Field Programmable Gate Arrays for Radar Front-End Digital Signal Processing

by

Tyler J. Moeller

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and
Computer Science

and

Master of Engineering in Electrical Engineering and
Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 22, 1999

Author ....
Department of Electrical Engineering and Computer Science
May 22, 1999

Supervised by ....
David R. Martinez, Thesis Supervisor
Group Leader, M.I.T. Lincoln Laboratory

Certified by ....
Saman P. Amarasinghe, Thesis Supervisor
Assistant Professor, M.I.T. Laboratory for Computer Science

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Field Programmable Gate Arrays for Radar Front-End Digital Signal Processing

by

Tyler J. Moeller

Submitted to the Department of
Electrical Engineering and Computer Science

May 22, 1999

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

As field programmable gate array (FPGA) technology has steadily improved, FPGAs have become viable alternatives to other technology implementations for high-speed classes of digital signal processing (DSP) applications. In particular, radar front-end signal processing, an application formerly dominated by custom very large scale integration (VLSI) chips, may now be a prime candidate for migration to FPGA technology. As this thesis demonstrates, current FPGA devices have the power and capacity to implement a FIR filter with the performance and specifications of an existing, in-system, front-end signal processing custom VLSI chip. A 512-tap, 18-bit FIR filter was built that could achieve sample rates of 7 MHz (with a clock rate of 117 MHz) using Xilinx Virtex FPGA technology, and was demonstrated through simulation and hardware implementation. Distributed arithmetic, bit-level systolic arrays, parallel multiplier/accumulator (MAC) cells, fast FIR algorithms, and frequency domain filtering were investigated to determine the most optimal structure for a FPGA FIR design, with distributed arithmetic resulting in the best performance. A custom VHDL cell-based layout tool was designed to improve the placement strategies of the Xilinx FPGA place and route tools, and improved the speed performance of the distributed arithmetic design by 37%.

Thesis Supervisors:

David R. Martinez
Group Leader, Digital Radar Technology Group, MIT Lincoln Laboratory

Saman P. Amarasinghe
Assistant Professor, MIT Laboratory for Computer Science

Acknowledgments

I'd like to thank...

Dave Martinez for being interested in and for giving me a reconfigurable-computing project for my thesis, for supporting, encouraging, and funding my work, for encouraging me to submit a paper [MM99] to and participate in the 1999 Field-Programmable Custom Computing Machines Symposium, and for allowing me to co-author a paper for the *Journal of VLSI Signal Processing Systems* [MMT00]. My thesis work has been in an area that I find very interesting and think has a lot of promise for the future.

Saman Amarasinghe for being my thesis advisor at MIT. Your research interest in reconfigurable computing helped better my understanding of the subject and how it might apply to signal processing. I really appreciate your support, advice, and reading of my thesis in a short period of time.

Bill Song for being my mentor for three years, for trusting me and giving me a lot of freedom within unbelievable, mission-critical projects unusual for an intern to work on, for helping me with school and career decisions, and for teaching me a lot about working with cutting-edge technologies.

Mike Killoran for continuously and selflessly helping me with programming issues and design ideas, giving me moral support when I was stressed out, and being someone I could always turn to at Lincoln to bounce something off of or joke around with. Your FIR and other utility programs were a huge help.

Bob Ford at Lincoln and Brent Nelson at BYU for giving me the original idea for this thesis and providing support along the way; Ed McGettigan at Xilinx for providing me optimized Virtex multipliers and tirelessly giving me Virtex support directly from the applications group at Xilinx; Huy Nguyen for helping me with my MATLAB simulations and fixed-point noise analyses; Harry Levinson for endlessly helping me with my computer, and for installing all the software and hardware I needed.

Edith Gardner, Bob D'Ambra, and Jack Selfridge for putting up with me for three years and making my internship a lot of fun.

All my friends who made the past five years the best of my life. I don't know how I found you all, but thanks, and I will never forget you. We'll keep in touch.

My parents, who made MIT possible for me and never stopped supporting me or losing faith in me through thick and thin.

And Maggie. You've been there when I've been stressed or worried about this thesis, school, or anything. Thanks for always making me happy and being the one person I can *always* turn to. You've made the last year of my life incredible.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Thesis Overview

As field programmable gate array (FPGA) technology has steadily improved, reconfigurable computing using FPGAs has become a viable alternative to other technology implementations, including custom very large scale integration (VLSI) devices and processor-based systems, for high-speed classes of digital signal processing (DSP). In particular, radar front-end signal processing, an application formerly dominated by custom VLSI chips, may now be a prime candidate for migration to FPGA technology.

To demonstrate the feasibility of using reconfigurable computing to implement radar front-end signal processing, FPGA-based DSP solutions meeting the specifications of an existing, in-system, radar front-end custom VLSI chip were investigated. These specifications required a 512-tap real finite impulse response (FIR) filter be built that could operate on 16-bit data and 18-bit coefficients while outputting results with 18-bits of precision with a sample rate of 5 MHz and a clock rate of 40 MHz. Two banks of coefficients were required so that one could be active while another was loaded in the background. The designs were implemented in the largest FPGA available today, the Xilinx Virtex XCV1000.

The designs considered included distributed arithmetic, bit-level systolic arrays, parallel multiplier/accumulator (MAC) cells, fast FIR algorithms, and frequency domain filtering. Implementations using distributed arithmetic, parallel MAC cells, and fast FIR filtering were built and simulated. All of these designs met the custom VLSI chip's design specifications and exceeded its performance, with a distributed arithmetic design using linear-systolic cells having the best performance (7 MHz sample rate). An eight-tap distributed arithmetic design was implemented on an Annapolis Micro Systems Starfire reconfigurable-computing engine.

