfly element takes four complex inputs (8 real data values) and produces four complex outputs in parallel using word parallel arithmetic. Hardware to latch the data values for the four complex inputs from a single eight bit input bus is also included in the butterfly element as well as latches for the output values (see Figure 4.11).

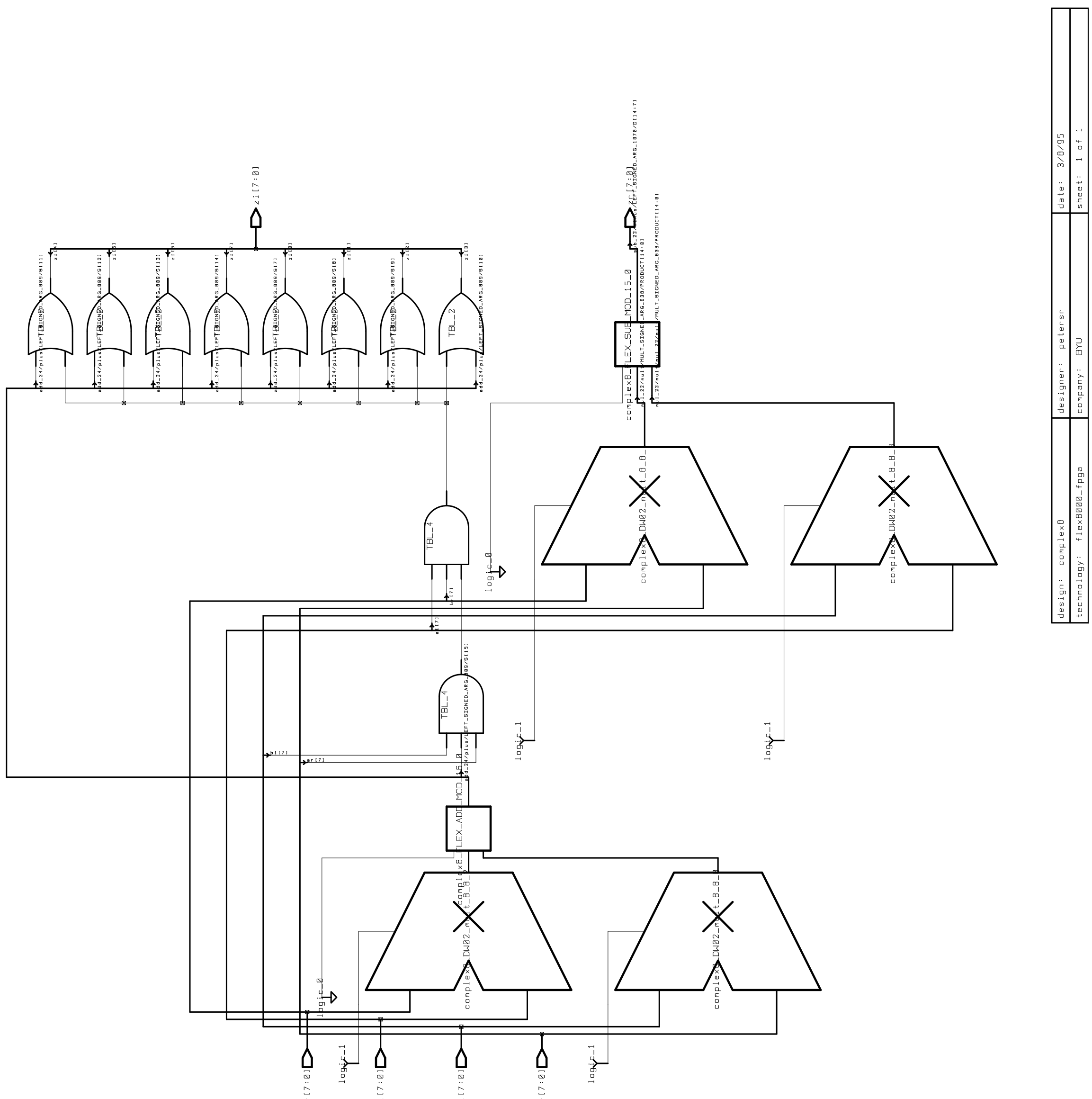


Figure 4.10: 8-bit complex multiplier

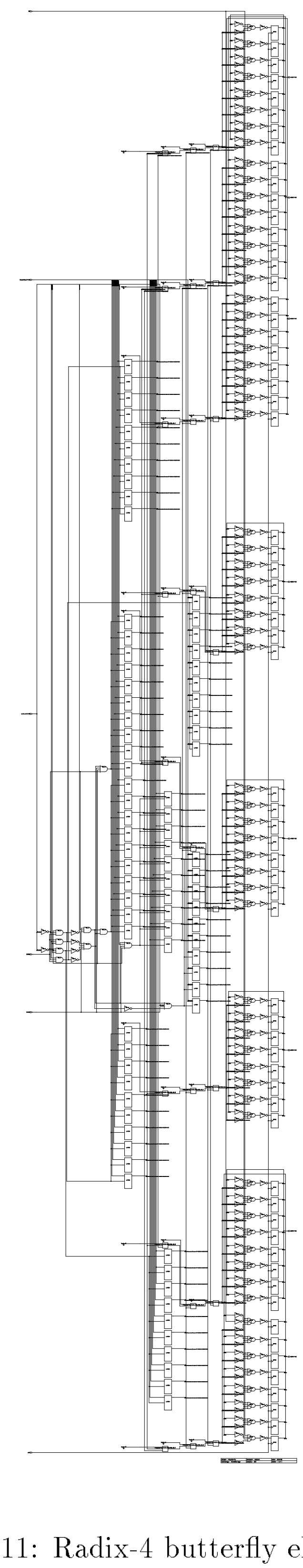


Figure 4.11: Radix-4 butterfly element.

The address generation hardware consists primarily of counters to produce the addresses for: the input data values, the coefficients used in the twiddle factor multiplications, and the output data values. Each counter is used to effectively produce two addresses simultaneously (one for the real data value and one for the imaginary data value) by adding 64 or 128 to each address (see Figures 4.12, 4.13, 4.14).

Datapath hardware

The datapath hardware for the FFT consists of a multiplexor to control the routing of the butterfly element outputs to the complex multiplier and a multiplexor to route one of two possible complex coefficient values to the complex multiplier. The first multiplexor is needed since the butterfly element produces four complex output values simultaneously which must be routed successively to the multiplier. The second multiplexor is used to allow loading of a second complex coefficient during multiplier operation. One coefficient is routed to the multiplier while the second is being loaded. Both of these multiplexors were implemented using simple if-then statements and behavioral VHDL. (see Figures 4.15, 4.16).

Control hardware

To implement the control necessary for the FFT hardware a fully synchronous, one-hot state machine design was chosen. A one-hot design method was chosen since one-hot methods result in minimization of the number of logic levels needed to form the next state and outputs of the system [24]. This is important to FPGA implementation since it is the number of levels of logic that must be evaluated that determines the overall speed of operation for the state machine. Fully structural VHDL was used to code the state machine since first attempts using behavioral VHDL resulted in the synthesis of large, low-performance designs.

The state diagram for the state machine appears in Figure 4.17. As shown the fully synchronous controller requires a total of nine states and hence nine flip-flops. The function of the state machine is to sequence the proper loading and storing of the data values for the complex multiplier and the butterfly element. The first five states of the controller serve to load the data necessary for the first

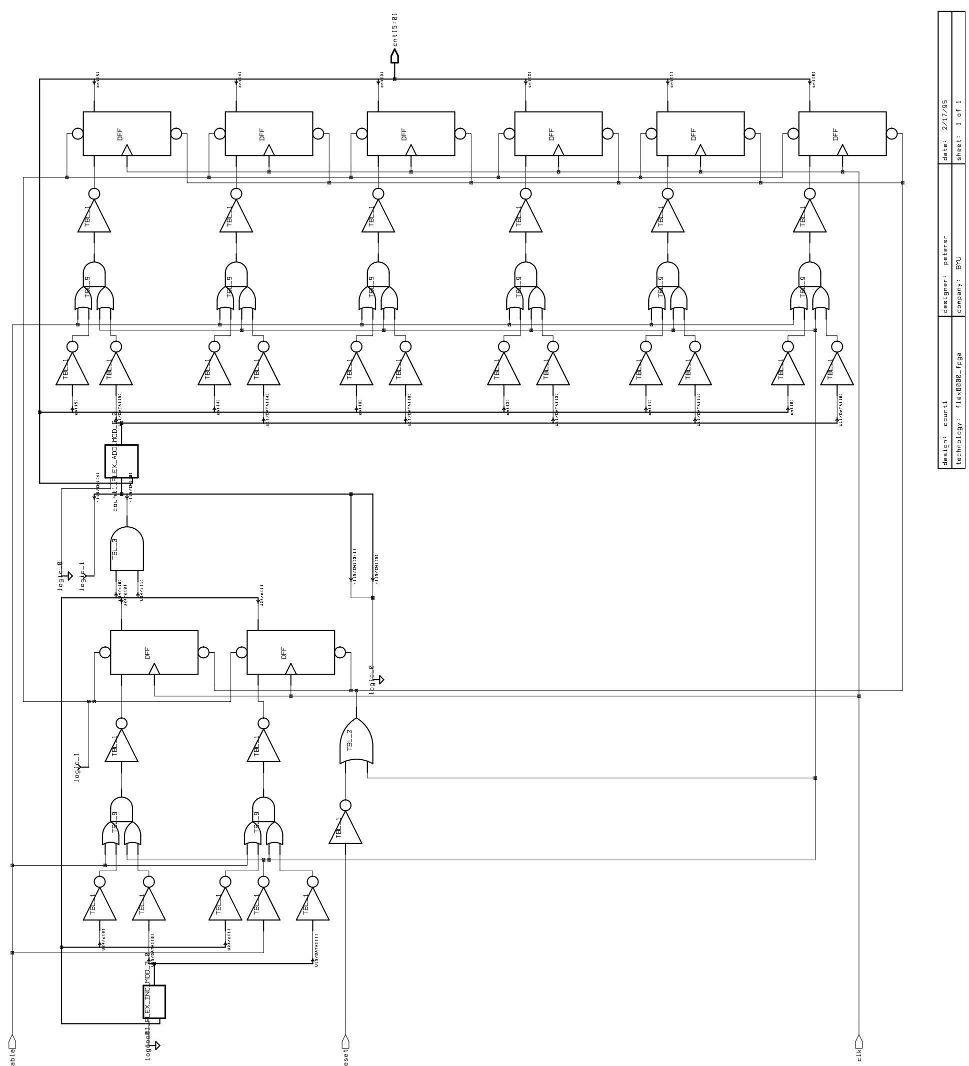


Figure 4.12: Address generator for the first column of the array

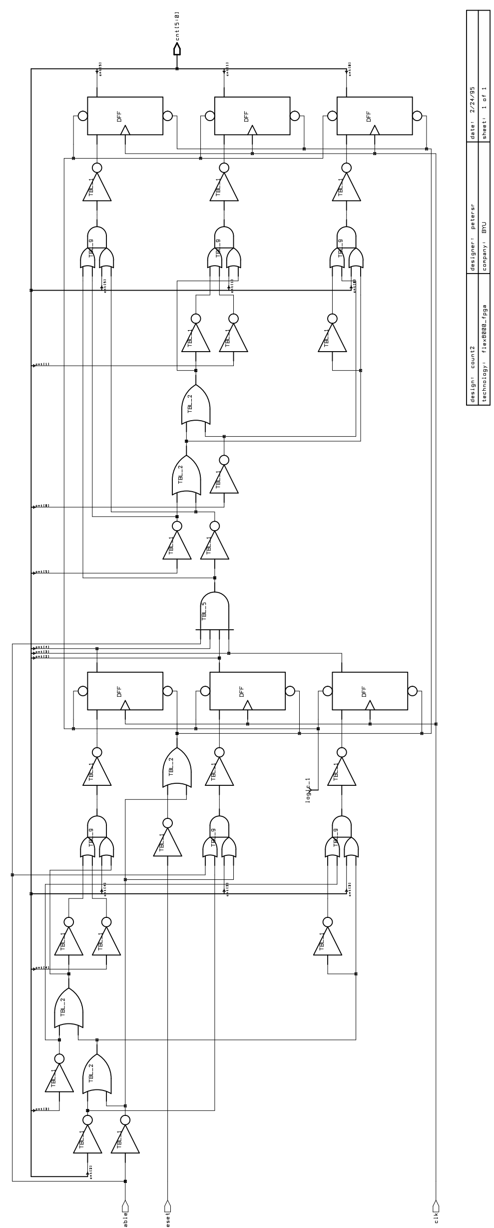


Figure 4.13: Address generator for the second column of the array

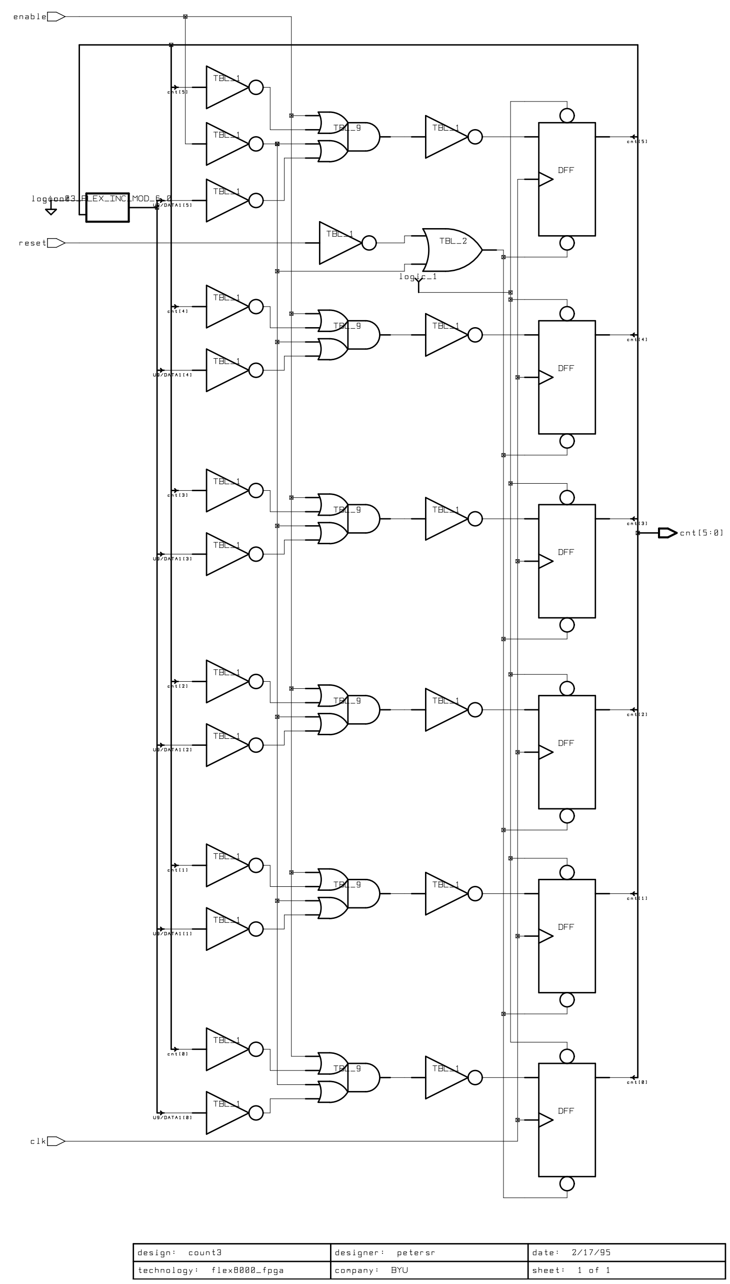


Figure 4.14: Address generator for the last column of the array

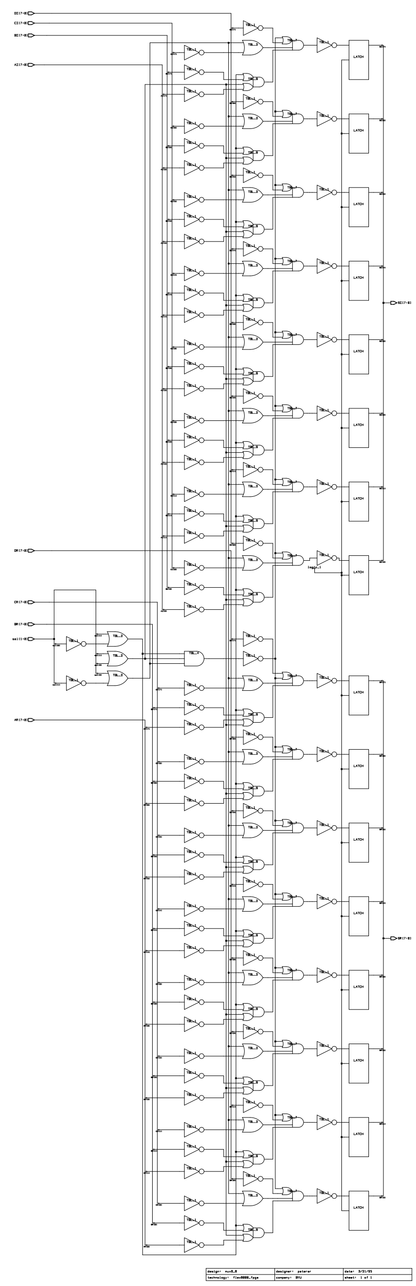


Figure 4.15: Multiplexor to select outputs of the butterfly element

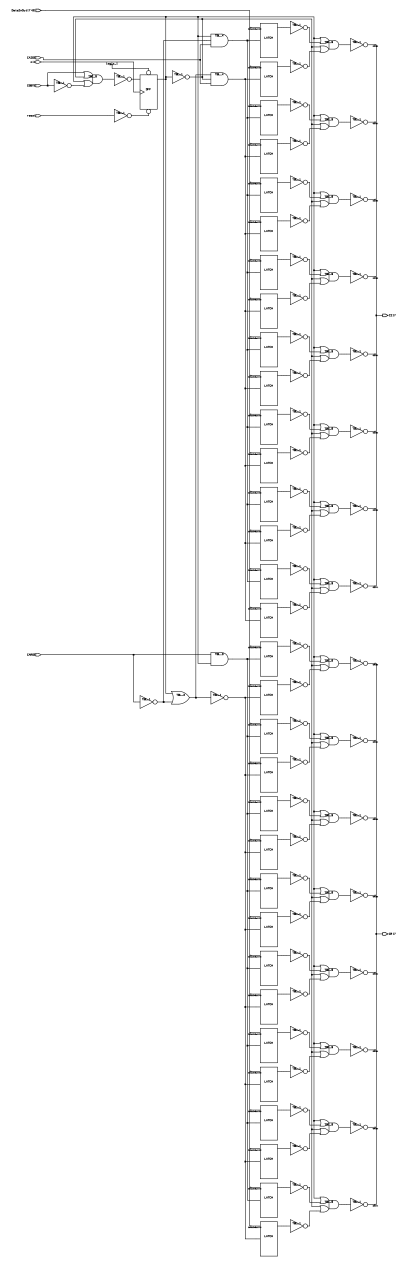


Figure 4.16: Multiplexor to control routing of coefficient values to the butterfly element

Table 4.2: Complex Radix-4 FFT Performance

System	Precision	# of Chips	Computation Time		
			64pt	**256pt**	**1024pt**
TI TMS320C5x	16-bit	1	108μs	617μs	3.84ms
Altera 81188 Parallel	8-bit	2	46μs	245μs	1.03ms
PDSP16116/A	16-bit	2	11.52μs	61.4μs	.307ms
PDSP16510A	16-bit	1	-	-	98μs
Sharp LH9124	24-bit	2	-	-	80.7μs

butterfly computation while states F,G, and H load the data for the next multiplication and butterfly calculation during the multiplier operation. The results of the previous multiplication are also written to memory during the successive multiplication. Thus, the function of the system is to keep the multiplier busy 100% of the time since the multiplier is the limiting path of the system. Outputs of the state machine are also used to control the multiplexors on the output of the butterfly element and the input of the complex multiplier as well as produce the proper read and write signals for the data ram.

4.2.2 FFT hardware performance

The system as built requires two Altera 81188 FPGAs and can operate at a system clock rate of 25 MHz. This allows the computation of a 64-point FFT in 46 μs. Small changes can be made to the addressing circuitry to allow the same system to be used to compute 256- and 1024-point FFTs with computation times of 245 μs and 1.03 ms respectively. Table 4.2 compares these times to the computation times of 64-, 256-, and 1024-point FFTs using ASICs and DSP processors.

As can be seen from the table the FPGA implementation achieves speedup factors of 2.3, 2.5, and 3.7 over the TMS320C5x DSP processor for FFTs of length 64, 256, and 1024 points respectively. The numbers listed for the DSP processor also represent a best case situation in that they are calculated assuming that the data is already in memory on the DSP chip and therefore neglects the overhead time imposed by the need to move data on and off chip. The DSP processor does however utilize greater precision in its calculations and hence the

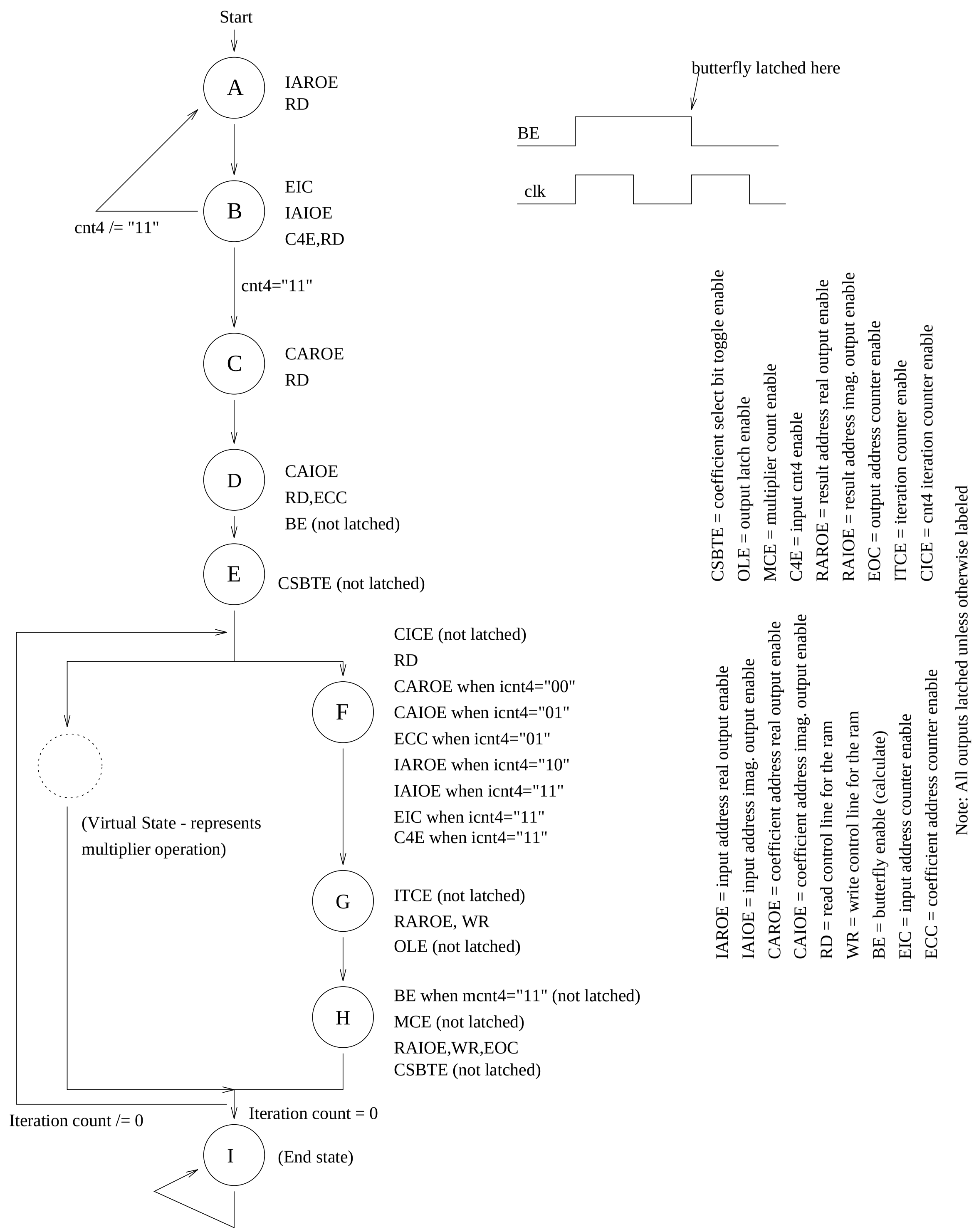


Figure 4.17: State diagram for the state machine controller

comparison is not completely equal. In fact, the FPGA-based FFT processor as implemented is impractical due to the lack of adequate precision in internal computations. Computation of the multiplications, additions, and butterflies (see Figure 4.7) in each stage of the FFT should utilize greater word length than 8 bits. The data word size, however, is adequate for many applications involving sensor data which is often of a low precision (8 bits or less). Figure 4.18 shows the output of the FPGA FFT processor with a white noise sequence as input plotted against the ideal output. The error in the output can be easily seen. Also, Figure 4.19 shows plots of the ideal and actual outputs of the FPGA-based FFT processor with a sine wave input. The bottom plot in the figure is the output of the FFT processor and shows the spectral artifacts that result from the lack of precision in intermediate computations.

A better comparison can be performed by replacing the FPGA used to implement the complex multiplier with a custom complex multiplier chip. This is labeled as *PDSP16116/A* in the table and utilizes the PDSP16116/A complex multiplier chip from GEC Plessey Semiconductors. This chip is capable of performing a 16-bit complex multiplication in 50 ns. The use of the custom multiplier chip greatly enhances the speed of the computation since the algorithm as layed out on the FPGA is primarily compute bound rather than I/O bound. The limiting computational element is thus the complex multiplier. With a 10 ns penalty for inter-chip delays added to the 50 ns multiplication rate speedups over the DSP processor of 9.4, 10, and 12.5 are obtained for the three FFT lengths. This is a more accurate comparison since the use of the external multiplier chip allows 16 bits of precision to be used for the real and imaginary portions of the complex data.

For comparison with the performance of ASIC-based implementations the table entries labeled *PDSP16510A* and *Sharp LH9124* are included. These are custom FFT processor ASICs with clock rates of 40 MHz each. The PDSP16510A and LH9124 obtain performance speedup factors of 10.5 and 12.8 respectively over the FPGA FFT processor discussed above. Speedup factors over the FPGA/custom multiplier solution are 3.1 and 3.8.

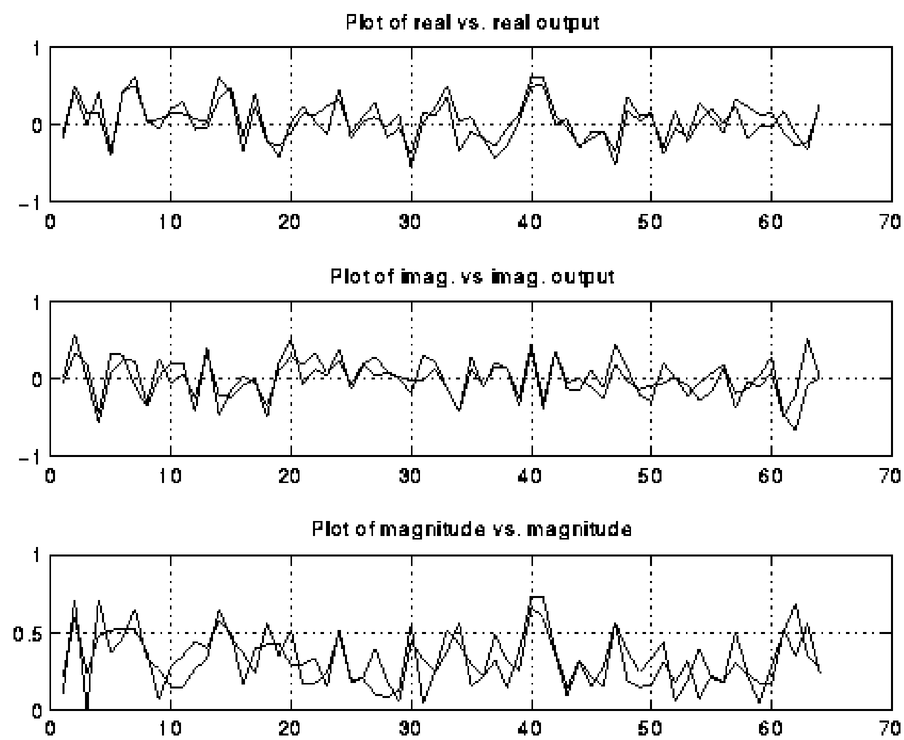


Figure 4.18: FFT of white noise sequence - Output of FPGA vs. ideal output

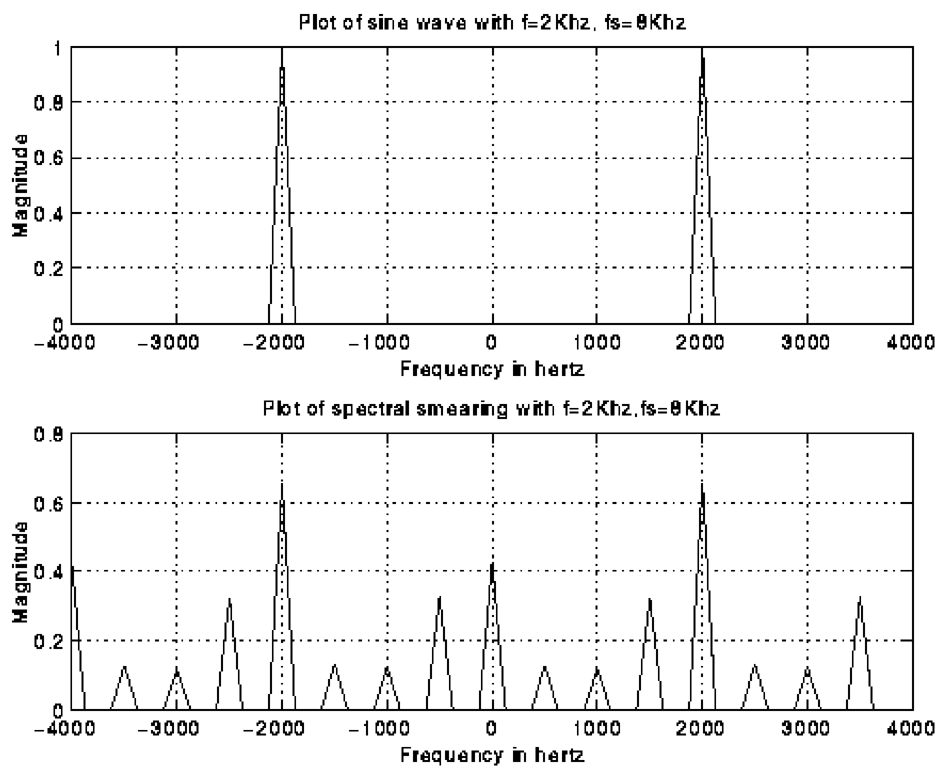


Figure 4.19: FFT of sine wave input - Output of FPGA vs. ideal output

4.2.3 Conclusions

These results place the performance of the FPGA-based FFT implementation at around halfway between that of the DSP processor and a full custom ASIC solution. Increased speedups can be achieved with the FPGA approach by using several FPGA-based FFT processors in parallel with each processor being used to compute one column of the FFT. For example, using one PDSP16116/A chip and FPGA per FFT column a 1024-point, radix-4 FFT could be performed in $307/5 = 61.4\mu$s. This approach, however, can also be applied to the ASIC-based approach. Hence, these results indicate that performance midway between that of the DSP processor and the ASIC is a reasonable expectation of an FPGA-based FFT processor.

The results for the FPGA-only versus the FPGA/custom multiplier solution also illustrate the difficulty FPGAs have implementing large, fast non-constant multipliers. In cases such as this it is better to use custom built VLSI multipliers rather than trying to implement the multiplication on the FPGA. One possible optimization that thus presents itself is the concept of adding fixed VLSI multipliers to an FPGA die. This possibility will be discussed in Chapter 5.

4.3 FPGA-based image convolution filter

The third application implemented in reconfigurable hardware was a two-dimensional (2-D) image convolution engine handling convolution kernel sizes of 3x3 or 5x5. The general equation for a two-dimensional convolution of this type is:

$$y[m,n] = \sum_{k=-M}^{M} \sum_{l=-N}^{N} x[m-k, n-l] w[m,n]$$

where N=M=1 for a 3x3 convolution kernel and N=M=2 for a 5x5 kernel. The input data x[m,n] in this case is a two-dimensional image that is convolved with the convolution kernel coefficients w[m,n]. Figure 4.20 shows the calculation of three successive outputs of a two-dimensional convolution using a 3x3 kernel mask size.

As can be seen in Figure 4.20 the image convolution operation makes repeated use of the input image pixels in calculating sucessive outputs while also requiring for each output 9 multiplications and 8 additions for a 3x3 kernel and 25

Coefficient Mask — Rows of pixels

C00 C01 C02 p00 p01 p02 p03 p04 . . . .

C10 C11 C12 $\longrightarrow$ p10 p11 p12 p13 p14 . . . .

C20 C21 C22 p20 p21 p22 p23 p24 . . . .

OUT0 = C02*p02 + C12*p12 + C22*p22
+ C01*p01 + C11*p11 + C21*p21‘
+ C00*p00 + C10*p10 + C20*p20

OUT1 = C02*p03 + C12*p13 + C22*p23
+ C01*p02 + C11*p12 + C21*p22
+ C00*p01 + C10*p11 + C20*p21

OUT2 = C02*p04 + C12*p14 + C22*p24
+ C01*p03 + C11*p13 + C21*p23
+ C00*p02 + C10*p12 + C20*p22

Figure 4.20: Convolution using a 3x3 convolution kernel

multiplications and 24 additions for a 5x5 kernel. A typical data rate for digital video is on the order of a *10 MHz pixel rate*. This rate would, for example, support images of 512 by 512 pixels at 30 frames per second. Supporting this data rate requires multiplication rates of 90 and 250 MHz for the 3x3 and 5x5 kernel sizes, respectively. Given these excessive multiplication rates in addition to the need for extensive data manipulation and product accumulation (the additions) it is easy to understand the need for special purpose hardware.

4.3.1 Hardware implementation

To meet the above performance requirements in reconfigurable hardware a total of three Altera FPGAs were used for the 3x3 kernel size while five were used for the 5x5 kernel. Each FPGA contained a nearly identical processor, each of which is capable of computing a single output. The processors were all designed entirely using VHDL to limit the development time required. Figure 4.21 shows a block diagram of the processor used for a 3x3 kernel size. Each processor in the 3x3 kernel simply multiplies and accumulates three successive columns of three input pixels, producing an output after the third mutiply-accumulate cycle. The outputs of each of the three processors are then time-multiplexed so that one output is produced after each column of pixel input. This process can be envisioned as sliding the coefficient mask along a set of three rows of pixels, producing an output at each step as shown in Figure 4.20. The coefficients of the kernel mask are stored internally to the processors and are multiplexed at the appropriate times to the internal processor multipliers to produce the proper products. Each of the three processors is thus capable of computing an entire 3x3 convolution with each processor using its own timing. Thus, while processor 1 is completing a convolution, processor 2 is in the middle stage and processor 3 is just beginning the convolution. Their outputs are then time-multiplexed to achieve one output per column of pixel input. For example, in Figure 4.20 OUT0,OUT1, and OUT2 would be successively calculated by the three separate processors. OUT3 would then be calculated by the first processor, OUT4 by the second, and OUT5 by the third. This continues with each processor calculating the successive outputs of the convolution. Figure 4.22 illustrates this time-multiplexing of the outputs for a 3x3 kernel size. The same method is used for a 5x5 kernel size but with 5 processors and 5 multipliers per processor.

To meet the 10 MHz pixel rate each processor must be capable of producing an output every $N*100$ ns for an NxN convolution. An output is required per processor only every $N*100$ ns rather than every 100 ns since there are N processors rather than one being used to compute the outputs. Each stage of the convolution, however, must be computed in 100 ns and requires N multiplications and $N-1$ additions. To meet this computation rate multipliers with speeds on the order of 12 MHz are required.

FPGA implementation detail

To implement the convolution processors with FPGAs the Altera PC-based RIPP10 board containing 8 Altera 81188 FPGAs and 4 128k 8-bit SRAMs was used. 8 bits of precision were used for the pixel inputs with 9 bits of coefficient precision. Input images were 256 level gray-scale images with 8-bit 256 level gray-scale images as output. 20 bits of internal accumulation precision were employed

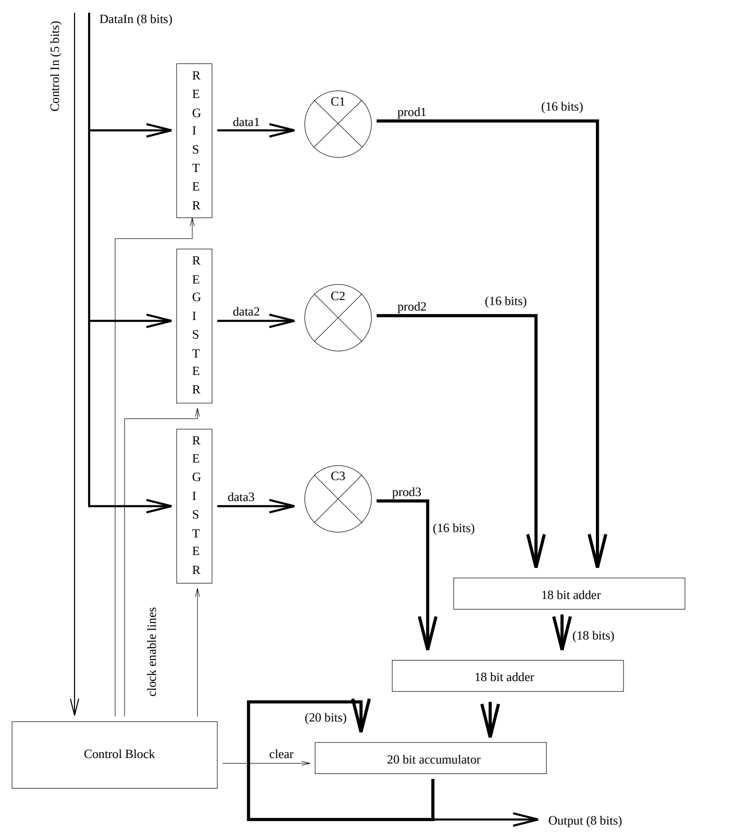


Figure 4.21: Convolution processor for 3x3 kernel size

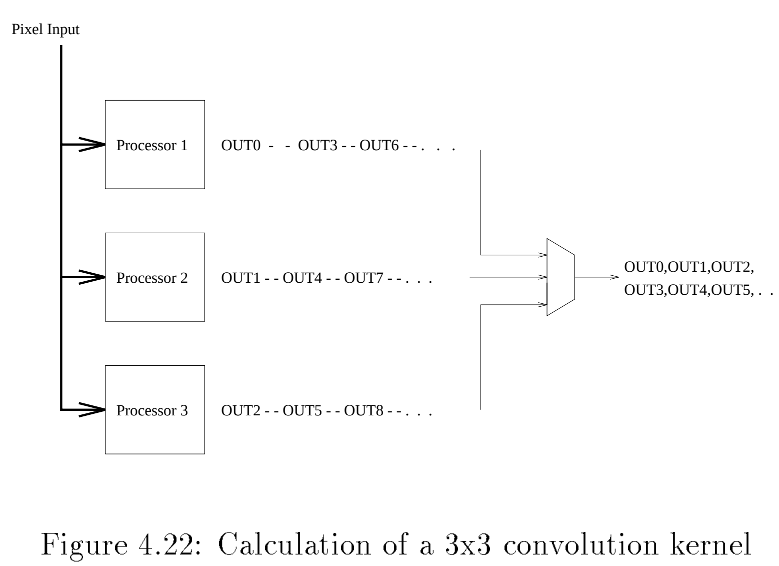


Figure 4.22: Calculation of a 3x3 convolution kernel

Table 4.3: Convolution Processor FPGA Resource Utilization

Type of Filter	% of 81188 Used
3x3 Mean	22
3x3 Laplacian Edge Detector	42
5x5 Mean	43

with the underflow and overflow cases being handled by setting the output pixel to the lowest or highest gray-scale values respectively (saturation arithmetic).

Because it was desired that the processor function as a coprocessing unit to a PC host, the data and control were fed directly from the host over the PC-ISA bus. This limited the system performance to less than one frame per second due to the slowness of the ISA bus and the limitation of 8-bit wide data transfers. Use of a local bus such as the PCI local bus ($>$ 100 Megabytes per second capacity) would, however, allow full speed operation since the required I/O bandwidth is on the order of 40 Megabytes per second for a 10 MHz pixel rate.

Implementation strategy

Each processor as implemented was placed onto a single FPGA for convenience in design and implementation. However, FPGA resource utilizations for several of the processors were low enough that two or more processors should ideally fit on a single FPGA. Table 4.3 lists the 81188 FPGA resource utilizations for 3x3 mean and Laplacian edge detection filters and a 5x5 mean filter.