To improve upon a poor linear-systolic design placement strategy by the synthesis and FPGA tools, a custom VHDL placement tool, CELL_MAKER, was built that could read VHDL, extract user placement constraints, and construct a placement strategy for the final FPGA. This tool improved the performance of the linear design by 37% due to layout alone.

This thesis demonstrates that current FPGA devices have the power and capacity to implement a FIR filter with the performance and specifications of an existing, in-system, front-end signal processing custom VLSI chip.

1.2 Contributions

This thesis contributes new research or data in several areas:

1. A unique parallel MAC design using shift registers to store the inputs when multiple taps per MAC are computed instead of using RAM to store the multiplier outputs was constructed, simulated, and area and performance results were determined.

2. A distributed arithmetic design was built that did not require constant coefficients. Instead, it could operate on two banks of coefficients (one active, one loadable), and was built, simulated, and implemented in hardware. Two variations of this design, one using a tree approach for summing the individid-

ual cell’s outputs together, and the other using a linear-systolic approach were built, and area and performance results were determined.

3. A new approach to reducing a parallel MAC design’s area using fast FIR algorithms was developed. A filter using this approach was constructed, simulated, and area and performance results were determined.

4. Area figures for a pipelined radix-2 FFT-based filter in the frequency domain were calculated.

5. The performance limitations of the linear-systolic distributed arithmetic design (which should have been the fastest design) were determined to be due to the placement strategy of the design, as it was not placed in a linear-systolic fashion. A custom tool was built to solve this problem that could read user-inserted constraints in the design’s VHDL code and generate a file with placement constraints for the Xilinx place and route tools to use. This tool improved the design’s performance by 37%.

6. The power and performance in terms of billions of operations per second (GOPS) were calculated for the linear-systolic distributed arithmetic design.

1.3 Outline of Thesis

Chapter 2 discusses the background for the research in this thesis, including reconfigurable computing for DSP using FPGAs, trends in FPGA development, and radar front-end signal processing. Chapter 3 gives an overview of a custom VLSI design built and used at MIT Lincoln Laboratory for a particular radar front-end signal processing system, including a description of its features, architecture, and an analysis of its output precision due to internal round-off noise. Chapter 4 discusses the problem this thesis investigated, which was building a FPGA implementation to meet the specifications of the custom VLSI chip, and why a FPGA implementation would be beneficial. Chapter 5 gives an overview of the FPGA used in this thesis, the Xilinx Virtex. Chapter 6 provides a background on each of the algorithms or design techniques investigated for implementing a FPGA-based FIR filter, and Chapter 7 presents the particulars and implementations of each algorithm or technique as applied to the design problem discussed in Chapter 4 within a Virtex FPGA. Chapter 8 introduces the custom placement tool, its motivation, operation, and results as applied to the linear-systolic distributed arithmetic design. Chapter 9 summarizes the area and performance results for each implementation built in a Virtex FPGA, and calculates the power and GOPS for the best implementation (linear-systolic distributed arithmetic after using the custom placement tool). Chapter 10 discusses how an eight-tap version of the distributed arithmetic design was implemented in real hardware, and Chapter 11 presents the conclusions of this thesis.

Chapter 2

Background

2.1 Radar Digital Signal Processing Development

Until recently, the first few stages of radar signal processing after an incoming radar signal was acquired were performed in the analog domain. As analog to digital converters (ADCs) have become faster in recent years, and digital hardware has become more capable, the trend has been to move the analog to digital converter closer to the radar antenna in the signal processing chain and perform more processing in the digital domain. Digital hardware offers more robust system stability, more flexibility in waveform and filter design, the ability to develop adaptive processing algorithms such as beamforming, adaptive nulling, or space-time adaptive processing that require fast changes in hardware configurations and system coefficients, and an easier upgrade path as digital electronics continue to advance. [MMT00]

As more and more front-end radar signal processing functions are moved into the digital domain, the signal processing requirements for the digital hardware executing them increases dramatically. This is due to fast ADC sampling rates, large numbers of sensor channels, and stringent requirements on filter designs. The computational demands range from tens to hundreds of billion operations per second (GOPS), as data throughputs often range in the hundreds of megabytes per second (MBytes/sec). Further restrictions on the digital hardware include size, weight, power, environment, and shock constraints as digital radar systems are fielded in platforms ranging from early warning radars, unmanned air vehicles, fighters, and space-borne surveillance and targeting radars. [MMT00]

For the past several years, commercially available digital signal processors (DSPs) or reduced instruction set (RISC) microprocessors have not been able to meet the system requirements for front-end radar signal processing. However, this processing is often very regular, is highly parallel, and is usually independent of the data. Therefore, most implementations to date have used either commercially available, dedicated, computing engines or custom very large scale integrated-circuit (VLSI) designs. [MMT00, Hau98, BAK96]

One such custom VLSI design was recently fielded at MIT Lincoln Laboratory that is capable of operating at 100 GOPS. It has the ability to change banks of coefficients in a few milliseconds, and is channel parallel, meaning it can be scaled to many hundreds of GOPS as the number of radar input channels is increased. [MMT00, Gre96]

2.2 Reconfigurable Computing for Digital Signal Processing

In the late 1980s and early 1990s, the conventional wisdom where hardware was fixed at design time and software contained the flexibility in a system was reexamined. The presence of chips (such as field programmable gate arrays, or FPGAs) that could adapt to the current demands of an application lead to the adoption of “generic” hardware, in which FPGAs, microcontrollers, and other reprogrammable parts could be combined together on a single board that could be reconfigured to serve many different applications. A computing device would not need to be as generic as a microprocessor, as reconfigurable elements could form special-purpose hardware to solve a specific problem, but would be flexible enough to change that special-purpose function to another function upon demand. [Hau98, BAK96]