As can be seen in Table 4.3 the utilization percentages depend greatly on the type of filter being implemented. This is a direct result of the size of the multipliers needed in the filter. The size of the multipliers used is dependent upon the type of filter being implemented due to the nature of the coefficients. The multiplier size is dependent on the coefficients since coefficients with a larger number of 1's require more hardware to implement. When a large number of zeros are present in the set of coefficients that will be applied to a multiplier, large portions of the multiplication array can be optimized out due to the disabling effect of a constant zero. Thus, it is often desirable to choose a coefficient set that will allow the greatest multiplier optimization to occur. Since the coefficients are successively multiplexed to the multiplier, the coefficient sets should be chosen so that their union has the greatest number of zeros. That is, the multipliers are implemented by taking the binary union of the coefficients and using this union to determine what portions of the multiplication array must be kept. For example, the union of 110011010 and 000100000 is 11011010. The portions of the multiplier array corresponding to each of the ones in the union must thus be kept in order to implement the multiplication of the coefficient set 110011010 and 000100000. A good example of the size reduction that can result from the choice of coefficients is shown by the coefficients used in the 3x3 mean and edge detection filters. For the mean filter the 9-bit coefficients used were 1/8 (000100000) and zero. Thus, their union is 000100000. For the edge detection filter the coefficients used were -1 (111110000) and 8 (01000000) (radix point placed after fifth digit). Their union is 111110000. As can be seen the union of the edge detection filter coefficients requires many more ones than the union of the coefficients in the mean filter. This results in a multiplier size ratio of 3 to 1 which accounts for the 50% decrease in hardware usage for the 3x3 mean filter over the 3x3 edge detection filter. The actual multiplier construction was handled by the VHDL synthesizer. Once the coefficient sets were chosen the VHDL synthesizer was allowed to optimize the multiplication arrays for the particular sets of coefficients used.

The VHDL code and the schematics produced by the VHDL synthesizer for the convolution processors appear in the appendix.

4.3.2 Performance results

The hardware as implemented is capable of a sustained 10 MHz pixel rate operation given the proper I/O coupling. To change the kernel coefficients only the constants in the VHDL description of each of the processors for the filter need be changed. The processors must then be re-synthesized and placed and routed to the FPGAs. Once the filters have been created, the FPGAs can be reprogrammed with a new filter in only a few seconds.

Two 3x3 and one 5x5 example filters have been implemented to date. Figure 4.23 shows the coefficient masks implemented by each of the filters while Figures 4.27, 4.28,4.29, and 4.30 illustrate the results of applying each of the filters to an input gray-scale image. The masks utilize simple coefficients whenever possible to reduce the size of the hardware but the processors are able to implement any set of desired coefficients. The only difference is in the size of the resulting multiplication arrays as discussed above. Figures 4.24, 4.25, and 4.26 show schematics of the multipliers used for each of the example filters.

-1	-1	-1
-1	8	-1
-1	-1	-1

3x3 Laplacian Edge Detector

.125	.125	.125
.125	0	.125
.125	.125	.125

3x3 Mean Filter

.04	.04	.04	.04	.04
.04	.04	.04	.04	.04
.04	.04	.04	.04	.04
.04	.04	.04	.04	.04
.04	.04	.04	.04	.04

5x5 Mean Filter

Figure 4.23: Example 3x3,5x5 filters

4.3.3 Performance comparisons with alternative implementations

As stated above, the image convolution processors as designed are capable of a 10 MHz pixel rate. This in itself is a performance level adequate for many video resolutions and rates. However, to better understand this performance comparisons with ASIC-based systems have been made. The purpose of these comparisons is to obtain a better reference with which to mark the performance obtained by the FPGA-based processors and hence gage their success. In addition, the potential performance of an FPGA-based convolution system employing the distributed arithmetic approach to multiplication (see Chapter 3) will be presented to illustrate the potential speedup that can be obtained even over the current FPGA solution. Table 4.4 lists these comparisons.

Table contents

The first entry in the table is for the Altera-based convolution processors already discussed in this section. The entries labeled *PDSP16488* refers to a 2-D convolution custom ASIC made by GEC Plessey. It can match the performance of the implemented FPGA-based solution for both 3x3 and 5x5 kernel sizes (10

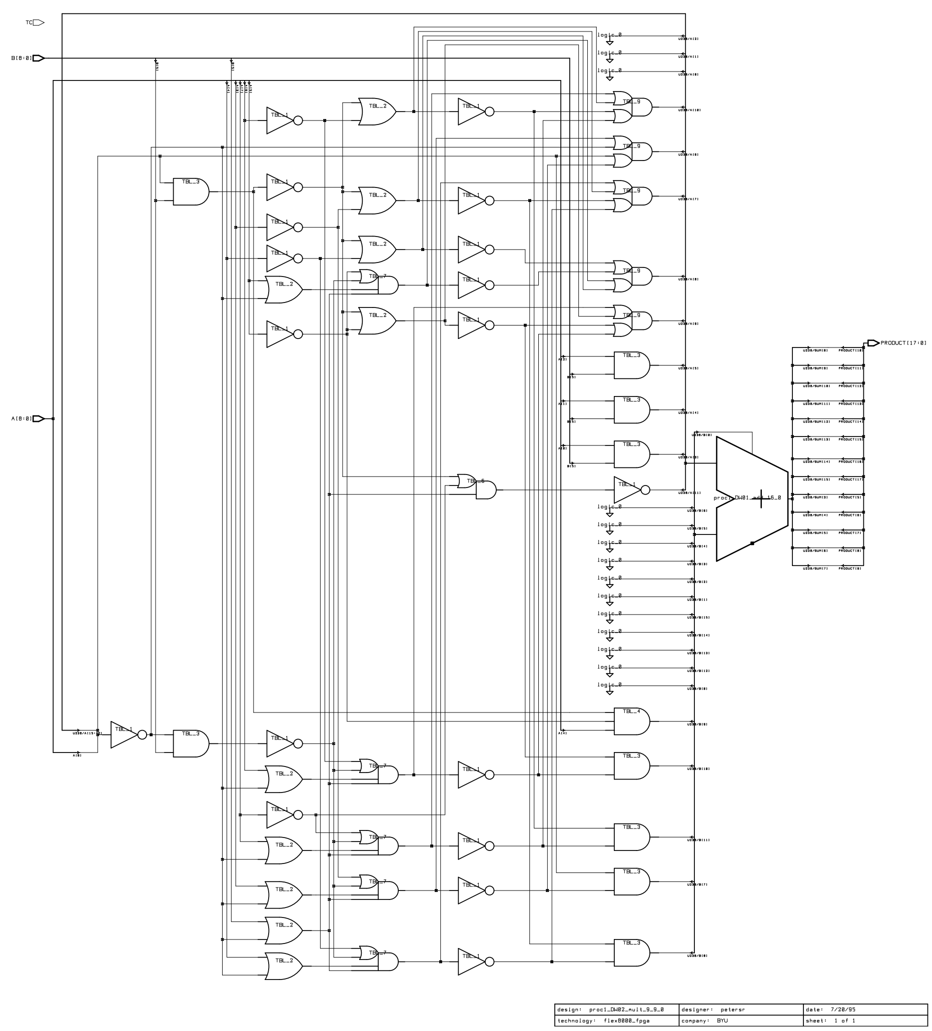


Figure 4.24: Multiplier for 3x3 mean filter

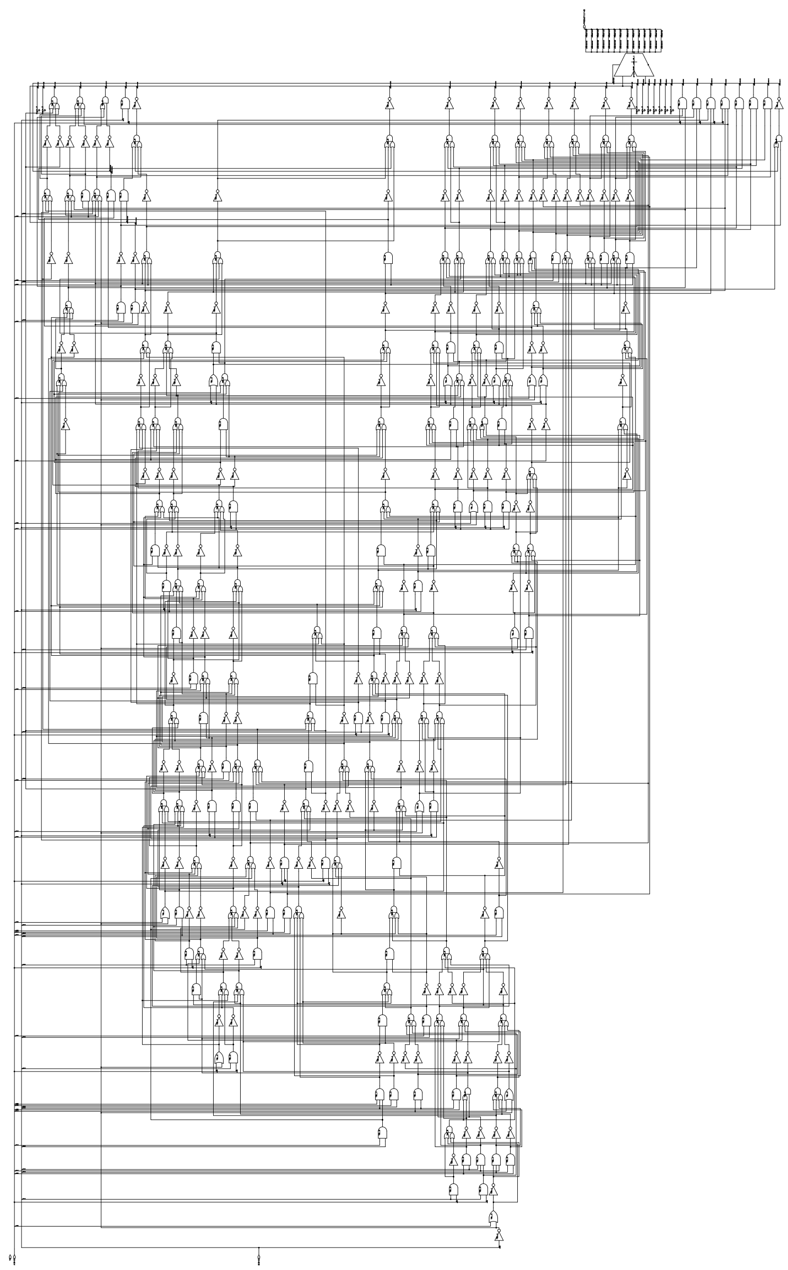


Figure 4.25: Multiplier for 3x3 edge detection filter

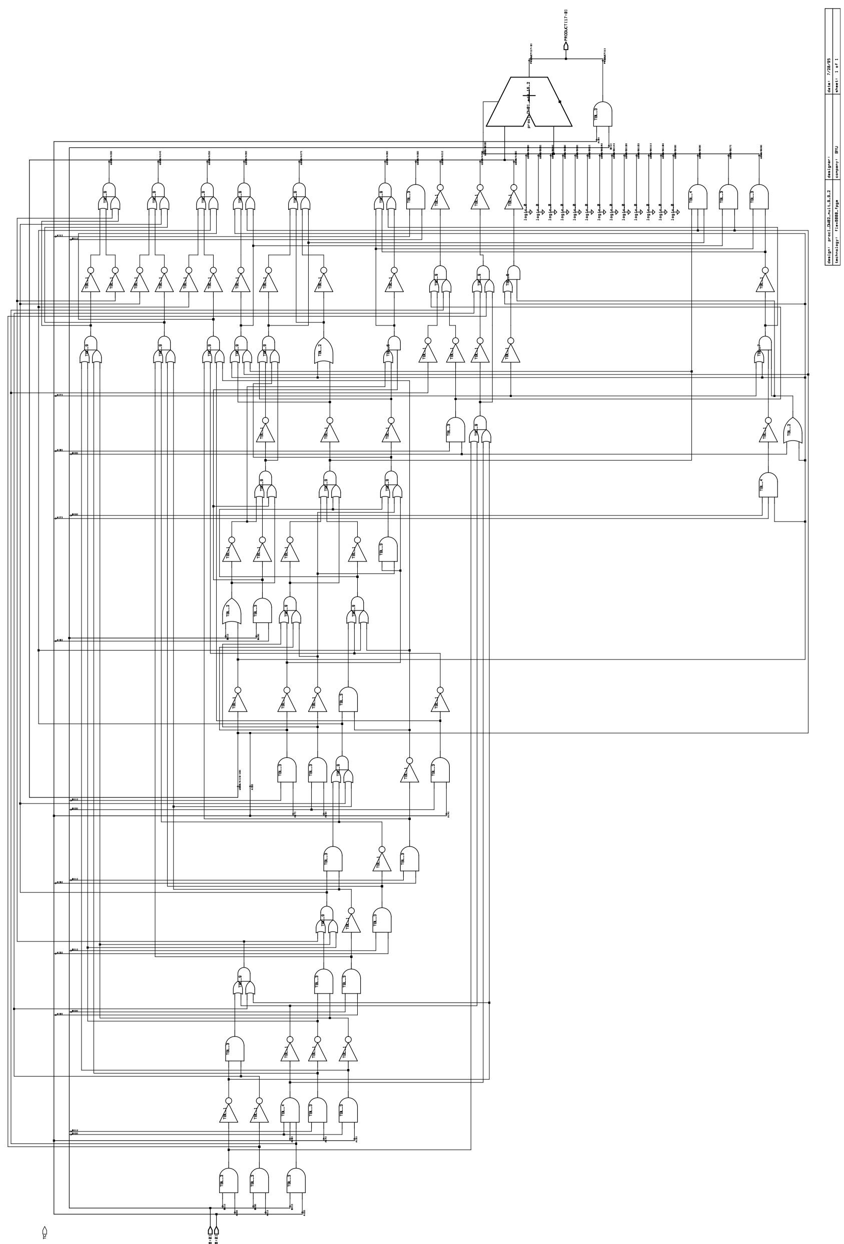


Figure 4.26: Multiplier for 5x5 mean filter

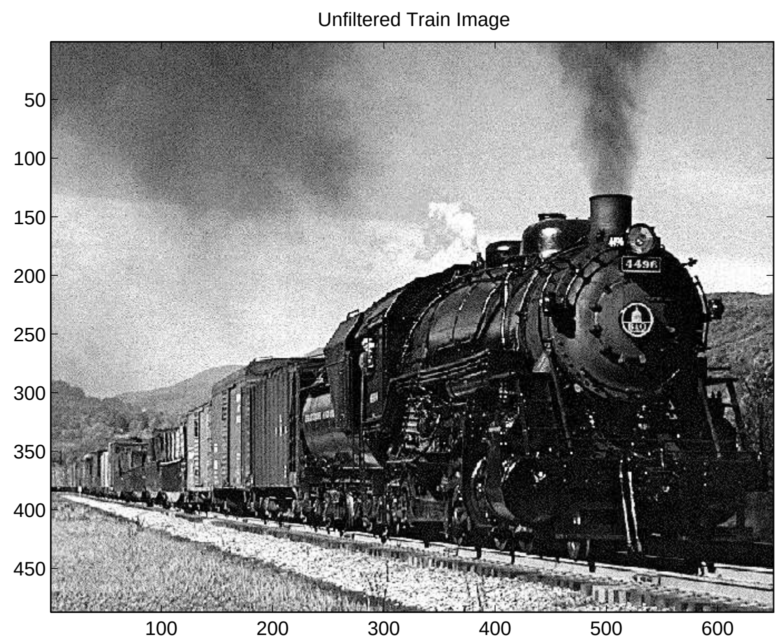


Figure 4.27: Unfiltered train image

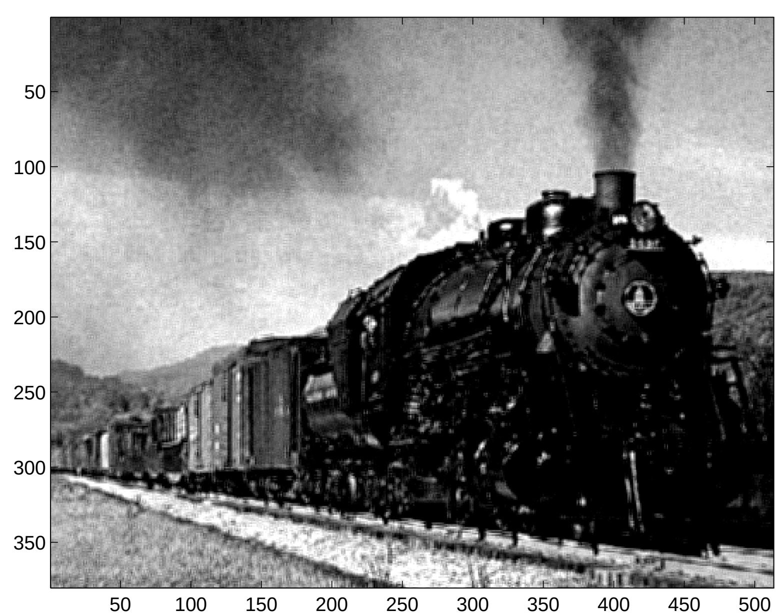


Figure 4.28: Train image filtered with 3x3 mean filter

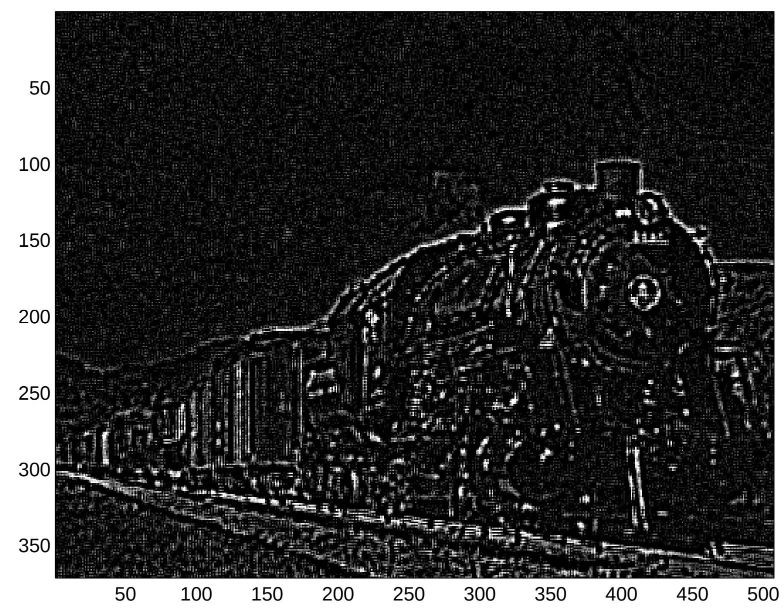


Figure 4.29: Train image filtered with 3x3 edge detection filter

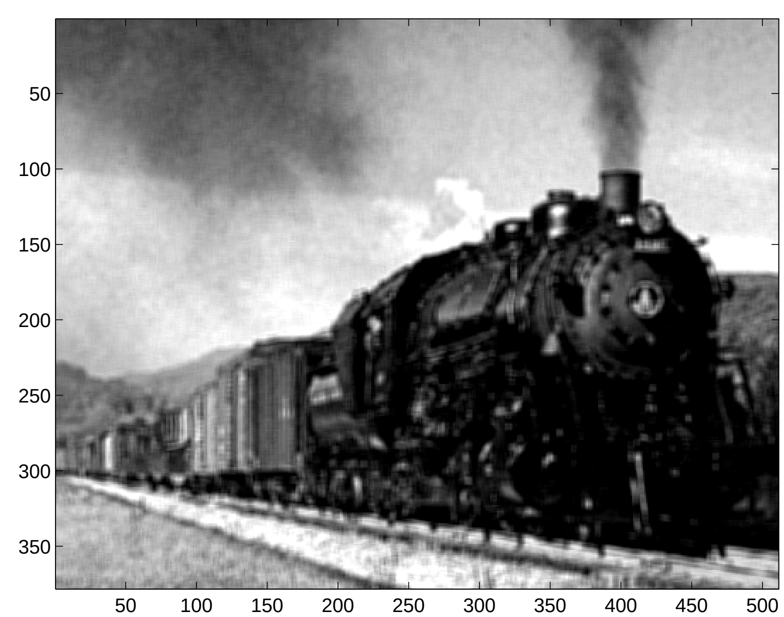


Figure 4.30: Train image filtered with 5x5 mean filter

Table 4.4: 2-D Video Convolution Processor Performance

Filter Implementation	**Pixel Rate**	**Kernel Size**	**Chips**
Altera 81188	10 MHz	3x3/5x5	3-5
PDSP16488	10 - 40 MHz	3x3	1 - 2
PDSP16488	10 - 40 MHz	5x5	1 - 4
LF48908	40 MHz	3x3	1
HSP48908	32 MHz	3x3	1
TMS320C80	13.8 MHz	3x3	?
TMS320C80	6.6 MHz	5x5	?
Xilinx 4010	35 MHz	3x3	1 - 2
Xilinx 4010	35 MHz	5x5	3 - 4

MHz pixel rate) using a single ASIC. In addition, it provides the delays needed to implement the pixel delay lines needed for 2-D convolution and can handle pixel sizes of 8 or 16 bits.