The ability of a system to change its functionality in hardware instead of in just software lead to the development of reconfigurable computing. Hardware was now able to become the medium for general-purpose computing, as it could adapt itself to compute algorithms normally handled by a processor. As the requirements for the algorithms changed, so could the hardware. Because the algorithms were being executed directly by hardware instead of by a computer that was fetching and decoding instructions sequentially, they would gain the performance boost inherent in being executed, in parallel, by processing units precisely designed for the algorithm. [Hau98, BAK96]

Currently, reconfigurable computing is a niche research field—it is not applicable for all applications. However, for applications characterized by deeply pipelined, highly parallel, and integer arithmetic processing, reconfigurable computing machines have shown performance improvements of an order of magnitude or more at a low cost. This is because current reconfigurable computing mediums are usually characterized by arrays of highly-replicated, pipelined, functional blocks. An algorithm that can be broken into many parallel tasks will map well into this architecture, especially if it is easily pipelined. A more complicated, irregular, structure will not map well into this architecture, as the number of unique functional units required to implement an irregular algorithm may surpass the capacity of even a large sized configurable computer. [Hau98, VH98]

A microprocessor achieves a variety of different functions temporally by executing multiple functions sequentially, during different cycles, while reconfigurable computing solutions achieve a variety of functions spatially by having different logic elements compute different functions. Therefore, microprocessors will execute irregular computations and complex data-flow manipulations better, while reconfigurable computing machines can be superior for data-parallel applications, where large amounts of data must be acted on in a similar manner. Some examples of tasks that are suitable for configurable computing include: image processing, pattern matching, target recognition, cryptography, filtering, convolution, FFTs, and some database tasks [Hau98, VH98].

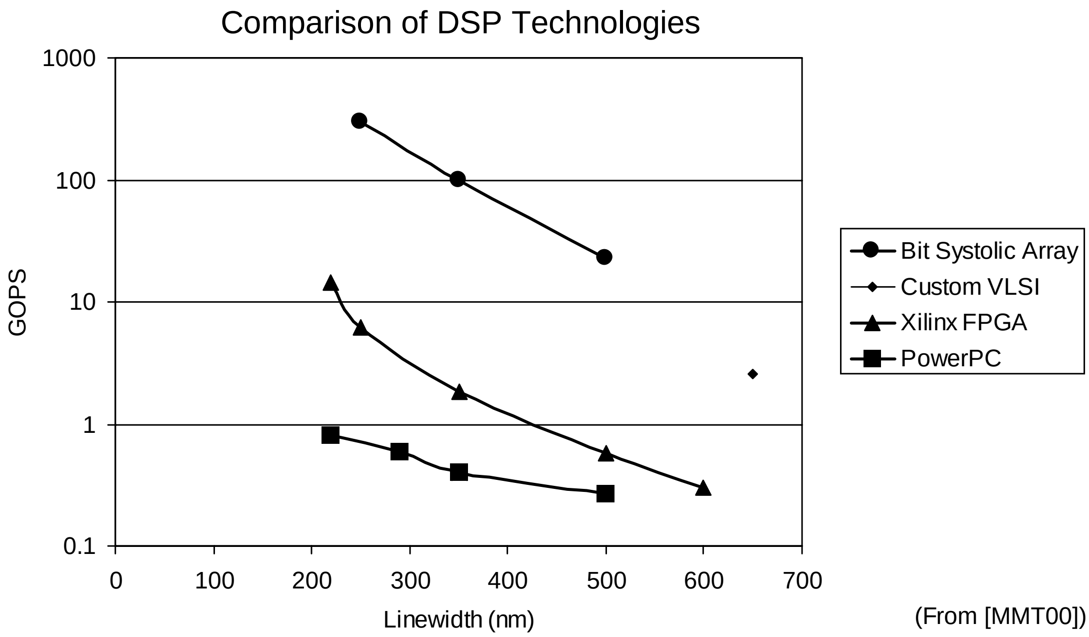
FPGAs are a good medium for custom computing applications because of their highly-replicated regular structure of configurable logic combined with many pipeline registers that can be easily programmed to perform a series of parallel computing tasks. As FPGAs have grown in capacity, improved in performance, and decreased in cost, FPGA based custom computing machines have become an ideal medium for DSP applications. Several studies performed by Xilinx, researchers at BYU, Intel, and other institutions have shown that a single FPGA can outperform a DSP chip by an order of magnitude or more for pipelined, parallel DSP applications [ASR98, PH95, Gos96a, Xila, Kna95, VH98, Con96].

As FPGA technology has improved over the past several years, it may now be feasible to field a reconfigurable computing solution to the radar front-end signal processing problem instead of a custom VLSI solution. Such a solution would be more cost effective (typically FPGAs have a much lower cost per-part than custom chips due to non-recurring engineering costs and time-to-market factors for production runs of 100,000 to 400,000 units [Liu95]) and will require less time and fewer resources for design, development, testing, and fielding than custom VLSI. In addition, because of the high demand and general purpose nature of FPGA devices, FPGA manufactures have the ability to reap the benefits of the latest reductions in lithography feature size, enabling FPGAs to take advantage of technologies that may not be available for custom VLSI for a significant period of time. Finally, the most significant benefit of using reconfigurable computing would be the ability of a system to adapt its configuration dynamically, in-system, a benefit previously only available to software running on slow microprocessor-based designs. [MM99]

There is a range of signal processing requirements (typically data throughputs exceeding gigabytes per second) for especially demanding applications which exceeds the maximum performance FPGAs can deliver, and for which custom VLSI devices are still the only viable solution. However, for many systems, FPGAs are becoming powerful enough to meet the system's signal processing demands and may be a better choice than a custom VLSI solution given the benefits of reconfigurable computing described above.

2.3 FPGA Trends

Shrinking process geometry and reduced supply voltages in FPGAs have resulted in enormous growth in terms of their computational capability and power efficiency [MMT00]. The growth in FPGA computational capability versus process geometry is shown in Figure 2.1. Curves are shown for FPGAs, the Motorola Power PC, and custom VLSI for comparison purposes. The data for the custom VLSI is based on several development efforts at MIT Lincoln Laboratory over the past several years [MMT00].

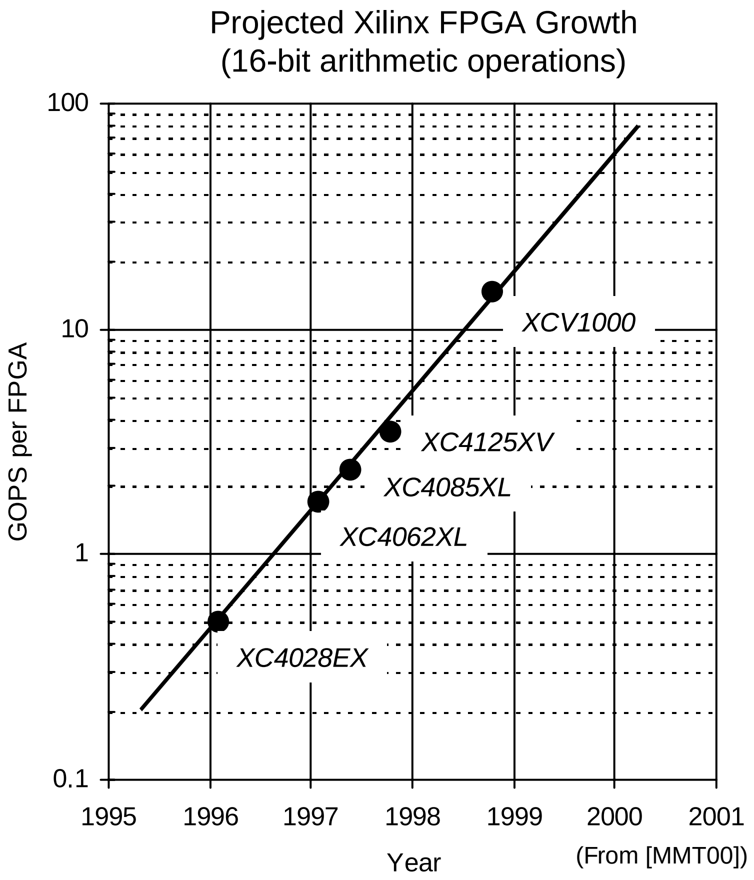


(From [MMT00])

Figure 2.1: Growth in FPGA Performance Versus Process Geometry

Figure 2.1 shows that FPGAs offer an intermediate capability between that offered by programmable processors and custom VLSI. Presently, FPGAs are about an order of magnitude more capable than programmable processors, and an order of magnitude less capable than full custom VLSI. Furthermore, the gap between FPGAs and microprocessors is widening, as FPGA vendors use increased density to increase the number of logic cells, whereas microprocessor developers often use increased density for caches and reducing die size. [MMT00]

The growth in the computational capability of FPGAs for DSP applications as a function of time is shown in Figure 2.2 in billions of 16-bit arithmetic operations per second. Within the last 5 years, the computational capability of FPGAs have increased by an order of magnitude every two years. It is now feasible to explore the implementation of complex DSP systems using FPGAs as computational building blocks.



(From [MMT00])

Figure 2.2: Growth of Computational Capability for Xilinx FPGAs

2.4 Radar Front-End Digital Signal Processing

A typical radar signal processing flow is illustrated in Figure 2.3. In this flow, the first stage of digital filtering is what is referred to in this thesis as front-end radar signal processing. This stage of filtering converts the data from real ADC samples to complex digital in-phase and quadrature (DIQ) samples, and is referred to as DIQ sampling [Sti98]. DIQ sampling is important to preserve the target's Doppler information. Although the remaining processing stages, Doppler filtering, adaptive nulling, and post-nulling processing, can be very demanding, it is the first stage of digital filtering that has the most stringent processing requirements that have necessitated custom VLSI designs until now. The rest of the signal processing flow is described extensively in [War94].

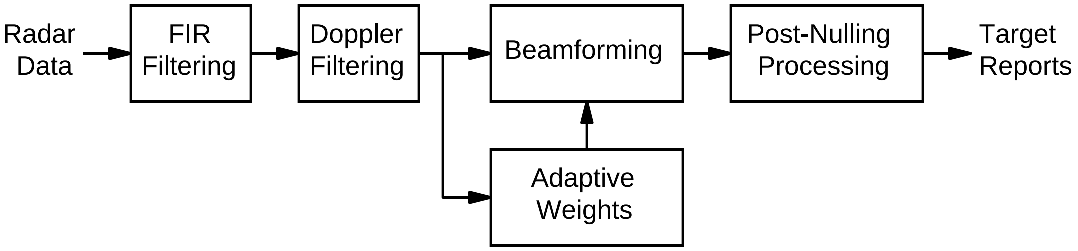


Figure 2.3: Typical Radar Signal Processing Flow

The DIQ filtering is designed to extract one sideband after ADC sampling, map the sideband to baseband, and remove any remaining spectrum images and DC offsets. The exact DIQ filter coefficients depend on the characteristics of the bandwidth present in the transmitted pulse, and are required to be dynamically alterable. The DIQ architecture for the custom VLSI design is shown in Figure 2.4, and is described in detail in [MMT00, Sti98, War94]. As the figure shows, the DIQ architecture requires two finite impulse response (FIR) filters followed by decimation. This design required a filter with 208 complex taps.