The entries labeled *LF48908* and *HSP48908* are essentially equivalent devices from Logic Devices and Harris Semiconductor respectively that are capable of also equaling and exceeding the performance of the Altera FPGA-based solution. Both of these chips assume 8-bit pixel and coefficient values.

The entries labeled *TMS320C80* refer to the use of a new 4 million transistor high-performance DSP processor by Texas Instruments that employs four fixed-point digital signal processors and a floating-point processor in addition to specialized data handling units on a single chip [20]. The individual processors on the C80 are organized in a multiple-instruction, multiple-data (MIMD) fashion and are hence programmed independently. The performance numbers listed in Table 4.4 are based on TMS320C80-based video system developed at the University of Washington [28] using 8-bit pixel data. The number of chips required for the supported pixel rates was listed as questionable since the video system utilizes many support chips used to interface the system with a PC. No other performance numbers were available for the TMS320C80 since it is a new product from Texas Instruments and has not been extensively used to date.

The final entries labeled *Xilinx 4010* refer to a possible solution (not actually implemented) using the distributed arithmetic approach to multiplication in the Xilinx parts as outlined in Chapter 3. The estimates were derived using the sizes of 8-bit pipelined multipliers and 16- to 20-bit adders on the Xilinx 4010 allowing 25% area overhead on the chip for routing. This solution nearly equals the performance of the ASIC-based solutions for 8-bit pixel inputs. In fact, it is able to provide a performance level that exceeds that of the HSP48908 in throughput although it requires a larger chip count.

4.3.4 Conclusions

As can be seen from Table 4.4, the implemented FPGA-based approach to 2-D convolution achieves a performance level greater than that of the new TMS320C80 DSP from Texas Instruments (5x5 kernel size only) while approaching or exceeding the performance of ASIC-based systems. Most important is that the FPGA-based approach is capable of supporting a sustained 10 MHz pixel rate which is considered a standard video bandwidth. The additional benefit that the FPGA-based approach provides over the ASIC-based systems is it ability to allow hardware reuse through reconfiguration. Thus, an FPGA-based system can be used not only to perform multiple 2-D convolutions but other video operations as well.

4.4 Conclusions on example application performance

In all three of the example applications studied in this chapter reconfigurable hardware (FPGAs) DSP systems were able to significantly outperform equivalent implementations using digital signal processors. The FPGA-based approaches were also able to approach or exceed the performance of the ASIC-based implementations. This clearly indicates the potential of FPGA-based signal processing systems to provide performance increases over DSP processors while maintaining flexibility. In addition, the flexibility provided by FPGA reprogramming allows unique optimizations not available to either ASICs or DSP processors. These potential optimizations along with possible FPGA architecture and FPGA-design tool improvements will be discussed in the next chapter.

CHAPTER 5

CONCLUSIONS AND FURTHER WORK

The goal of this thesis was to quantitatively determine if reconfigurable hardware using FPGAs was capable of providing a significant performance improvement over the DSP processor for digital signal processing applications. The performance of the three example applications implemented on FPGAs, employing results of the multiplier performance study, indicated that an order of magnitude performance increase over the DSP processor performance can generally be expected with FPGA-based reconfigurable systems. In addition, FPGA-based systems are able to often approach or exceed the performance of custom DSP hardware solutions based on ASICs. These results thus justify the application of FPGAs and reconfigurable systems to digital signal processing. From the results of the multiplier study and example application implementations, several conclusions can be formed regarding the application of reconfigurable hardware based on FPGAs to digital signal processing as well as suggestions for further work in this area. These conclusions and suggestions will be discussed next.

5.1 Thesis conclusions

5.1.1 FPGA multiplier performance

The results from the multiplier study performed in this thesis indicated that multiplier performance on FPGAs was highly dependent upon the specific multiplier architecture used. In specific, constant multipliers using distributed arithmetic were shown to provide the highest multiplication performance while signed bit-serial multipliers provided the lowest multiplication performance. Bit-serial multipliers, however, were also shown to be compact and are easily pipelined which can lead to an overall area-time performance metric that is less than for equivalent parallel multipliers. The performance of all of the multipliers compared to equivalent custom VLSI multipliers was also shown to range from as little as $\frac{1}{10}$th (for 16-bit multipliers) to equal to or greater than the performance of the custom multiplier. The conclusions can thus be formed that it is best to use constant multipliers when possible on the FPGA, especially those based on the distributed arithmetic approach, to employ bit-serial techniques when space is at a premium, and to use external custom multiplier chips when high-performance, large precision (8 bits or greater) non-constant multiplications must be performed.

5.1.2 Increased application specificity

An additional advantage of the use of reconfigurable hardware over the DSP processor and DSP ASICs that was mentioned briefly in Chapter 2 is the ability of reconfigurable hardware to employ increased application specificity through hardware reconfiguration. A reconfigurable hardware system can be made to be even more application specific than a VLSI ASIC since the specific requirements of the system can be fixed in hardware at the time the system is to be used. For example, ASICs, while designed to be applied to a specific application class, almost always allow some flexibility through the setting of switches or the programming of function registers. This allows the user of the ASIC to adjust certain basic operational parameters specific to his application of the ASIC. This is not necessary with reconfigurable hardware since the necessary parameters can be adjusted by changing the data stream used to program the hardware. This has the advantage of reducing the need for the extra circuitry needed to allow the parameter adjustments and may also lead to an increase in performance since the adjustments have been made to actual hardware rather than through the setting of "soft" values.

Although it was not presented in the FFT discussion this approach is amenable to the FFT implementation discussed in Chapter 4. In this application the addressing hardware of the FFT must be adjusted according to the length and column (stage) of the FFT to be calculated. In Figure 4.9, which shows the block diagram of the FFT hardware, the only portions of the hardware that must be changed to adjust for FFT length and column number are the counters. This is a simple change and allows the counters used to be smaller and hence faster since they are no longer required to generate all possible address sequences but only those for the specific length of FFT and the FFT column number being computed. This optimization of the address counters allows them to not be the performance bottleneck for the the algorithm as is otherwise the case with the FPGA-based implementation. The FIR filter application of Chapter 4 also benefits from increased application specificity in that it allows the use of constant multipliers employing the distributed arithmetic approach rather than general purpose multipliers which must be loaded with coefficient values at start-up time such as for an ASIC filter chip.

5.1.3 Device and design process enhancement

It is a personal opinion of the author that in order for reconfigurable hardware to become acceptable and widely used for digital signal processing the associated design-cycle time must be shortened. Currently the design of a DSP system using reconfigurable hardware can take as much as ten times as long as an equivalent design using DSP processors due to device usability problems, the lack of sufficient high-level design support, and difficulty with application debugging. Each of these problems will be discussed briefly.

Device usability

At current most FPGA compiler software has difficulty placing and routing a given circuit onto the FPGA when greater than about 70% of the FPGA resources are required. This is because if a greater percentage of the resources are utilized the design may fail to route due to the lack of sufficient routing resources within the FPGA. It is the author's opinion that a designer using reconfigurable hardware should not have to be burdened with caring about whether or not a given design will route to a particular device. The designer should only be concerned about total resource utilization, content with the knowledge that 100% of the stated resources of the device are available for use (ie: the design will route up to 100% logic utilization). In addition, much of the partitioning of a design onto multiple devices should be handled automatically by the compiler, requiring at most only high-level directives from the user.

In addition to device utilization limitations, most FPGA compilers take a significantly long time to place and route the circuit onto the FPGA when a large percentage of the chip is utilized. For example, it took over seven minutes to place and route a 16-bit multiplier to a Xilinx 4010 FPGA with 59% device utilization on a Hewlett Packard 735/125 workstation. With a large system consisting of several FPGAs the time to recompile the entire design can become extensive making the use of an iterative design style tedious. This is a disadvantage to reconfigurable systems since DSP processors allow simple design changes through software recompilation, taking usually only a few seconds. FPGA compilers should also be capable of routing a completely utilized device in only a few seconds.

A good example of a design that attempts to overcome these limitations is the Teramac reconfigurable hardware system designed at Hewlett-Packard Laboratories [2]. The Teramac is a configurable hardware system capable of executing synchronous logic designs of up to one million gates at rates up to 1 MHz. It utilizes custom FPGAs organized into multi-chip-modules (MCMs) placed on PCBs. Each FPGA can usually be placed and routed in 3 seconds or less and is designed to achieve high utilization. In addition, partitioning of a design among the FPGAs and MCMs is automated and invisible to the designer. The great advantage provided by the Teramac to the designer is that the designer no longer must concentrate on what will fit where and on what FPGA but must only consider the total resource requirements of the design in terms of the number of gates required. This eases considerably the burden placed on the designer thus allowing a much shorter design-cycle. In addition, an iterative design style is possible using the Teramac since recompilation of even large designs can be performed in only a few minutes.

High-level design support

One reason that DSP processor system design is faster than designs implemented with custom hardware is that DSP processors are software-programmable and hence the design can be implemented at a level that is closer to the actual algorithm. Custom hardware designs however are commonly designed using schematic capture utilities which require the designer to think at a level much lower than the actual flow of the algorithm, often immersing the designer in details unrelated to the algorithm being implemented. Some success at removing the burden of this detail from the designer has been achieved through the use of high-level hardware design languages such as VHDL and the inclusion of DSP macros such as FIR filters within the design language. However, much work still needs to be done in this area to allow the designer to work at a level closer to the algorithm flow when designing for reconfigurable hardware. This is largely the problem of hardware synthesis from high-level languages and is a research area within itself.

Application debugging

One of the current difficulties in developing applications for reconfigurable systems is the lack of adequate debug support for the currently available systems. With a DSP processor a designer can single-step through the code using the actual part or a simulator and can examine the value of each variable at the end of each clock cycle. This is a powerful tool that can significantly reduce the time required to develop an application. With many reconfigurable hardware boards based on FPGAs no such capabilities are present. Often the only visibility the designer has is through a logic analyzer attached to the pins of the device with even the ability to single-step the system clock lacking on some boards. This makes it extremely difficult and tedious for a designer to debug a design once it has been placed onto actual hardware. Designers are often required to build special test hardware which is time consuming and which can alter the operation of the system being tested.

These problems need not exist however as some FPGAs support the ability to be single-stepped with state readback of all flip-flops within the device through serial scanning. An example of the use of this feature can be seen in the FPGA-based Splash 2 computer software [4]. This feature makes it possible for hardware to be tested by scanning back the state of the system after every clock cycle or every n clock cycles. Each hardware variable can then be examined to track system operation. With the CLAy FPGAs additional flexibility is provided since the FPGA can be partially reconfigured to allow both state readback and writing. This provides the advantage of being able to both read and update the values of internal state variables. Thus, much of the flexibility in debugging provided by the DSP processor can be achieved with properly designed reconfigurable systems based on FPGAs. This is a critical point as it can significantly reduce the time to develop a complete application on a board. Most current reconfigurable systems, however, are being developed without considerations to test and debug abilities, hence the need to emphasize this point.

5.2 Future Work

This thesis has shown the possibility of using reconfigurable hardware based on FPGAs to obtain acceptable performance increases over the DSP processor in the area of digital signal processing. Only the overall performance of the systems was considered without regard to cost, power, or other performance metrics. This section discusses suggestions for further work in this area.

5.2.1 Power studies

Digital signal processing systems are often used in mobile applications where power is at a premium. Hence, a major consideration of many DSP systems is their power usage. Further work could profitably be done to study the power consumption of various DSP architectures when implemented on FPGAs. This work could include power studies for various multiplication implementation alternatives as well as overall system power usage and methods specific to FPGAs and reconfigurable hardware useful in reducing power consumption. One possibility for reconfigurable hardware specific power savings, for example, would be to reconfigure the hardware to a lower power state when particular portions are not in use.

5.2.2 Future devices and further multiplication studies

New FPGA devices are currently being developed which will allow even more powerful applications to be implemented on a single chip. The largest Altera 10k series parts, for example, will have well over 4 times the logic resources of the 81188 part used in this thesis along with on-chip memory that can be configured to be of different sizes. The new chips should also perform 75% faster than Altera's current FPGAs. These chips will, for example, make it possible to implement the entire 2-D 3x3 convolution system as discussed in this thesis on a single chip with an increase in speed also being obtained. The new 10k series parts should also reconfigure faster due to the use of newer high-speed parallel configuration interfaces. Similar advances in speed and density are also being made by Xilinx and National Semiconductor as well as other FPGA manufacturers. Thus, the performance of the systems implemented in this thesis can be expected to be easily surpassed by the newer devices, hence further increasing the level of FPGA-based DSP system performance over DSP processor system performance. Further work utilizing the new special features of these devices could be done to help evaluate the usefulness of the new features relative to DSP applications.

Multiplication studies using features of new FPGA devices could also be performed to evaluate their performance relative to previous FPGAs. In addition, a particularly attractive idea that should be investigated is the possibility of combining the performance of the VLSI multiplier with the performance of the FPGA. This could be done by placing VLSI multipliers onto the same die as FPGA resources forming a multi-chip-module or a new type of hybrid-chip. This would allow the high-speed multiplication possible with a VLSI multiplier along with the custom control and datapath that can be provided by the FPGA. The success of such an idea is hinted at by the results obtained using the external complex multiplier in the FFT implementation discussed in this thesis. Having the multiplier on chip with the FPGA resources would eliminate the chip-to-chip delays and the added I/O constraints that are present with an external multiplier chip.

5.3 Thesis conclusion

This thesis has shown that FPGA-based DSP systems can generally be expected to obtain an order of magnitude of performance increase over a DSP processor-based system. The FPGA-based systems are able to provide this performance increase due to their increased use of concurrency and specialization. Increased concurrency can be found in the use of multiple data paths and execution units such as multipliers while increased specialization is found in the replacement of more flexible control hardware and datapaths with ones specific to the application being implemented. Much of the architectural enhancements of the DSP processor are aimed at removing the control overhead associated with software-based control. The FPGA is able to take this one step further in that it is able to specialize the control and datapath to each application individually. Use of external components such as custom VLSI multipliers were also shown to be of value in increasing the performance of FGPA-based systems. Future devices hold promise for yet further performance increases and increased ease of use.

APPENDIX A

VHDL CODE AND SYNTHESIZED SCHEMATICS

This appendix contains all of the VHDL code for the processors used in the 3x3 and 5x5 2D image convolution filters along with their schematics produced through synthesis.

A.1 VHDL Code for 3x3 Mean Filter Processor 1

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- First of three processors in 3x3 image filter
-- Clears accumulator on DC = 00

-- For RIPP10 board:
-- Uses global bus lines 11-8 for cntrl(3 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 19-12 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk
-- Placed and routed to an odd numbered FPGA on RIPP10 board


entity proc1 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl20,gl21,gl22,gl23,gl24,gl25           : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
       fast1,fast2,fast3,rdy,cs                            : in std_logic;
       clk :in std_logic;
       cntrl : in std_logic_vector (3 Downto 0);
       datain : in std_logic_vector (7 Downto 0);
       dataout : out std_logic_vector (7 Downto 0));
```

```
end proc1;

architecture proc1_arch of proc1 is

  signal data11,data12,data13 : std_logic_vector (8 Downto 0) := "000000000";
  signal coeff11,coeff12,coeff13 : std_logic_vector (8 Downto 0) := "000000000";
  signal prod11,prod12,prod13 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum1,sum2 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum3,temp,accumulator : std_logic_vector (19 Downto 0)
    := "00000000000000000000";
  signal Aclk,clkI,tclka,tclkb : std_logic := '0';
  signal tclk1,tclk2,tclk3 : std_logic := '0';
  signal Clk1,Clk2,Clk3 : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 16);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;

  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= accumulator + ("00" & sum2);

  load_clks:
  process(clkI,cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI = '1') then
      Clk1 <= tclk1;
      Clk2 <= tclk2;
      Clk3 <= tclk3;

      if (cntrl(1 Downto 0) = "00") then
        tclk2 <= '0';
        tclk3 <= '0';
        if (tclk1 = '1') then
          tclk1 <= tclk1;
        else
          tclk1 <= '1';
        end if;
```

```
    elsif (cntrl(1 Downto 0) = "01") then
      tclk1 <= '0';
      tclk3 <= '0';
      if (tclk2 = '1') then
        tclk2 <= tclk2;
      else
        tclk2 <= '1';
      end if;
    elsif (cntrl(1 Downto 0) = "10") then
      tclk1 <= '0';
      tclk2 <= '0';
      if (tclk3 = '1') then
        tclk3 <= tclk3;
      else
        tclk3 <= '1';
      end if;
    else
      tclk1 <= '0';
      tclk2 <= '0';
      tclk3 <= '0';
    end if;
  end if;
end process load_clks;


accum_clk:
process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
begin
  if (clkI'event and clkI = '1') then
    Aclk <= tclkb;
    tclkb <= tclka;
    if (cntrl(1 Downto 0) = "10") then -- Load accum. on data set three
      if (tclka = '1') then
        tclka <= tclka;
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(3) nor cntrl(2)) = '1' or (cntrl(1) and cntrl(0)) = '1') then
      accumulator <= "00" & sum2; -- Clear the accumulator when
    else                          -- DC = 00 or BA = 11
      accumulator <= sum3;
```

```
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(3 downto 0) = "1100") then
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or
            ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(3),cntrl(2))
begin
  if (cntrl(3 Downto 2) = "00") then
    coeff11 <= "000100000";
    coeff12 <= "000100000";
    coeff13 <= "000100000";
  elsif (cntrl(3 Downto 2) = "01") then
    coeff11 <= "000100000";
    coeff12 <= "000000000";
    coeff13 <= "000100000";
  elsif (cntrl(3 Downto 2) = "10") then
    coeff11 <= "000100000";
    coeff12 <= "000100000";
    coeff13 <= "000100000";
  else
    coeff11 <= "000000000";
    coeff12 <= "000000000";
    coeff13 <= "000000000";
  end if;
end process coefficient_select;

input1_latch:  -- Latch input #1 to the system
process (Clk1)
begin
  if (Clk1'event and Clk1='1') then
    if (cntrl(1 Downto 0) = "00") then
      data11 <= '0' & datain;
    end if;
  end if;
```

```
  end process input1_latch;

  input2_latch:  -- Latch input #2 to the system
  process (Clk2)
  begin
    if (Clk2'event and Clk2='1') then
      if (cntrl(1 Downto 0) = "01") then
        data12 <= '0' & datain;
      end if;
    end if;
  end process input2_latch;

  input3_latch:  -- Latch input #3 to the system
  process (Clk3)
  begin
    if (Clk3'event and Clk3='1') then
      if (cntrl(1 Downto 0) = "10") then
        data13 <= '0' & datain;
      end if;
    end if;
  end process input3_latch;

end proc1_arch;
```

A.2 VHDL Code for 3x3 Mean Filter Processor 2

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Second of three processors in 3x3 image filter
-- Clears accumulator on DC = 01

-- For RIPP10 board:
-- Uses global bus lines 12-8 for cntrl(4 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 20-13 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk
-- Placed and routed to an even numbered FPGA on RIPP10 board

entity proc2 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
```

```
      gr31,r00,r01,r02,r03,r04                            : in std_logic;
      r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
      r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
      r29,r30,r31,gl20,gl21,gl22,gl23,gl24,gl25           : in std_logic;
      gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
      fast1,fast2,fast3,rdy,cs                            : in std_logic;
      clk :in std_logic;
      cntrl : in std_logic_vector (4 Downto 0);
      datain : in std_logic_vector (7 Downto 0);
      dataout : out std_logic_vector (7 Downto 0));
end proc2;

architecture proc2_arch of proc2 is

  signal data11,data12,data13 : std_logic_vector (8 Downto 0) := "000000000";
  signal coeff11,coeff12,coeff13 : std_logic_vector (8 Downto 0) := "000000000";
  signal prod11,prod12,prod13 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum1,sum2 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum3,temp,accumulator : std_logic_vector (19 Downto 0)
    := "00000000000000000000";
  signal Aclk,clkI,tclk : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 16);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;

  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= accumulator + ("00" & sum2);

  accum_clk:
  process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
  begin
    if (clkI'event and clkI = '1') then
      Aclk <= tclk;
      if (cntrl(1 Downto 0) = "10" and cntrl(4) = '1') then
        if (tclk = '1') then  -- Load accum. on data set three
          tclk <= tclk;
```

```
      else
        tclk <= '1';
      end if;
    else
      tclk <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if (((not cntrl(3)) and cntrl(2)) = '1' or (cntrl(1) and cntrl(0)) = '1') then
      accumulator <= "00" & sum2; -- Clear the accumulator when
    else                           -- DC = 01 or BA = 11
      accumulator <= sum3;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(3 downto 0) = "1101") then
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or
            ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(3),cntrl(2))
begin
  if (cntrl(3 Downto 2) = "01") then
    coeff11 <= "000100000";
    coeff12 <= "000100000";
    coeff13 <= "000100000";
  elsif (cntrl(3 Downto 2) = "10") then
    coeff11 <= "000100000";
    coeff12 <= "000000000";
    coeff13 <= "000100000";
  elsif (cntrl(3 Downto 2) = "00") then
    coeff11 <= "000100000";
```

```
      coeff12 <= "000100000";
      coeff13 <= "000100000";
    else
      coeff11 <= "000000000";
      coeff12 <= "000000000";
      coeff13 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch the inputs to the system
  process (clkI,cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(1 Downto 0) = "00") then
        data11 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "01") then
        data12 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "10") then
        data13 <= '0'& datain;
      end if;
    end if;
  end process inputs_latch;

end proc2_arch;
```