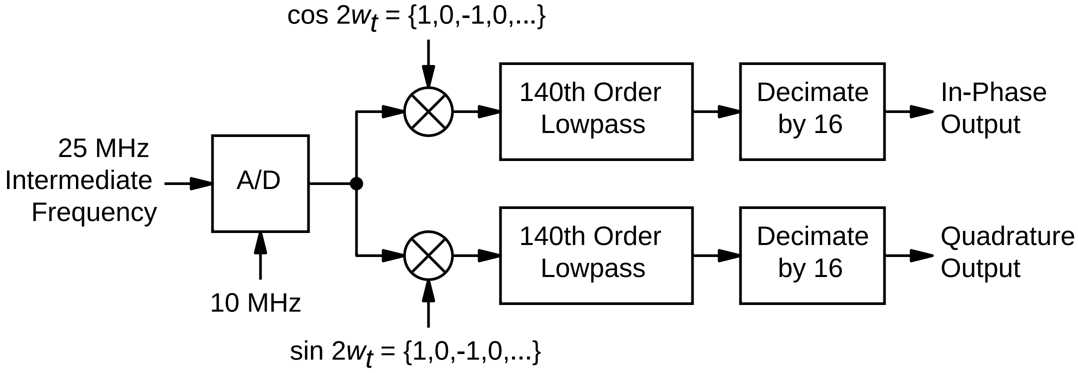


Figure 2.4: Digital In-phase and Quadrature Sampling Architecture

Chapter 3

Custom VLSI Implementation

3.1 Overview

A custom VLSI implementation of the DIQ architecture in Figure 2.4 was begun in 1996 and finished in 1998 at MIT Lincoln Laboratory. It was concluded that no commercially available DSP or RISC processors or dedicated filtering chips would meet the processing requirements of the DIQ filtering stage, leaving a custom solution as the only alternative. The design that was chosen was a custom chip made up of a mix of standard cells and datapath multiply-accumulate (MAC) cells [Gre96]. The overall system consisted of several boards using these custom chips, and is described in detail in [MMT00].

The custom VLSI chip, designed using 1996 process technology, is shown in Figure 3.1. The standard cells are used in the chip control interface, barrel bit selector, and downsampler. The datapath blocks form the MAC cells. There are a total of 64 MACs available on the chip. Depending on the input sample rate and mode, these MACs can be used either as 64, 128, or 256 complex taps, or as 128, 256, or 512 real taps. Since the chip was to compute the in-phase (I) and quadrature (Q) data from real ADC data, 32 MACs processed up to 128 real taps for the I samples and, in parallel, another 32 MACs up to 128 taps for the Q samples. [Gre96, MMT00]

For the specific technology demonstration discussed in Section 2.4, ADC samples arrive at 10 MHz. Samples alternate between the I computation and the Q computation. Thus, two sets of outputs are computed at 10 MHz (5 MHz

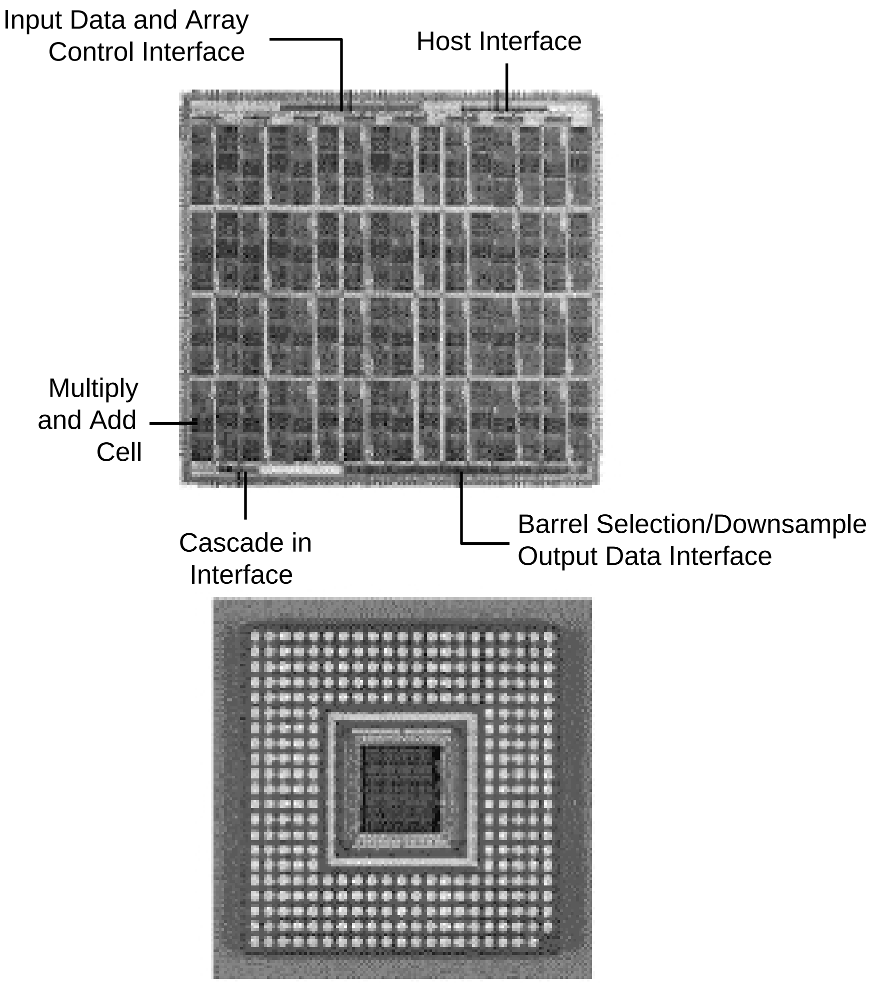

Input Data and Array Control Interface
Host Interface
Multiply and Add Cell
Cascade in Interface
Barrel Selection/Downsample Output Data Interface