A.3 VHDL Code for 3x3 Mean Filter Processor 3

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Third of three processors in 3x3 image filter
-- Clears accumulator on DC = 10

-- For RIPP10 board:
-- Uses global bus lines 12-8 for cntrl(4 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 20-13 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk
-- Placed and routed to an odd numbered FPGA on RIPP10 board

entity proc3 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
```

```
      gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
      gr31,r00,r01,r02,r03,r04                            : in std_logic;
      r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
      r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
      r29,r30,r31,gl20,gl21,gl22,gl23,gl24,gl25           : in std_logic;
      gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
      fast1,fast2,fast3,rdy,cs                            : in std_logic;
      clk :in std_logic;
      cntrl : in std_logic_vector (4 Downto 0);
      datain : in std_logic_vector (7 Downto 0);
      dataout : out std_logic_vector (7 Downto 0));
end proc3;

architecture proc3_arch of proc3 is

  signal data11,data12,data13 : std_logic_vector (8 Downto 0) := "000000000";
  signal coeff11,coeff12,coeff13 : std_logic_vector (8 Downto 0) := "000000000";
  signal prod11,prod12,prod13 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum1,sum2 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum3,temp,accumulator : std_logic_vector (19 Downto 0)
    := "00000000000000000000";
  signal Aclk,clkI,tclk : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 16);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;

  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= accumulator + ("00" & sum2);

  accum_clk:
  process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
  begin
    if (clkI'event and clkI = '1') then
      Aclk <= tclk;
      if (cntrl(1 Downto 0) = "10" and cntrl(4) = '1') then
        if (tclk = '1') then  -- Load accum. on data set three
```

```
        tclk <= tclk;
      else
        tclk <= '1';
      end if;
    else
      tclk <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(3) and (not cntrl(2))) = '1' or (cntrl(1) and cntrl(0)) = '1') then
      accumulator <= "00" & sum2; -- Clear the accumulator when
    else                          -- DC = 10 or BA = 11
      accumulator <= sum3;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(3 downto 0) = "1110") then
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or
            ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(3),cntrl(2))
begin
  if (cntrl(3 Downto 2) = "10") then
    coeff11 <= "000100000";
    coeff12 <= "000100000";
    coeff13 <= "000100000";
  elsif (cntrl(3 Downto 2) = "00") then
    coeff11 <= "000100000";
    coeff12 <= "000000000";
    coeff13 <= "000100000";
  elsif (cntrl(3 Downto 2) = "01") then
```

```
      coeff11 <= "000100000";
      coeff12 <= "000100000";
      coeff13 <= "000100000";
    else
      coeff11 <= "000000000";
      coeff12 <= "000000000";
      coeff13 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch the inputs to the system
  process (clkI)
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(1 Downto 0) = "00") then
        data11 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "01") then
        data12 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "10") then
        data13 <= '0'& datain;
      end if;
    end if;
  end process inputs_latch;

end proc3_arch;
```

A.4 VHDL Code for 3x3 Edge Detection Filter Processor 1

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- First of three processors in 3x3 image filter
-- Clears accumulator on DC = 00

-- For RIPP10 board:
-- Uses global bus lines 12-8 for cntrl(4 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 20-13 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk
-- Placed and routed to an odd numbered FPGA on RIPP10 board

entity proc1 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
```

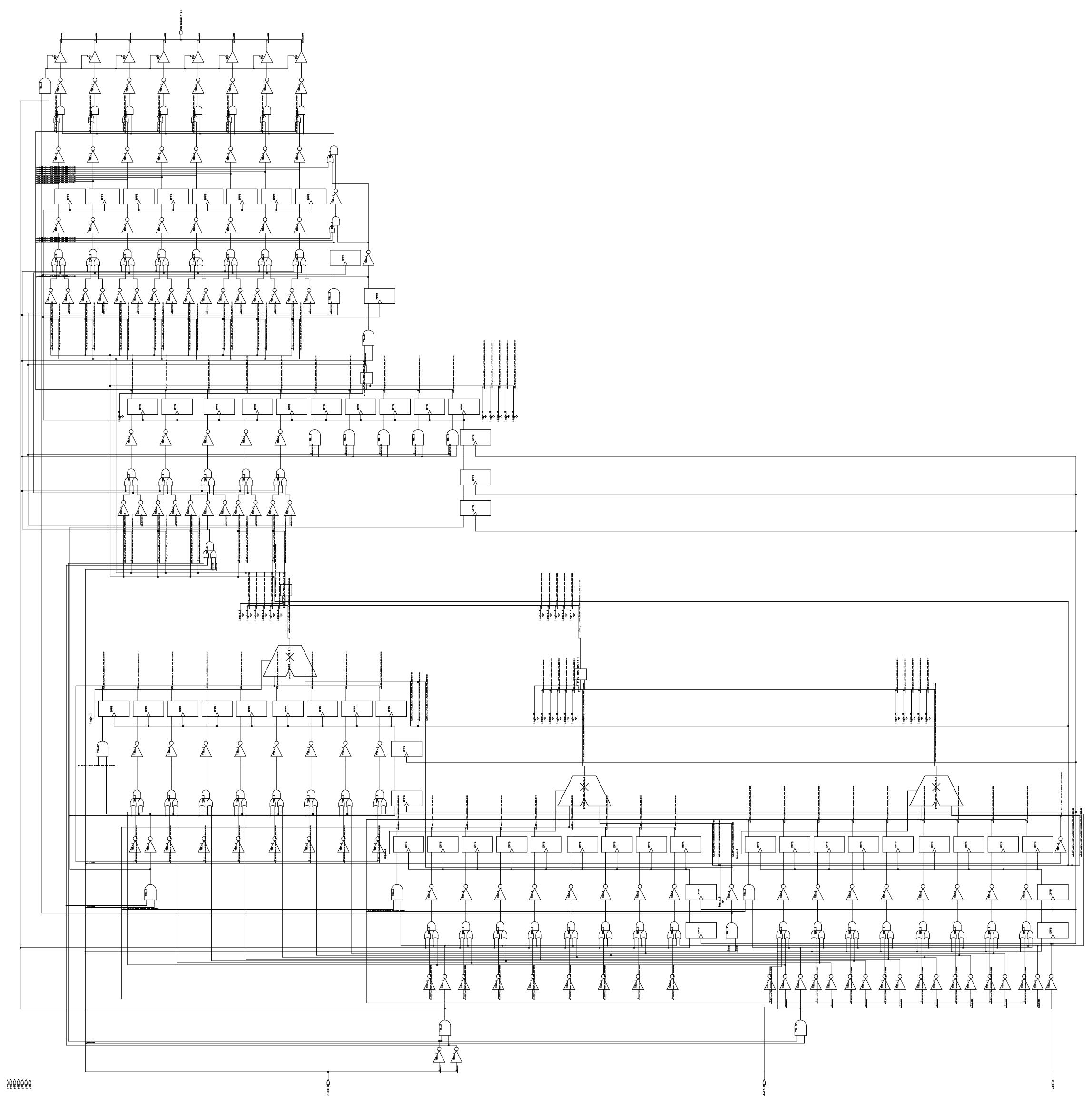


Figure A.1: Synthesized Schematic for 3x3 Mean Filter Processor 1

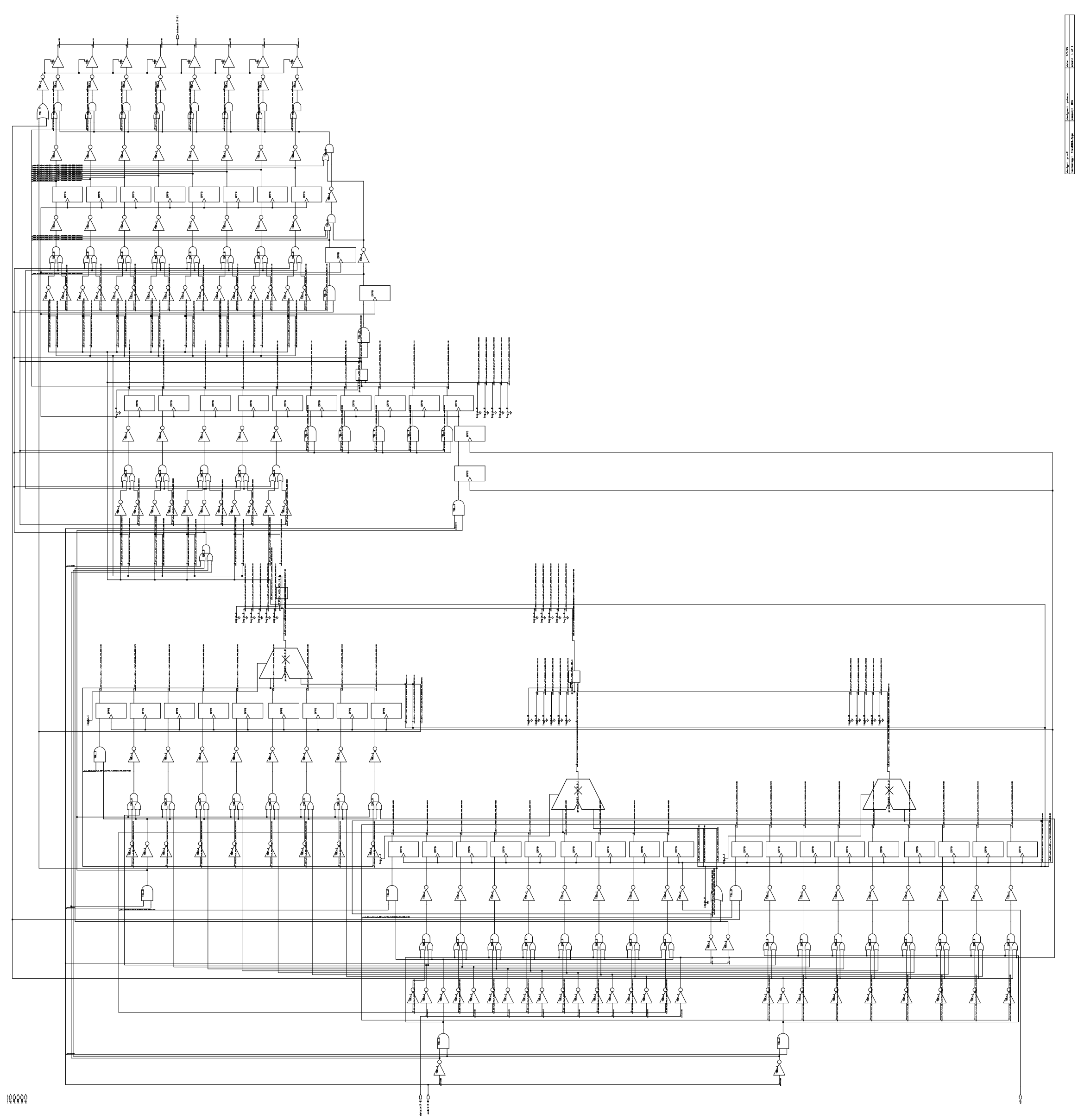


Figure A.2: Synthesized Schematic for 3x3 Mean Filter Processor 2

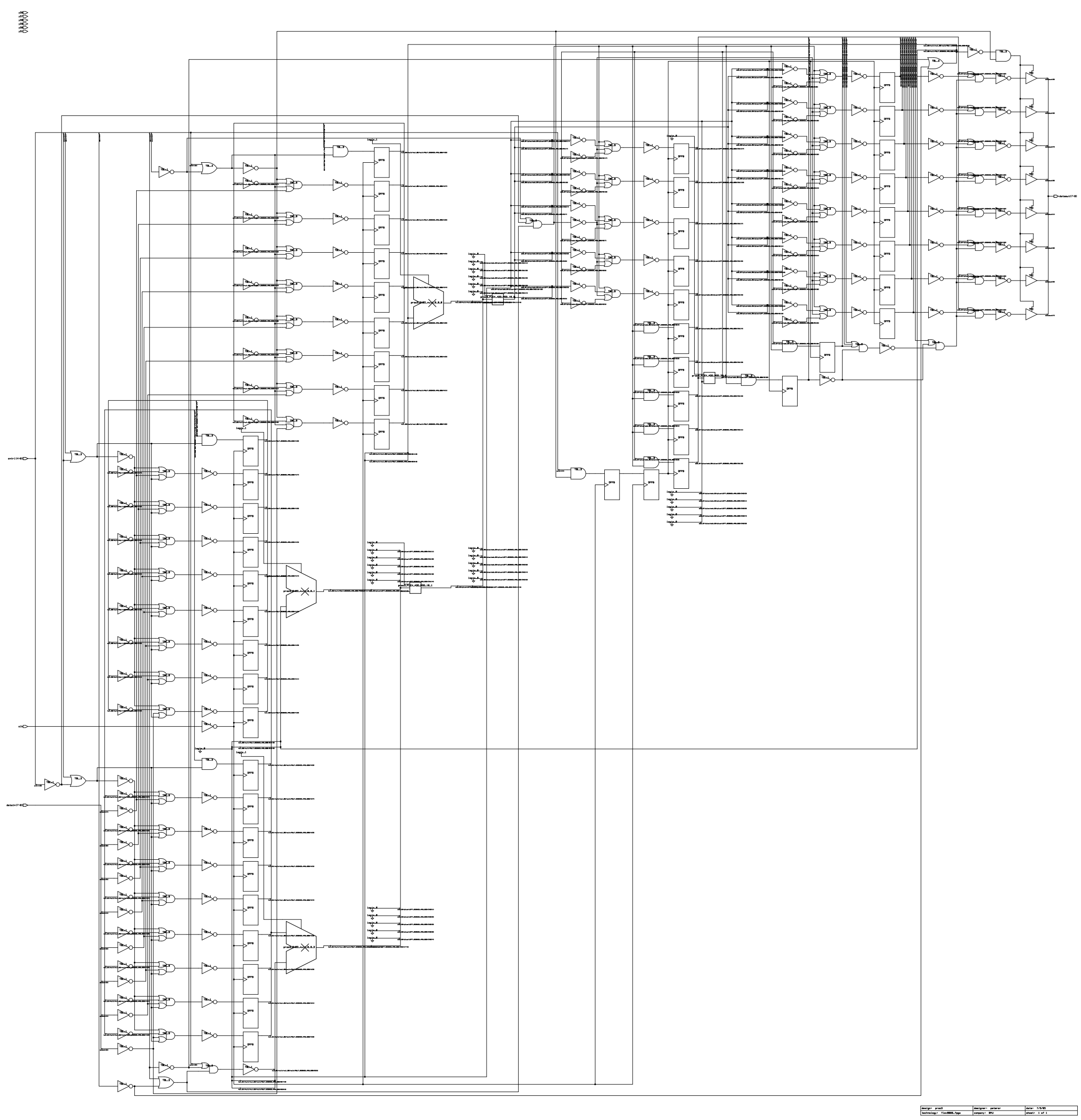


Figure A.3: Synthesized Schematic for 3x3 Mean Filter Processor 3

```
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl21,gl22,gl23,gl24,gl25                : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
       fast1,fast2,fast3,rdy,cs                            : in std_logic;
       clk :in std_logic;
       cntrl : in std_logic_vector (4 Downto 0);
       datain : in std_logic_vector (7 Downto 0);
       dataout : out std_logic_vector (7 Downto 0));
end proc1;

architecture proc1_arch of proc1 is

  signal data11,data12,data13 : std_logic_vector (8 Downto 0) := "000000000";
  signal coeff11,coeff12,coeff13 : std_logic_vector (8 Downto 0) := "000000000";
  signal prod11,prod12,prod13 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum1,sum2 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum3,temp,accumulator : std_logic_vector (19 Downto 0)
    := "00000000000000000000";
  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (7 downto 0) := "00000000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 12);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;

  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= accumulator + ( sum2(17) & sum2(17) & sum2);


  accum_clk:
  process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
  begin
    if (clkI'event and clkI = '1') then
```

```
    Aclk <= tclka;
    if (cntrl(1 Downto 0) = "10" and cntrl(4) = '1') then
      if (tclka = '1') then  -- Load accum. on data set three
        tclka <= tclka;
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(3) nor cntrl(2)) = '1' or (cntrl(1) and cntrl(0)) = '1') then
      accumulator <= sum2(17) & sum2(17) & sum2; -- Clear the accumulator when
    else                          -- DC = 00 or BA = 11
      accumulator <= sum3;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(3 downto 0) = "1100") then
    if (ofdetect(7 Downto 6) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(7) or ofdetect(6) or ofdetect(5) or ofdetect(4) or ofdetect(3) or
            ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(11 Downto 4);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(3),cntrl(2))
begin
  if (cntrl(3 Downto 2) = "00") then
    coeff11 <= "111110000";
    coeff12 <= "111110000";
    coeff13 <= "111110000";
  elsif (cntrl(3 Downto 2) = "01") then
    coeff11 <= "111110000";
```

```
      coeff12 <= "010000000";
      coeff13 <= "111110000";
    elsif (cntrl(3 Downto 2) = "10") then
      coeff11 <= "111110000";
      coeff12 <= "111110000";
      coeff13 <= "111110000";
    else
      coeff11 <= "000000000";
      coeff12 <= "000000000";
      coeff13 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch inputs to the system
  process (clkI,cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(1 Downto 0) = "00") then
        data11 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "01") then
        data12 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "10") then
        data13 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc1_arch;
```

A.5 VHDL Code for 3x3 Edge Detection Filter Processor 2

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Second of three processors in 3x3 image filter
-- Clears accumulator on DC = 01

-- For RIPP10 board:
-- Uses global bus lines 12-8 for cntrl(4 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 20-13 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk
-- Placed and routed to an odd numbered FPGA on RIPP10 board

entity proc2 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
```

```
      y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
      y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
      gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
      gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
      gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
      gr31,r00,r01,r02,r03,r04                            : in std_logic;
      r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
      r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
      r29,r30,r31,gl21,gl22,gl23,gl24,gl25                : in std_logic;
      gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
      fast1,fast2,fast3,rdy,cs                            : in std_logic;
      clk :in std_logic;
      cntrl : in std_logic_vector (4 Downto 0);
      datain : in std_logic_vector (7 Downto 0);
      dataout : out std_logic_vector (7 Downto 0));
end proc2;

architecture proc2_arch of proc2 is

  signal data11,data12,data13 : std_logic_vector (8 Downto 0) := "000000000";
  signal coeff11,coeff12,coeff13 : std_logic_vector (8 Downto 0) := "000000000";
  signal prod11,prod12,prod13 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum1,sum2 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum3,temp,accumulator : std_logic_vector (19 Downto 0)
    := "00000000000000000000";
  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (7 downto 0) := "00000000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 12);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;

  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= accumulator + ( sum2(17) & sum2(17) & sum2);


  accum_clk:
  process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
```

```
begin
  if (clkI'event and clkI = '1') then
    Aclk <= tclka;
    if (cntrl(1 Downto 0) = "10" and cntrl(4) = '1') then
      if (tclka = '1') then  -- Load accum. on data set three
        tclka <= tclka;
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if (((not cntrl(3)) and cntrl(2)) = '1' or (cntrl(1) and cntrl(0)) = '1') then
      accumulator <= sum2(17) & sum2(17) & sum2; -- Clear the accumulator when
    else                           -- DC = 01 or BA = 11
      accumulator <= sum3;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(3 downto 0) = "1101") then
    if (ofdetect(7 Downto 6) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(7) or ofdetect(6) or ofdetect(5) or ofdetect(4) or ofdetect(3) or
            ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(11 Downto 4);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(3),cntrl(2))
begin
  if (cntrl(3 Downto 2) = "01") then
    coeff11 <= "111110000";
    coeff12 <= "111110000";
    coeff13 <= "111110000";
```

```
    elsif (cntrl(3 Downto 2) = "10") then
      coeff11 <= "111110000";
      coeff12 <= "010000000";
      coeff13 <= "111110000";
    elsif (cntrl(3 Downto 2) = "00") then
      coeff11 <= "111110000";
      coeff12 <= "111110000";
      coeff13 <= "111110000";
    else
      coeff11 <= "000000000";
      coeff12 <= "000000000";
      coeff13 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch inputs to the system
  process (clkI,cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(1 Downto 0) = "00") then
        data11 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "01") then
        data12 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "10") then
        data13 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc2_arch;
```

A.6 VHDL Code for 3x3 Edge Detection Filter Processor 3

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Third of three processors in 3x3 image filter
-- Clears accumulator on DC = 10

-- For RIPP10 board:
-- Uses global bus lines 12-8 for cntrl(4 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 20-13 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk
-- Placed and routed to an odd numbered FPGA on RIPP10 board

entity proc3 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
```

```
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl21,gl22,gl23,gl24,gl25                : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
       fast1,fast2,fast3,rdy,cs                            : in std_logic;
       clk :in std_logic;
       cntrl : in std_logic_vector (4 Downto 0);
       datain : in std_logic_vector (7 Downto 0);
       dataout : out std_logic_vector (7 Downto 0));
end proc3;

architecture proc3_arch of proc3 is

  signal data11,data12,data13 : std_logic_vector (8 Downto 0) := "000000000";
  signal coeff11,coeff12,coeff13 : std_logic_vector (8 Downto 0) := "000000000";
  signal prod11,prod12,prod13 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum1,sum2 : std_logic_vector (17 Downto 0)
    := "000000000000000000";
  signal sum3,temp,accumulator : std_logic_vector (19 Downto 0)
    := "00000000000000000000";
  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (7 downto 0) := "00000000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 12);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;

  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= accumulator + ( sum2(17) & sum2(17) & sum2);
```

```
accum_clk:
process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
begin
  if (clkI'event and clkI = '1') then
    Aclk <= tclka;
    if (cntrl(1 Downto 0) = "10" and cntrl(4) = '1') then
      if (tclka = '1') then  -- Load accum. on data set three
        tclka <= tclka;
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(3) and (not cntrl(2))) = '1' or (cntrl(1) and cntrl(0)) = '1') then
      accumulator <= sum2(17) & sum2(17) & sum2; -- Clear the accumulator when
    else                             -- DC = 10 or BA = 11
      accumulator <= sum3;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(3 downto 0) = "1110") then
    if (ofdetect(7 Downto 6) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(7) or ofdetect(6) or ofdetect(5) or ofdetect(4) or ofdetect(3) or
            ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(11 Downto 4);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(3),cntrl(2))
begin
  if (cntrl(3 Downto 2) = "10") then
    coeff11 <= "111110000";
```

```
      coeff12 <= "111110000";
      coeff13 <= "111110000";
    elsif (cntrl(3 Downto 2) = "00") then
      coeff11 <= "111110000";
      coeff12 <= "010000000";
      coeff13 <= "111110000";
    elsif (cntrl(3 Downto 2) = "01") then
      coeff11 <= "111110000";
      coeff12 <= "111110000";
      coeff13 <= "111110000";
    else
      coeff11 <= "000000000";
      coeff12 <= "000000000";
      coeff13 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch inputs to the system
  process (clkI,cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(1 Downto 0) = "00") then
        data11 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "01") then
        data12 <= '0' & datain;
      elsif (cntrl(1 Downto 0) = "10") then
        data13 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc3_arch;
```

A.7 VHDL Code for 5x5 Mean Filter Processor 1

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- First of five processors in 5x5 image filter
-- Clear accumulator when CBA = 111 or FED = 000

-- For RIPP10 board:
-- Uses global bus lines 14-8 for cntrl(6 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 22-15 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk
```

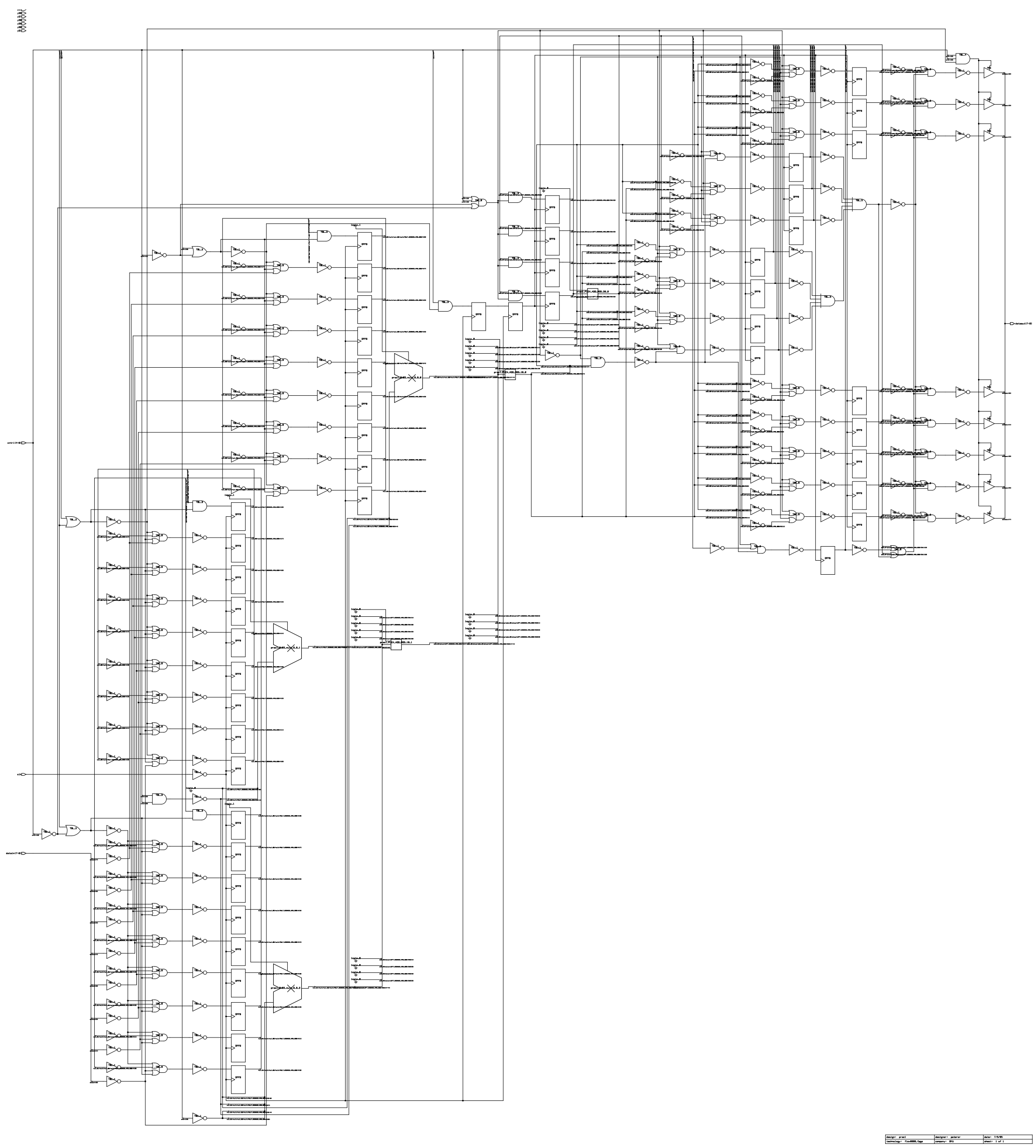


Figure A.4: Synthesized Schematic for 3x3 Edge Detection Filter Processor 1

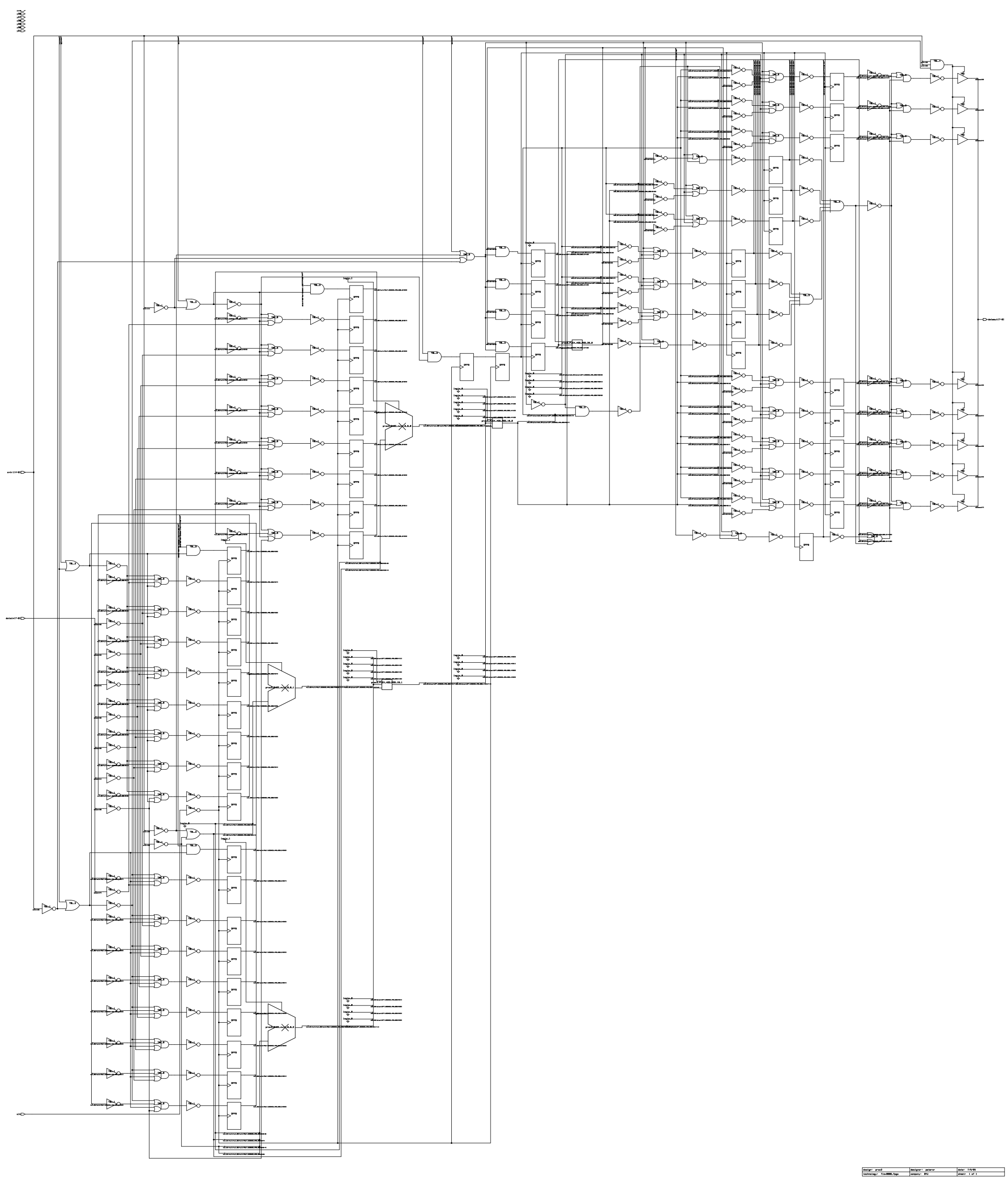


Figure A.5: Synthesized Schematic for 3x3 Edge Detection Filter Processor 2

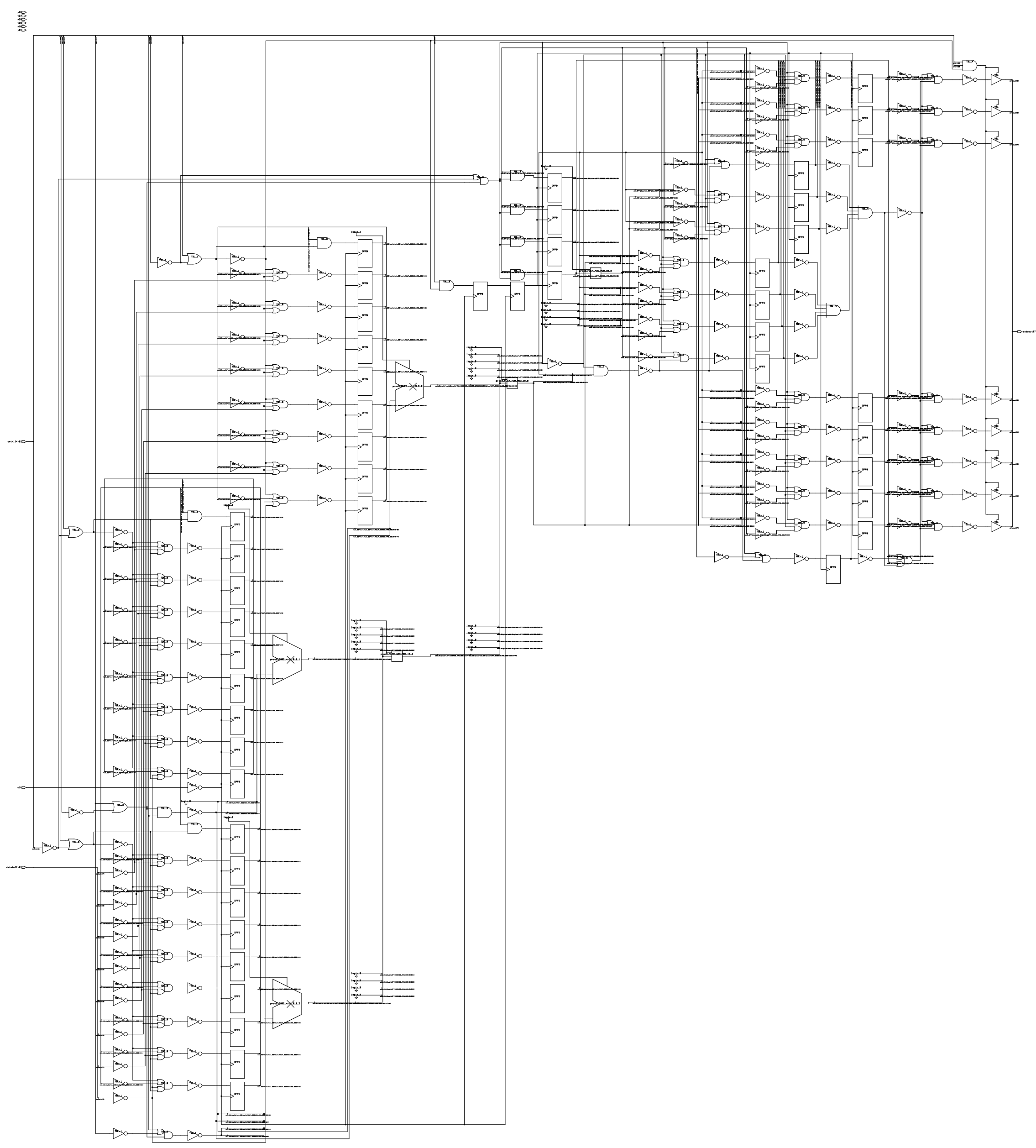


Figure A.6: Synthesized Schematic for 3x3 Edge Detection Filter Processor 3

```
entity proc1 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl23,gl24,gl25                          : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
       fast1,fast2,fast3,rdy,cs                            : in std_logic;
       clk :in std_logic;
       cntrl : in std_logic_vector (6 Downto 0);
       datain : in std_logic_vector (7 Downto 0);
       dataout : out std_logic_vector (7 Downto 0));
end proc1;

architecture proc1_arch of proc1 is

  signal data11,data12,data13,data14,data15 : std_logic_vector (8 Downto 0);
  signal coeff11,coeff12,coeff13,coeff14,coeff15 : std_logic_vector (8 Downto 0);
  signal prod11,prod12,prod13,prod14,prod15 : std_logic_vector (17 Downto 0);
  signal sum1,sum2,sum3 : std_logic_vector (17 Downto 0);
  signal sum4 : std_logic_vector (18 Downto 0);
  signal sum5,temp,accumulator : std_logic_vector (19 Downto 0);

  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 16);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;
  prod14 <= coeff14 * data14;
  prod15 <= coeff15 * data15;


  -- Add the products
```

```
sum1 <= prod11 + prod12;
sum2 <= prod13 + sum1;
sum3 <= sum2 + prod14;
sum4 <= ('0'&sum3) + ('0'&prod15);
sum5 <= accumulator + ('0'&sum4);


accum_clk:
process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
begin
  if (clkI'event and clkI = '1') then
    Aclk <= tclka;
    if (cntrl(2 Downto 0) = "100" and cntrl(6) = '1') then
      if (tclka = '1') then  -- Load accum. on data set five
        tclka <= tclka;
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(5 Downto 3) = "000")  or (cntrl(2 Downto 0) = "111")) then
      accumulator <= '0'&sum4; -- Clear accumulator when CBA = 111 or FED = 000
      else
        accumulator <= sum5;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(5 downto 0) = "111000") then
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;
```

```
coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(5),cntrl(4),cntrl(3))
begin
  if (cntrl(5 Downto 3) = "000") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "001") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "010") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "011") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "100") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  else
    coeff11 <= "000000000";
    coeff12 <= "000000000";
    coeff13 <= "000000000";
    coeff14 <= "000000000";
    coeff15 <= "000000000";
  end if;
end process coefficient_select;

inputs_latch:  -- Latch inputs to the system
process (clkI,cntrl(2),cntrl(1),cntrl(0))
begin
  if (clkI'event and clkI='1') then
    if (cntrl(2 Downto 0) = "000") then
      data11 <= '0' & datain;
    elsif (cntrl(2 Downto 0) = "001") then
      data12 <= '0' & datain;
```

```
      elsif (cntrl(2 Downto 0) = "010") then
        data13 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "011") then
        data14 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "100") then
        data15 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc1_arch;
```

A.8 VHDL Code for 5x5 Mean Filter Processor 2

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Second of five processors in 5x5 image filter
-- Clear accumulator when CBA = 111 or FED = 001

-- For RIPP10 board:
-- Uses global bus lines 14-8 for cntrl(6 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 22-15 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk


entity proc2 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl23,gl24,gl25                          : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
       fast1,fast2,fast3,rdy,cs                            : in std_logic;
       clk :in std_logic;
       cntrl : in std_logic_vector (6 Downto 0);
       datain : in std_logic_vector (7 Downto 0);
       dataout : out std_logic_vector (7 Downto 0));
end proc2;
```

```
architecture proc2_arch of proc2 is

  signal data11,data12,data13,data14,data15 : std_logic_vector (8 Downto 0);
  signal coeff11,coeff12,coeff13,coeff14,coeff15 : std_logic_vector (8 Downto 0);
  signal prod11,prod12,prod13,prod14,prod15 : std_logic_vector (17 Downto 0);
  signal sum1,sum2,sum3 : std_logic_vector (17 Downto 0);
  signal sum4 : std_logic_vector (18 Downto 0);
  signal sum5,temp,accumulator : std_logic_vector (19 Downto 0);

  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 16);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;
  prod14 <= coeff14 * data14;
  prod15 <= coeff15 * data15;


  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= sum2 + prod14;
  sum4 <= ('0'&sum3) + ('0'&prod15);
  sum5 <= accumulator + ('0'&sum4);


  accum_clk:
  process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
  begin
    if (clkI'event and clkI = '1') then
      Aclk <= tclka;
      if (cntrl(2 Downto 0) = "100" and cntrl(6) = '1') then
        if (tclka = '1') then  -- Load accum. on data set five
          tclka <= tclka;
        else
          tclka <= '1';
        end if;
      else
        tclka <= '0';  -- Clear the accumulator clock
      end if;
    end if;
```

```
end process accum_clk;

load_accum:
process(Aclk,cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(5 Downto 3) = "001")  or (cntrl(2 Downto 0) = "111")) then
      accumulator <= '0'&sum4; -- Clear accumulator when CBA = 111 or FED = 001
      else
        accumulator <= sum5;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(5 downto 0) = "111001") then -- Get output when 111001
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(5),cntrl(4),cntrl(3))
begin
  if (cntrl(5 Downto 3) = "000") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "001") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "010") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
```

```
    elsif (cntrl(5 Downto 3) = "011") then
      coeff11 <= "000001010";
      coeff12 <= "000001010";
      coeff13 <= "000001010";
      coeff14 <= "000001010";
      coeff15 <= "000001010";
    elsif (cntrl(5 Downto 3) = "100") then
      coeff11 <= "000001010";
      coeff12 <= "000001010";
      coeff13 <= "000001010";
      coeff14 <= "000001010";
      coeff15 <= "000001010";
    else
      coeff11 <= "000000000";
      coeff12 <= "000000000";
      coeff13 <= "000000000";
      coeff14 <= "000000000";
      coeff15 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch inputs to the system
  process (clkI,cntrl(2),cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(2 Downto 0) = "000") then
        data11 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "001") then
        data12 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "010") then
        data13 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "011") then
        data14 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "100") then
        data15 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc2_arch;
```