Figure 3.1: Custom VLSI Chip

for I and Q), each using 104 real taps. Since there are two real operations per tap (a multiply and an add), the total computation throughput for these parameters is 2.08 GOPS. These operations are performed on 18-bit sign extended data (from 14 bit ADC samples) using 18-bit coefficients. The resulting outputs (I and Q samples) are selected from the most significant 24 bits. [MMT00, Gre96]

The key features of this chip are: [MMT00, Gre96]

- 2.08 GOPS in DIQ mode
- 5.02 GOPS in 512-tap real mode

- 40 MHz operating frequency (tested to 44 MHz)
- 18-bit input data and coefficients; 24-bit output data
- 585 mil x 585 mil die size
- 1.5 million transistors
- 0.65 mm feature size
- CMOS using three-layer metal
- Designed for 4 watts power dissipation; measured 3.2 watts in operation
- DIQ mode throughput/power = 0.65 GOPS/W
- 512-tap real mode throughput/power = 1.57 GOPS/W

3.2 Custom VLSI Architecture

A block diagram of the custom VLSI chip's architecture is shown in Figure 3.2. As was discussed in Section 3.1, the custom VLSI chip consists of 64 MAC units. Each MAC unit contains a multiplier, accumulator and intermediate storage memory, and two banks of coefficient memory. The two banks of coefficient memory allows one set of coefficients to be active and is used by the multipliers while a new set can be loaded into the other coefficient bank. Once the new set has been loaded, it can now become active, allowing the chip to instantly change from one set of coefficients to another.

Each MAC, by using the accumulator and intermediate storage memory, is capable of forming the products of the current chip input and up to eight filter taps (i.e. eight coefficients). These products are accumulated together as required within the MAC, and then added to the other MACs' results when a new input is present. If the MACs are operating in their eight tap mode, they must run at a clock rate eight times the input sample rate so that all eight taps' products are computed each time a new input arrives. In this mode, the 64 MACs can compute a 512-tap real filter, the mode used as a benchmark for a FPGA implementation (Chapter 4).

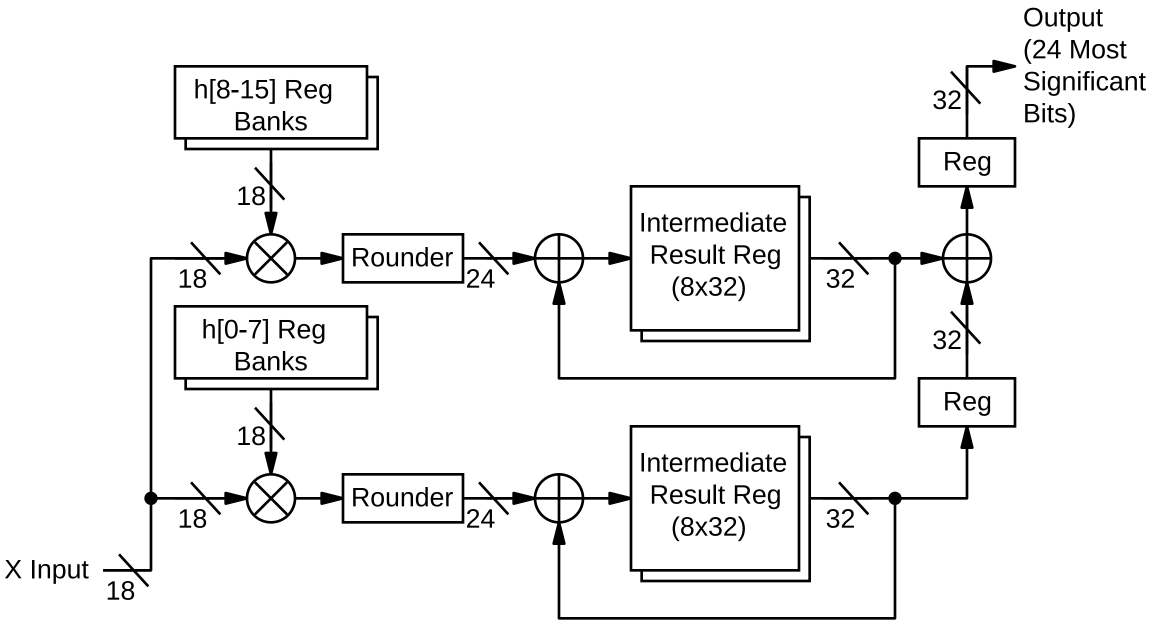


Figure 3.2: Custom VLSI Chip Architecture

3.3 Custom VLSI Noise Analysis

As shown in Figure 3.2, the multiplier in each MAC multiplies an 18-bit input by an 18-bit coefficient. This would produce a 36-bit output. However, to keep the logic downstream of the multiplier a reasonable size, the bit-width of the multiplier's output has been reduced to 24-bits by a rounder. The rest of the chip uses full-precision, i.e., each time a summation is performed (the only operation that occurs after the multiplication), the summation's output grows a bit to handle the one-bit word growth normal in addition. [Gre96, OS89]