A.9 VHDL Code for 5x5 Mean Filter Processor 3

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Third of five processors in 5x5 image filter
-- Clear accumulator when CBA = 111 or FED = 010
```

```
-- For RIPP10 board:
-- Uses global bus lines 14-8 for cntrl(6 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 22-15 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk


entity proc3 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl23,gl24,gl25                          : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
       fast1,fast2,fast3,rdy,cs                            : in std_logic;
       clk :in std_logic;
       cntrl : in std_logic_vector (6 Downto 0);
       datain : in std_logic_vector (7 Downto 0);
       dataout : out std_logic_vector (7 Downto 0));
end proc3;

architecture proc3_arch of proc3 is

  signal data11,data12,data13,data14,data15 : std_logic_vector (8 Downto 0);
  signal coeff11,coeff12,coeff13,coeff14,coeff15 : std_logic_vector (8 Downto 0);
  signal prod11,prod12,prod13,prod14,prod15 : std_logic_vector (17 Downto 0);
  signal sum1,sum2,sum3 : std_logic_vector (17 Downto 0);
  signal sum4 : std_logic_vector (18 Downto 0);
  signal sum5,temp,accumulator : std_logic_vector (19 Downto 0);

  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 16);

  -- Form the products
  prod11 <= coeff11 * data11;
```

```
prod12 <= coeff12 * data12;
prod13 <= coeff13 * data13;
prod14 <= coeff14 * data14;
prod15 <= coeff15 * data15;


-- Add the products
sum1 <= prod11 + prod12;
sum2 <= prod13 + sum1;
sum3 <= sum2 + prod14;
sum4 <= ('0'&sum3) + ('0'&prod15);
sum5 <= accumulator + ('0'&sum4);


accum_clk:
process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
begin
  if (clkI'event and clkI = '1') then
    Aclk <= tclka;
    if (cntrl(2 Downto 0) = "100" and cntrl(6) = '1') then
      if (tclka = '1') then  -- Load accum. on data set five
        tclka <= tclka;
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(5 Downto 3) = "010")  or (cntrl(2 Downto 0) = "111")) then
      accumulator <= '0'&sum4; -- Clear accumulator when CBA = 111 or FED = 010
      else
        accumulator <= sum5;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(5 downto 0) = "111010") then -- Get output when 111010
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
```

```
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(5),cntrl(4),cntrl(3))
begin
  if (cntrl(5 Downto 3) = "000") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "001") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "010") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "011") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "100") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  else
    coeff11 <= "000000000";
    coeff12 <= "000000000";
    coeff13 <= "000000000";
    coeff14 <= "000000000";
    coeff15 <= "000000000";
  end if;
end process coefficient_select;

inputs_latch:  -- Latch inputs to the system
```

```
  process (clkI,cntrl(2),cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(2 Downto 0) = "000") then
        data11 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "001") then
        data12 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "010") then
        data13 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "011") then
        data14 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "100") then
        data15 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc3_arch;
```

A.10 VHDL Code for 5x5 Mean Filter Processor 4

```
Library IEEE,WORK,altera;
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Fourth of five processors in 5x5 image filter
-- Clear accumulator when CBA = 111 or FED = 011

-- For RIPP10 board:
-- Uses global bus lines 14-8 for cntrl(6 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 22-15 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk


entity proc4 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl23,gl24,gl25                          : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
```

```
      fast1,fast2,fast3,rdy,cs                            : in std_logic;
      clk :in std_logic;
      cntrl : in std_logic_vector (6 Downto 0);
      datain : in std_logic_vector (7 Downto 0);
      dataout : out std_logic_vector (7 Downto 0));
end proc4;

architecture proc4_arch of proc4 is

  signal data11,data12,data13,data14,data15 : std_logic_vector (8 Downto 0);
  signal coeff11,coeff12,coeff13,coeff14,coeff15 : std_logic_vector (8 Downto 0);
  signal prod11,prod12,prod13,prod14,prod15 : std_logic_vector (17 Downto 0);
  signal sum1,sum2,sum3 : std_logic_vector (17 Downto 0);
  signal sum4 : std_logic_vector (18 Downto 0);
  signal sum5,temp,accumulator : std_logic_vector (19 Downto 0);

  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
  clkI <= not clk;

  -- Pull off bits used in overflow/underflow detection
  ofdetect <= accumulator (19 Downto 16);

  -- Form the products
  prod11 <= coeff11 * data11;
  prod12 <= coeff12 * data12;
  prod13 <= coeff13 * data13;
  prod14 <= coeff14 * data14;
  prod15 <= coeff15 * data15;


  -- Add the products
  sum1 <= prod11 + prod12;
  sum2 <= prod13 + sum1;
  sum3 <= sum2 + prod14;
  sum4 <= ('0'&sum3) + ('0'&prod15);
  sum5 <= accumulator + ('0'&sum4);


  accum_clk:
  process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
  begin
    if (clkI'event and clkI = '1') then
      Aclk <= tclka;
      if (cntrl(2 Downto 0) = "100" and cntrl(6) = '1') then
        if (tclka = '1') then  -- Load accum. on data set five
          tclka <= tclka;
```

```
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(5 Downto 3) = "011")  or (cntrl(2 Downto 0) = "111")) then
      accumulator <= '0'&sum4; -- Clear accumulator when CBA = 111 or FED = 011
      else
        accumulator <= sum5;
    end if;
  end if;
end process load_accum;

output_read:
process (cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(5 downto 0) = "111011") then -- Get output when 111011
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(5),cntrl(4),cntrl(3))
begin
  if (cntrl(5 Downto 3) = "000") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "001") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
```

```
      coeff15 <= "000001010";
    elsif (cntrl(5 Downto 3) = "010") then
      coeff11 <= "000001010";
      coeff12 <= "000001010";
      coeff13 <= "000001010";
      coeff14 <= "000001010";
      coeff15 <= "000001010";
    elsif (cntrl(5 Downto 3) = "011") then
      coeff11 <= "000001010";
      coeff12 <= "000001010";
      coeff13 <= "000001010";
      coeff14 <= "000001010";
      coeff15 <= "000001010";
    elsif (cntrl(5 Downto 3) = "100") then
      coeff11 <= "000001010";
      coeff12 <= "000001010";
      coeff13 <= "000001010";
      coeff14 <= "000001010";
      coeff15 <= "000001010";
    else
      coeff11 <= "000000000";
      coeff12 <= "000000000";
      coeff13 <= "000000000";
      coeff14 <= "000000000";
      coeff15 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch inputs to the system
  process (clkI,cntrl(2),cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(2 Downto 0) = "000") then
        data11 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "001") then
        data12 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "010") then
        data13 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "011") then
        data14 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "100") then
        data15 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc4_arch;
```

A.11 VHDL Code for 5x5 Mean Filter Processor 5

```
Library IEEE,WORK,altera;
```

```
   use IEEE.std_logic_1164.all;
   use IEEE.std_logic_arith.all;
   use IEEE.std_logic_signed.ALL;

-- Fifth of five processors in 5x5 image filter
-- Clear accumulator when CBA = 111 or FED = 100

-- For RIPP10 board:
-- Uses global bus lines 14-8 for cntrl(6 Downto 0),
-- global bus lines 7-0 for datain (7 Downto 0),
-- and global bus lines 22-15 for Dataout(7 Downto 0).
-- The fast0 line is used to provide the clk


entity proc5 is
  port(wr,oe,rcs,a0,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10,a11,a12,a13 : in std_logic;
       a14,a15,a16,a17,a18,d0,d1,d2,d3,d4,d5,d6,d7,b00     : in std_logic;
       b01,b02,b03,b04,b05,b06,b07,b08,b09,b10,b11,b12     : in std_logic;
       b13,b14,b15,y00,y01,y02,y03,y04,y05,y06,y07,y08     : in std_logic;
       y09,y10,y11,y12,y13,y14,y15,y16,y17,y18,y19,y20     : in std_logic;
       y21,y22,y23,y24,y25,y26,y27,y28,y29,y30,y31,gr00    : in std_logic;
       gr01,gr02,gr03,gr04,gr05,gr06,gr07,gr08,gr09,gr10   : in std_logic;
       gr11,gr12,gr13,gr14,gr15,gr16,gr17,gr18,gr19,gr20   : in std_logic;
       gr21,gr22,gr23,gr24,gr25,gr26,gr27,gr28,gr29,gr30   : in std_logic;
       gr31,r00,r01,r02,r03,r04                            : in std_logic;
       r05,r06,r07,r08,r09,r10,r11,r12,r13,r14,r15,r16     : in std_logic;
       r17,r18,r19,r20,r21,r22,r23,r24,r25,r26,r27,r28     : in std_logic;
       r29,r30,r31,gl23,gl24,gl25                          : in std_logic;
       gl26,gl27,gl28,gl29,gl30,gl31,gl32,gl33,gl34,gl35   : in std_logic;
       fast1,fast2,fast3,rdy,cs                            : in std_logic;
       clk :in std_logic;
       cntrl : in std_logic_vector (6 Downto 0);
       datain : in std_logic_vector (7 Downto 0);
       dataout : out std_logic_vector (7 Downto 0));
end proc5;

architecture proc5_arch of proc5 is

  signal data11,data12,data13,data14,data15 : std_logic_vector (8 Downto 0);
  signal coeff11,coeff12,coeff13,coeff14,coeff15 : std_logic_vector (8 Downto 0);
  signal prod11,prod12,prod13,prod14,prod15 : std_logic_vector (17 Downto 0);
  signal sum1,sum2,sum3 : std_logic_vector (17 Downto 0);
  signal sum4 : std_logic_vector (18 Downto 0);
  signal sum5,temp,accumulator : std_logic_vector (19 Downto 0);

  signal Aclk,clkI,tclka  : std_logic := '0';
  signal ofdetect : std_logic_vector (3 downto 0) := "0000";

begin

  -- Invert the incoming clock line
```

```
clkI <= not clk;

-- Pull off bits used in overflow/underflow detection
ofdetect <= accumulator (19 Downto 16);

-- Form the products
prod11 <= coeff11 * data11;
prod12 <= coeff12 * data12;
prod13 <= coeff13 * data13;
prod14 <= coeff14 * data14;
prod15 <= coeff15 * data15;


-- Add the products
sum1 <= prod11 + prod12;
sum2 <= prod13 + sum1;
sum3 <= sum2 + prod14;
sum4 <= ('0'&sum3) + ('0'&prod15);
sum5 <= accumulator + ('0'&sum4);


accum_clk:
process(clkI,cntrl(1),cntrl(0),cntrl(2),cntrl(3))
begin
  if (clkI'event and clkI = '1') then
    Aclk <= tclka;
    if (cntrl(2 Downto 0) = "100" and cntrl(6) = '1') then
      if (tclka = '1') then  -- Load accum. on data set five
        tclka <= tclka;
      else
        tclka <= '1';
      end if;
    else
      tclka <= '0';  -- Clear the accumulator clock
    end if;
  end if;
end process accum_clk;

load_accum:
process(Aclk,cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (Aclk'event and Aclk = '1') then
    if ((cntrl(5 Downto 3) = "100")  or (cntrl(2 Downto 0) = "111")) then
      accumulator <= '0'&sum4; -- Clear accumulator when CBA = 111 or FED = 100
      else
        accumulator <= sum5;
    end if;
  end if;
end process load_accum;

output_read:
```

```
process (cntrl(5),cntrl(4),cntrl(3),cntrl(2),cntrl(1),cntrl(0))
begin
  if (cntrl(5 downto 0) = "111100") then -- Get output when 111100
    if (ofdetect(3 Downto 2) = "11") then
      dataout <= "00000000";
    elsif ((ofdetect(3) or ofdetect(2) or ofdetect(1) or ofdetect(0)) = '1') then
      dataout <= "11111111";
    else
      dataout <= accumulator(15 Downto 8);
    end if;
  else
    dataout <= "ZZZZZZZZ";
  end if;
end process output_read;

coefficient_select: -- Select the coefficients to gate to the multipliers
process(cntrl(5),cntrl(4),cntrl(3))
begin
  if (cntrl(5 Downto 3) = "000") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "001") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "010") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "011") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  elsif (cntrl(5 Downto 3) = "100") then
    coeff11 <= "000001010";
    coeff12 <= "000001010";
    coeff13 <= "000001010";
    coeff14 <= "000001010";
    coeff15 <= "000001010";
  else
    coeff11 <= "000000000";
    coeff12 <= "000000000";
```

```
      coeff13 <= "000000000";
      coeff14 <= "000000000";
      coeff15 <= "000000000";
    end if;
  end process coefficient_select;

  inputs_latch:  -- Latch inputs to the system
  process (clkI,cntrl(2),cntrl(1),cntrl(0))
  begin
    if (clkI'event and clkI='1') then
      if (cntrl(2 Downto 0) = "000") then
        data11 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "001") then
        data12 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "010") then
        data13 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "011") then
        data14 <= '0' & datain;
      elsif (cntrl(2 Downto 0) = "100") then
        data15 <= '0' & datain;
      end if;
    end if;
  end process inputs_latch;