Rounding is performed by adding the most significant bit (shifted right to bit position 0) of the bits to be removed to the bits that are to remain. For example, when rounding the 36-bit multiplication output to 24 bits, the top 24-bits of the 36-bit output are kept and added to the 11th bit of the 36-bit output. Since each bit represents ½ the power of the bit to its left, adding the most significant bit of the part of a number to be rounded off to the bits on its left is the same as rounding the bits on its left up by one if the bits on the right are greater than 0.5, the typical method of rounding in math. However, since the rounded result is no longer exact (because bits were removed), noise is introduced. This noise can be approximated as additive noise added at the rounder with a mean of zero and a variance of ([OS89, OW75])

$$\sigma_e^2 = \frac{2^{-2B}}{12}, \tag{3.1}$$

where B is the number of bits the result has been rounded to. The noise approximation is modeled simply as additive noise as shown in Figure 3.3. [OS89, OW75]

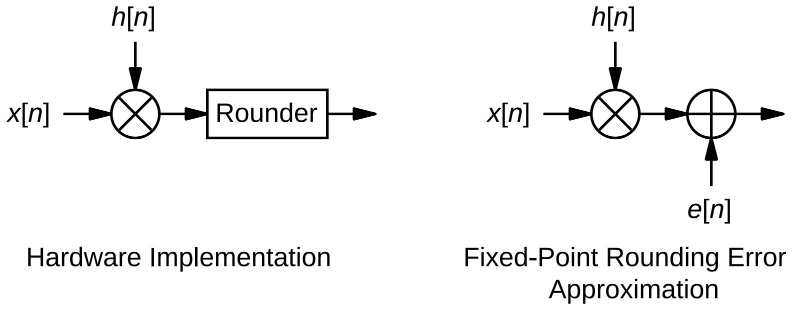


Figure 3.3: Rounding Noise Approximation

Examining Figure 3.2 with the rounders replaced with noise sources (as shown in Figure 3.3) shows that the noise sources will simply be added together. Since they are assumed to be independent, their means and variances can be summed, giving a total noise source for the 512-tap custom VLSI chip filter with 24-bit rounded multiplier outputs a mean of zero and a variance of ([OS89, OW75])

$$\sigma_e^2 = \frac{512}{12}2^{-2(24)} = 6.06\times10^{-13}. \tag{3.2}$$

With the chip's output variance computed, the number of bits required for a single rounder to achieve the variance in Equation (3.2) can be solved (i.e. if the custom VLSI chip had only one rounder at its output instead of 512 separate rounders):

$$B_{filter} = \frac{\log_2\frac{1}{12} - \log_2\sigma_e^2}{2} = \frac{\log_2\frac{1}{12} - \log_2 6.06\times10^{-13}}{2} = 19.5\text{ bits}. \tag{3.3}$$

This means that the final output of the custom VLSI chip is the same as if the filter had no rounding internally, and the output was rounded to 19.5 bits. Therefore, only the upper 19 bits of the chip's 24-bit output are exact; the extra bits in the chip's output data bus contain noise introduced by the rounding at each multiplier.

To experientially determine the chip's output precision, a fixed-point MATLAB simulation of the custom VLSI chip was built. The simulation performed 24-bit rounding at the output of each multiplier as described above, and compared this filtered output for a random set of data and coefficients with the ideal convolution of the data and coefficients. The difference of the ideal convolution's results and the rounding filter's results was equal to the noise introduced by rounding. The variance of this difference (the rounding noise variance) was then computed and used in Equation (3.3) to compute the number of valid bits output by the custom VLSI chip. This simulation indicated that (with several different sets of random data and coefficients) the custom VLSI chip actually outputs 18 bits of valid data.

Chapter 4

Problem

4.1 FPGA Benefits to Radar Front-End Signal Processing

As FPGA technologies become more capable, the same filtering operations currently implemented in custom VLSI devices can now be attempted to be implemented in a FPGA. a FPGA implementation would have several major benefits:

1. Reduction in design, manufacturing, and testing costs, time, and resources relative to custom VLSI designs.

2. Flexibility to implement different filtering functions using the same re-programmable FPGA devices.

3. Ability to upgrade the design to higher ADC sampling rates by substituting more capable FPGA devices commensurate with the Moore's law progression in silicon technology (without requiring the new fabrication runs custom chips necessitate).

4. Wider vendor sources of FPGA technologies than available with custom VLSI designs.

The most limiting factor in permitting the use of FPGAs to date for the more advanced classes of DSP applications (in particular, radar front-end signal processing) is their inability to reach the required throughputs with high precision, which are on the order of several billion GOPS with at least 16-bits. Most high performance demonstrations to date are based on a few bits of precision. Therefore, this thesis attempts to demonstrate that FPGA technology can implement filtering fast enough to allow the benefits of reconfigurable computing to be applied to radar front-end signal processing DSP functions.

4.2 Problem

To demonstrate the current state of FPGA technology and its application to front-end signal processing, FPGA implementations meeting the design requirements of MIT Lincoln Laboratory's custom VLSI FIR chip discussed in Chapter 3 were created. The FPGA designs were required to:

- Perform 512-tap real FIR filtering
- Accept 16-bit data at a maximum input rate of 5 MHZ
- Operate with a maximum chip clock frequency of 40 MHz (eight times the input rate)
- Output data with the same precision as the custom VLSI chip (18-bits as described in Section 3.3)
- Use two swapable banks of 18-bit coefficients, one active and one load-able
- Have coefficient banks that must be able to be switched every 1 ms (i.e. reloading all 512 coefficients must take place in less than 1 ms)
- Fit into the largest Xilinx FPGA available, the Virtex XCV1000.