end proc5_arch;
```

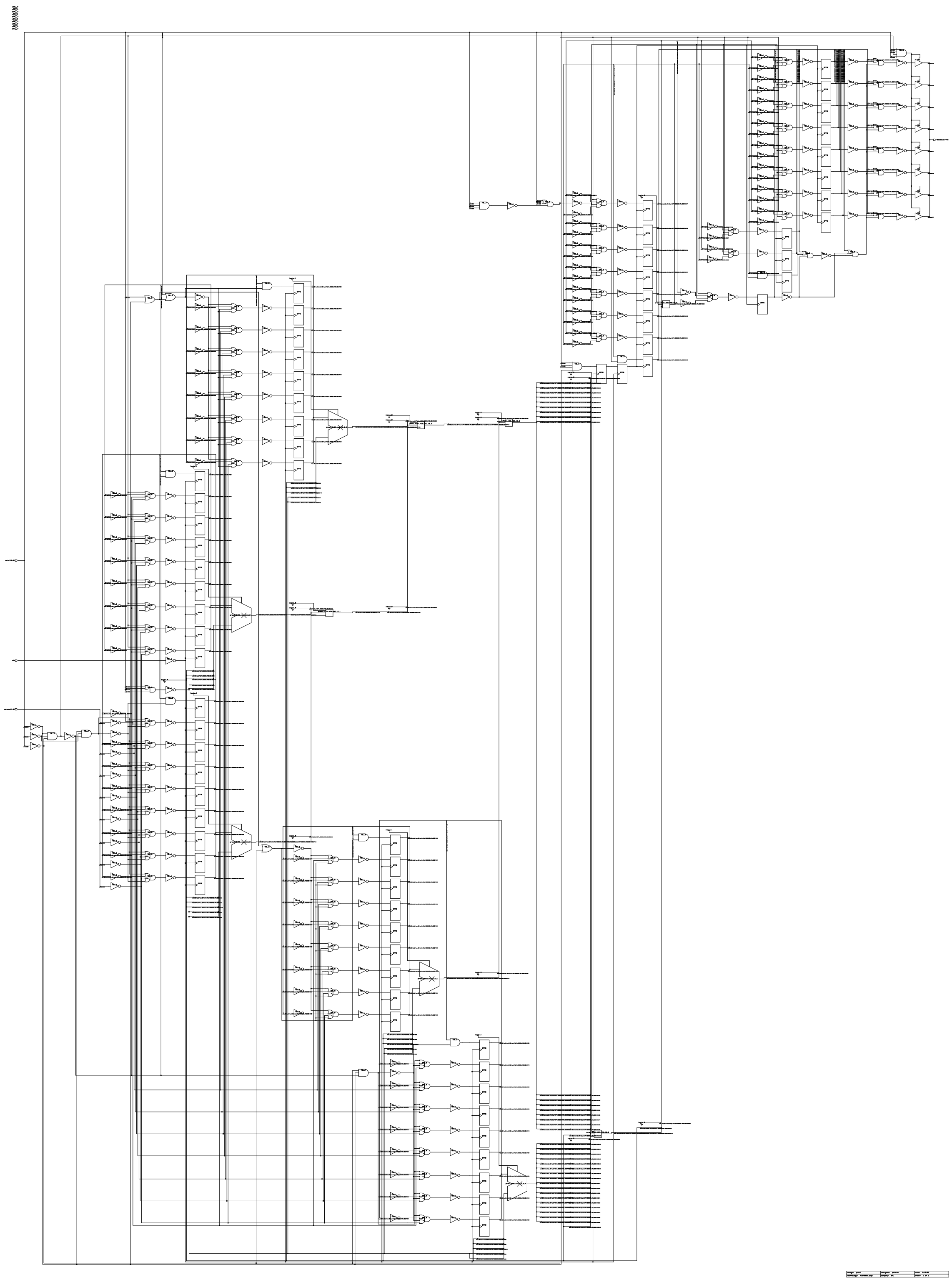


Figure A.7: Synthesized Schematic for 5x5 Mean Filter Processor 1

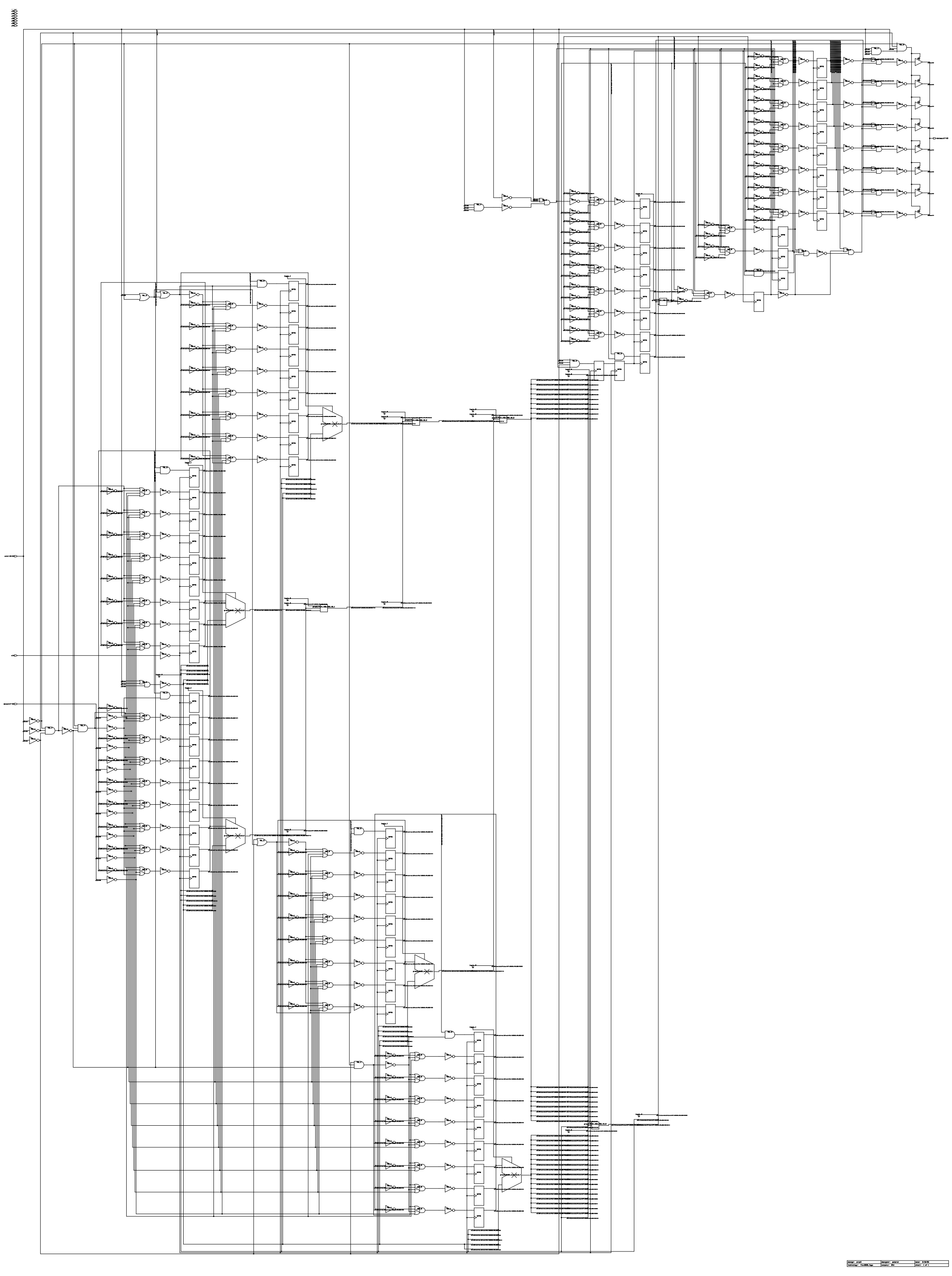


Figure A.8: Synthesized Schematic for 5x5 Mean Filter Processor 2

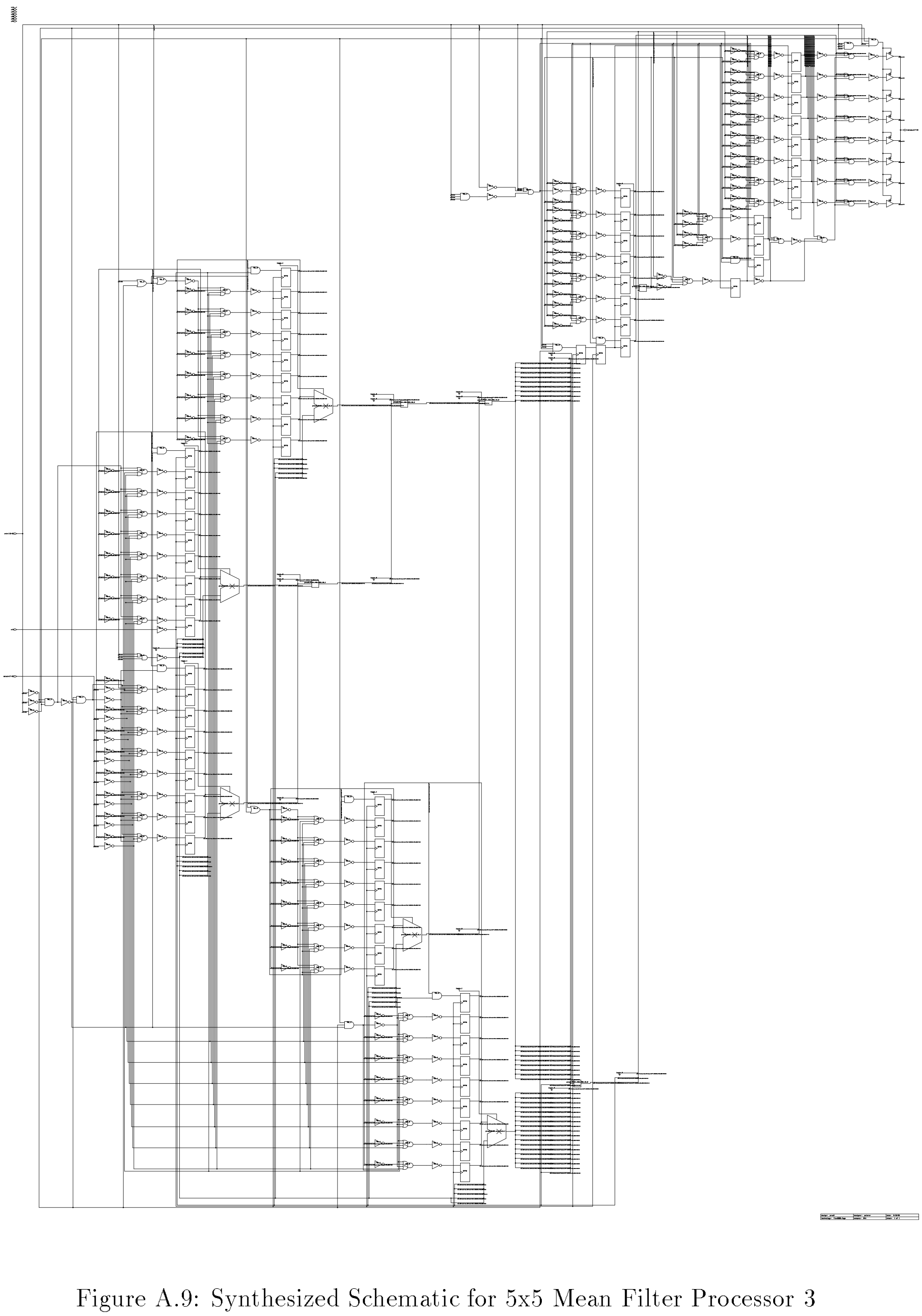


Figure A.9: Synthesized Schematic for 5x5 Mean Filter Processor 3

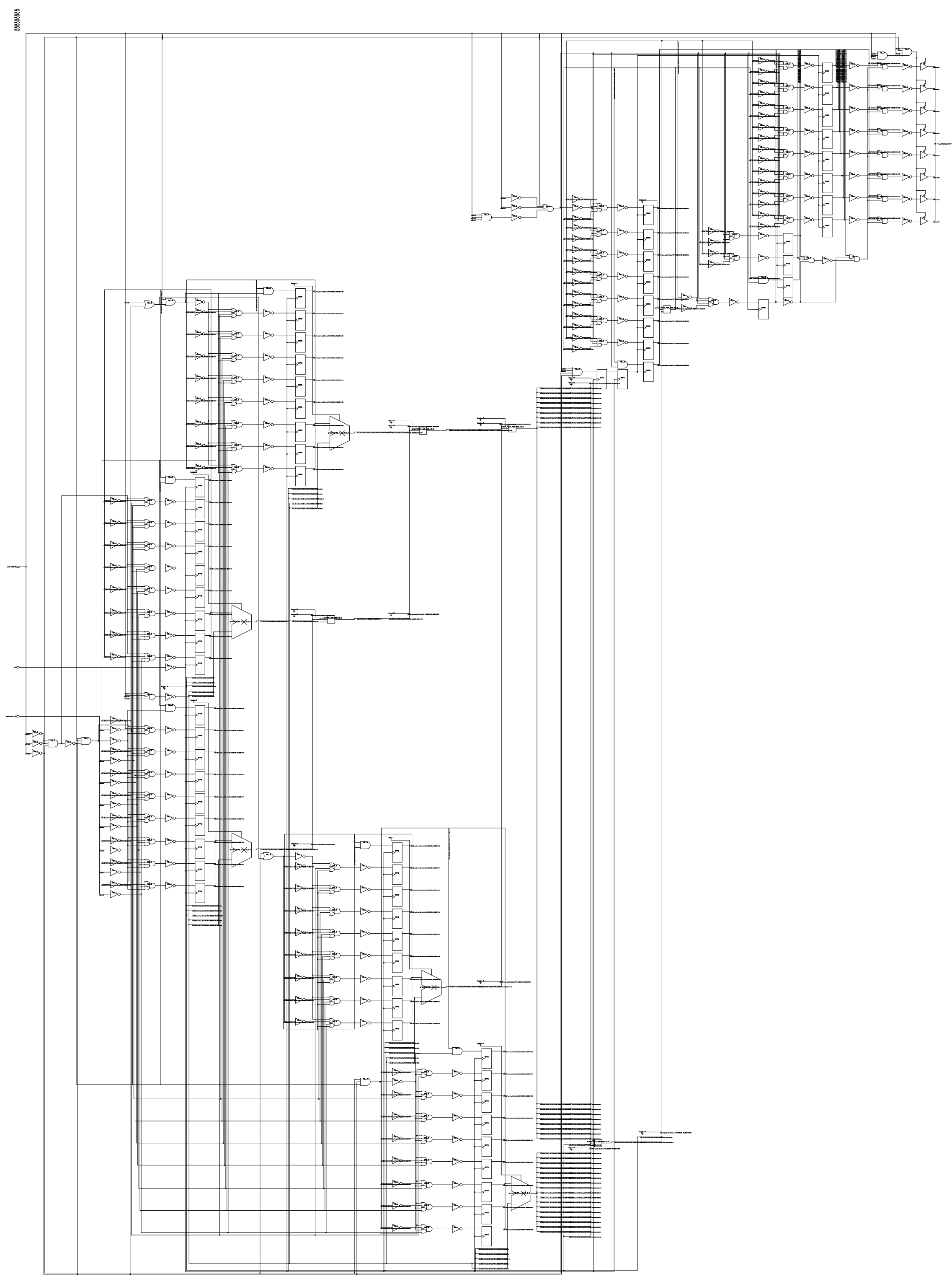


Figure A.10: Synthesized Schematic for 5x5 Mean Filter Processor 4

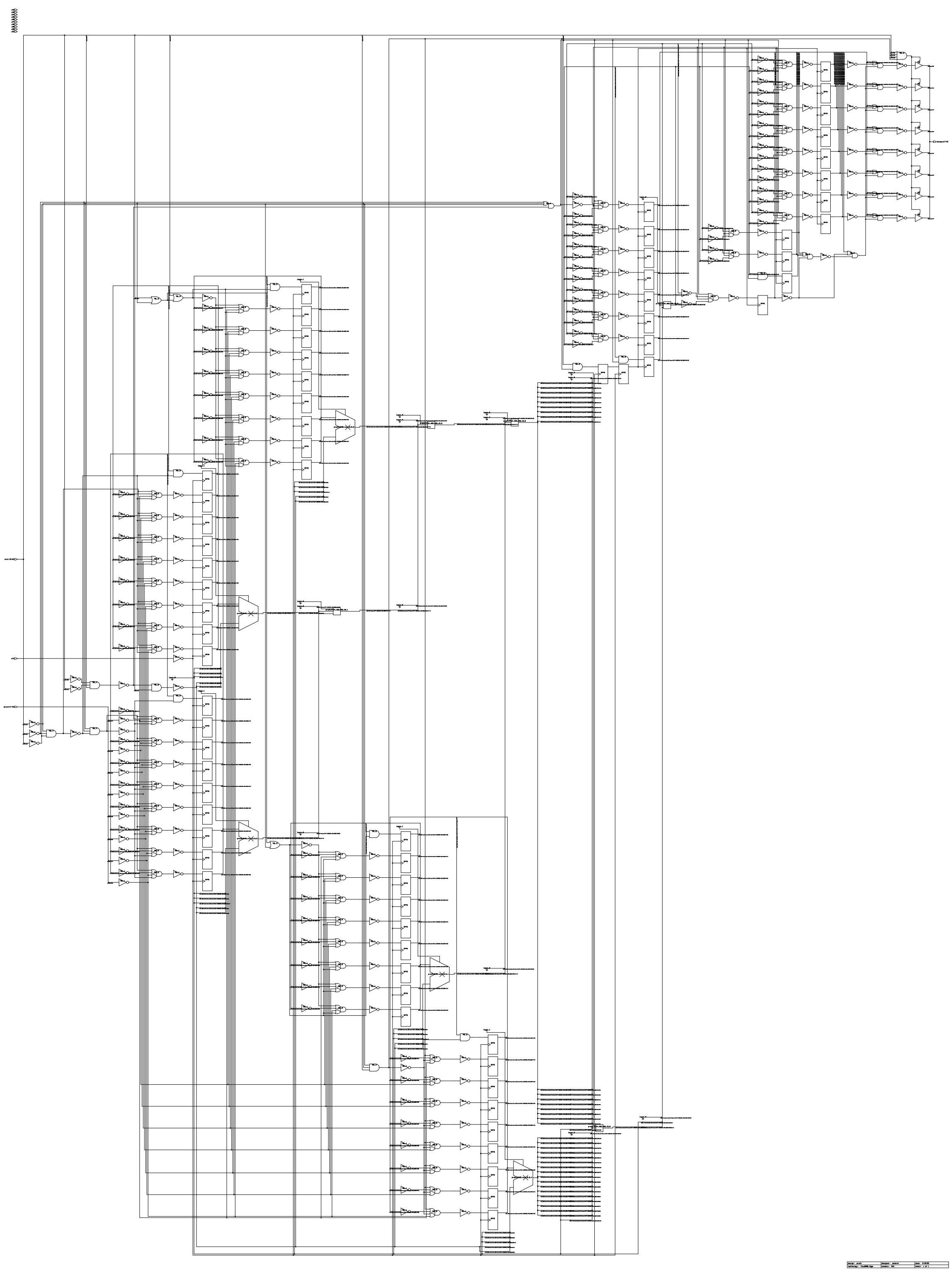


Figure A.11: Synthesized Schematic for 5x5 Mean Filter Processor 5

APPENDIX B

DESIGN METHODOLOGY

The designs in this thesis were created solely for the purpose of illustrating the feasibility of the use of FPGAs as reconfigurable hardware for digital signal processing. They are not intended to represent absolute *best case* performance. Many were designed with the criteria that the FPGA place and route software handle most if not all of the design placement and routing. Hence, placement and routing on the FPGAs may often be far less than optimal. In addition, the VHDL-based designs represent attempts at rapid development of DSP hardware, a feature desirable for reconfigurable hardware. Many of the designs could be optimized or implemented differently to obtain increased performance but at the cost of increased development time. Also, no attempt has been made to adjust the results for the various FPGAs to account for differences in process technology. This would be relevant to an FPGA comparison since the technology used to implement the FPGA affects its overall performance. Finally, several of the designs (including multipliers) were implemented on only one or two of the three FPGAs discussed in this thesis. This, however, does not necessarily indicate the inability of the non-used FPGAs to implement the design efficiently. The primary criteria used in choosing whether a design would be implemented on a particular FPGA was the required design time and not the ability of the FPGA to efficiently implement the design.

Thus, the design results in this thesis should only be used to indicate the possibility of using FPGAs for reconfigurable digital signal processing hardware. No attempt should be made to use the results as a comprehensive comparison as to which FPGA is the best for digital signal processing. Only general trends and strengths of the various FPGAs have been indicated and all results should be taken in that context.

APPENDIX C

LIST OF IMPLEMENTED DESIGNS

The following lists all of the designs referenced in tables throughout the thesis according to which of the designs were actually implemented by the author, which of the designs were obtained from previous work or external sources, and which of the designs were designed only on paper.

C.1 Implemented designs:

- All of the multiplication designs listed in Chapter 3.
- The Xilinx 4010-based 20-tap FIR filters using constant multipliers listed in Table 4.1.
- The 8-bit,64-pt. Radix-4 FFT implemented on the Altera FPGA listed in Table 4.2.
- The 2-D convolution processors using Altera FPGAs listed in Tables 4.3 and 4.4.

C.2 Designs obtained from previous work or external sources:

- Data for the custom multiplier chips listed in Table 3.3.
- All data for the TI TMS320C5X DSP processor.
- The CLAy31-based 20-tap FIR filters listed in Table 4.1.
- The PDSP1610A and Sharp FFT designs in Table 4.2.
- The PDSP16488, LF48908, HSP48908, and TMS320C80-based 2-D convolution processor designs in Table 4.4.

C.3 Designs implemented purely on paper:

- The Altera, ASIC, LD LMU08, and LD LMU18-based 20-tap FIR filters in Table 4.1.
- The PDSP16116/A-based FFT design listed in Table 4.2.
- The Xilinx 4010-based 2-D convolution processors in Table 4.4

BIBLIOGRAPHY

[1] Altera Corporation, San Jose, CA. *FLEX 8000 Handbook*, 1994.

[2] Rick Amerson, Richard J. Carter, W. Bruce Culbertson, Phil Kuekes, and Greg Snider. Teramac - configurable custom computing. In *To appear in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines 95*, Napa, CA, April 1995.

[3] Raymond J. Andraka. FIR filter fits in an FPGA using a bit-serial approach. In *Third Annual PLD Design Conference and Exhibit*, 1993.

[4] J. M. Arnold. The Splash 2 software environment. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 88–93, Napa, CA, April 1993.

[5] P. M. Athanas and A. L. Abbott. Real-time image processing on a custom computing platform. *IEEE Computer*, 28(2):16–24, February 1995.

[6] Marcelo Alves De Barros and Mohamed Akil. Study and implementation of a real time 3x3 programmable convolver with reconfigurable technology. In *Proceedings of Euro ASIC '92*, pages 392–395. IEEE Computer Society Press, June 1992.

[7] Magdy A. Bayoumi. VLSI architectures for DSP applications: Current trends. In *35th Midwest Symposium on Circuits and Systems*, volume 1, pages 150–153, 1992.

[8] N. W. Bergmann and J. C. Mudge. Comparing the performance of FPGA-based custom computers with general-purpose computers for DSP applications. In D. A. Buell and K. L. Pocek, editors, *Proceedings of IEEE Workshop on FPGAs for Custom Computing Machines*, pages 164–171, Napa, CA, April 1994.

[9] Dave Bursky. Improved DSP ICs eye new horizons. *Electronic Design*, pages 69–82, November 1993.

[10] S.C. Chan, H.O. Ngai, and K.L. HO. A programmable image processing system using FPGAs. *International Journal of Electronics*, 75(4):725–730, October 1993.

[11] Kenneth David Chapman. Fast integer multipliers fit in FPGA's. *EDN*, page 80, May 12 1994.

[12] Dev C. Chen and Jan M. Rabaey. A reconfigurable multiprocessor IC for rapid prototyping of algorithmic-specific high-speed DSP data paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, December 1992.

[13] Chi-Jui Chou, Mohanakrishnan, and Joseph B. Evans. Fpga implementation of digital filters. In *Proceedings of the Fourth International Conference on Signal Processing Applications and Technology*, pages 80–88, 1994.

[14] Joseph B. Evans. An efficient FIR filter architecture. *IEEE International Symposium on Circuits and Systems*, 1(3):627–630, May 1993.

[15] O. Fatemi, F. Idris, and S. Panchanathan. FPGA implementation of the LRU algorithm for video compression. *IEEE Transactions on Consumer Electronics*, 40(3):337–343, August 1994.

[16] Catherine H. Gebotys and Robert J. Gebotys. Application-specific architectures for field-programmable VLSI technologies. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pages 124–130, 1994.

[17] R. Gnanasekaran. On a bit-serial input and bit-serial output multiplier. *IEEE Transactions on Computers*, 32(9):878–880, September 1983.

[18] Ben Gold and Theodore Bially. Parallelism in fast Fourier transform hardware. *IEEE Transactions on Audio and Electroacoustics*, 21(1):5–16, February 1973.

[19] Semiconductor Group. *Digital Signal Processing Products and Applications Primer*. Texas Instruments, 1991.

[20] Karl M. Guttag. Multimedia powerhouse. *BYTE*, pages 57–64, June 1994.

[21] Thomas M. Hopkinson and G. Michael Butler. A pipelined, high-precision FFT architecture. In *35th Midwest Symposium on Circuits and Systems*, volume 2, pages 835–838, 1992.

[22] Kai Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. John Wiley & Sons, 1979.

[23] Frederick R. Hood III. Reconfigurable hardware for digital signal processing algorithms. Master's thesis, University of Washington, Seattle, Seattle, Washington, August 1993.

[24] Steven K. Knapp. Accelerate FPGA macros with one-hot approach. *Electronic Design*, pages 71–78, September 13 1990.

[25] Israel Koren. *Computer Arithmetic Algorithms*. Prentice Hall, 1993.

[26] Edward A. Lee. Programmable DSP architectures, part 1. *IEEE ASSP Magazine*, pages 4–19, October 1988.

[27] Edward A. Lee. Programmable DSP architectures, part 2. *IEEE ASSP Magazine*, pages 4–13, January 1989.

[28] Woobin Lee, Robert J. Gove, Chris J. Read, and Youngmin Kim. Mediastation 5000: An integrated multimedia video and audio processing system. Submitted to IEEE Multimedia Magazine for Jan. 94, 1994.

[29] R.F. Lyon. Two's complement pipeline multipliers. *IEEE Transactions On Communications*, pages 418–424, April 1976.

[30] Gin-Kou Ma and Fred J. Taylor. Multiplier policies for digital signal processing. *IEEE ASSP Magazine*, pages 6–19, January 1990.

[31] National Semiconductor. *Configurable Logic Array (CLAy) Data Sheet*, December 1993.

[32] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1989.

[33] Ian Page. Constructing hardware-software systems from a single description. *Written for an upcoming special issue of VLSI signal processing*, July 1995.

[34] D. Ross, O. Vellacott, and M. Turner. An FPGA-based hardware accelerator for image processing. In W. Moore and W. Luk, editors, *More FPGAs: Proceedings of the 1993 International workshop on field-programmable logic and applications*, pages 299–306, Oxford, England, September 1993.

[35] Nabeel Shirazi, Al Walters, and Peter Athanas. Quantitative analysis of floating point arithmetic on FPGA based custom computing machines. In *To appear in Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines 95*, Napa, CA, April 1995.

[36] Mohamed Wahab and Darren Puckey. Fpga-based dsp systems. In *More FPGAs*, pages 291–298, 1993.

[37] Xilinx, San Jose, CA. *The Programmable Logic Data Book*, 1994.