The 512-tap real FIR filter specification was chosen as this is the mode in which the custom VLSI chip runs the most efficiently. Although the chip is used in-system in an in-phase and quadrature mode, the MACs are not fully utilized, so building an efficient FPGA design would not be a fair comparison. In the 512-tap design, each of the 64 custom VLSI MACs perform eight filter tap operations per input sample. The 512-tap design allows a direct performance comparison between the two types of hardware. Once a FPGA design can meet the 512-tap real mode filtering requirements of the custom VLSI chip, it will easily be able to meet those of the DIQ mode, as moving to that mode simply changes the manner in which the internal filter results are summed together.

The maximum input rate in the design specifications for the custom VLSI chip operating in its 512-tap real FIR filter mode is 5 MHz. In this mode, the chip requires a clock rate eight times the sample rate since it is processing eight taps per input sample. Therefore, the maximum clock rate is 40 MHz, although the custom VLSI chip was tested to 44 MHz.

16-bit inputs were chosen as the custom VLSI chip is currently being used with an ADC with a 14-bit output, and it is unlikely the VLSI chip would operate with an ADC of more than 16-bits in the future. The only reason for having the custom VLSI chip designed to accommodate 18-bit input data was to facilitate word growth if several custom A1000 chips were used in a cascaded mode.

Two banks of coefficients were used in the custom VLSI chip, and are required for the FPGA so that one bank stores the active filter’s coefficients while the other bank is being loaded with new coefficients. The banks may then be swapped by an external control so that the new coefficients may become instantly active.

Although the need to change coefficients could be viewed as an ideal application for reconfiguration, using swapable coefficient banks is more efficient. It has been proposed that the ability of a FPGA to be reconfigured in-system be used to implement a single filter with fixed coefficients that is reconfigured when a coefficient switch is desired. The design requirements specify an instantaneous switch between the active coefficients and the new set of loaded coefficients, which prohibits using a single bank of coefficients that could be changed via the FPGA's reconfiguration ability. The filter would be inactive during the reconfiguration time, which is unacceptable.

One solution might be to have one filter operational while a second filter was being configured in the same chip. This would require the FPGA to have partial reconfiguration ability, and wouldn't save any space over two coefficient banks, because two filters would still have to be present in the chip during coefficient updates. In addition, the filter coefficients would have to be known in advance so that configurations using constant coefficients could be mapped, placed, and routed by the Xilinx software to be ready for loading into the FPGA. These configurations would then have to be stored off-chip to be recalled and loaded into the FPGA. In most applications, the filter responses are not known beforehand, so creating and storing every configuration for every set of possible coefficients is not feasible.

Chapter 5

Xilinx Virtex FPGA Technology

5.1 FPGA Selection

The Xilinx Virtex FPGA was chosen to implement the 512-tap FIR design because of several factors. At the time of this thesis, the Virtex XCV1000, with 1,124,022 gates, had the most capacity of any commercially available FPGA [Xil98b]. The Virtex series of FPGAs were also determined to be ideal for DSP applications as they had been designed with vector-based routing intended to carry large data busses within the chip instead of traditional FPGA routing schemes designed more for control logic [Xil98a]. In addition, PC-based reconfigurable computing boards from Annapolis Micro Systems, a company that MIT Lincoln Laboratory and DARPA are working with to develop reconfigurable computing solutions, are available with Virtex devices and could be used to implement solutions developed in this thesis in real hardware. Finally, prior Xilinx knowledge finalized the decision to use Xilinx Virtex FPGAs for this thesis.

5.2 Virtex Overview

FPGAs are similar to custom designed chips in that they implement specific circuitry for a particular function. The major difference is that a FPGA is configured by a bitstream instead of by being hardwired through fabrication at a factory. This means that a FPGA's internal circuitry may be altered an unlimited number of times.

FPGAs may be classified as “coarse-grained” or “fine-grained,” referring to the number and complexity of each basic logic element in the FPGA. Xilinx Virtex series chips are coarse-grained, and have logic units based on look-up tables (LUTs) and registers. The basic Virtex logic element is a Configurable Logic Block (CLB) slice. Two slices are present in each CLB. Each slice contains two 4-input, 1-output LUTs and two registers. The interconnections between these elements are configured by multiplexors controlled by SRAM cells programmed by a user’s bitstream. The LUTs allow any function of five inputs, any two functions of four inputs, or some functions of up to nine inputs to be created within a CLB slice. The outputs of these functions may be registered, or the registers may be used independently of the LUTs. This structure allows a very powerful method of implementing arbitrary, complex digital logic. [Xil98a]

The Xilinx slices also have the ability to implement distributed memory instead of logic. Each 4-input LUT in a slice may be used to implement a 16x1 ROM or RAM, or the two LUTs may be combined together to create a 32x1 ROM or RAM or a 16x1 dual-port RAM. This allows each slice to trade logic resources for memory in order to maximize the resources available for a particular application. A block diagram of a Xilinx Virtex CLB showing both slices is illustrated in Figure 5.1. [Xil98a]

The CLBs in a Virtex FPGA are connected via programmable interconnect called the general purpose routing. This interconnect consists of differing length lines, some connecting adjacent CLBs together, while some span the entire length of the chip and others are designed for high fan-out signals such as clocks. The connections between the interconnect and the CLBS are con-

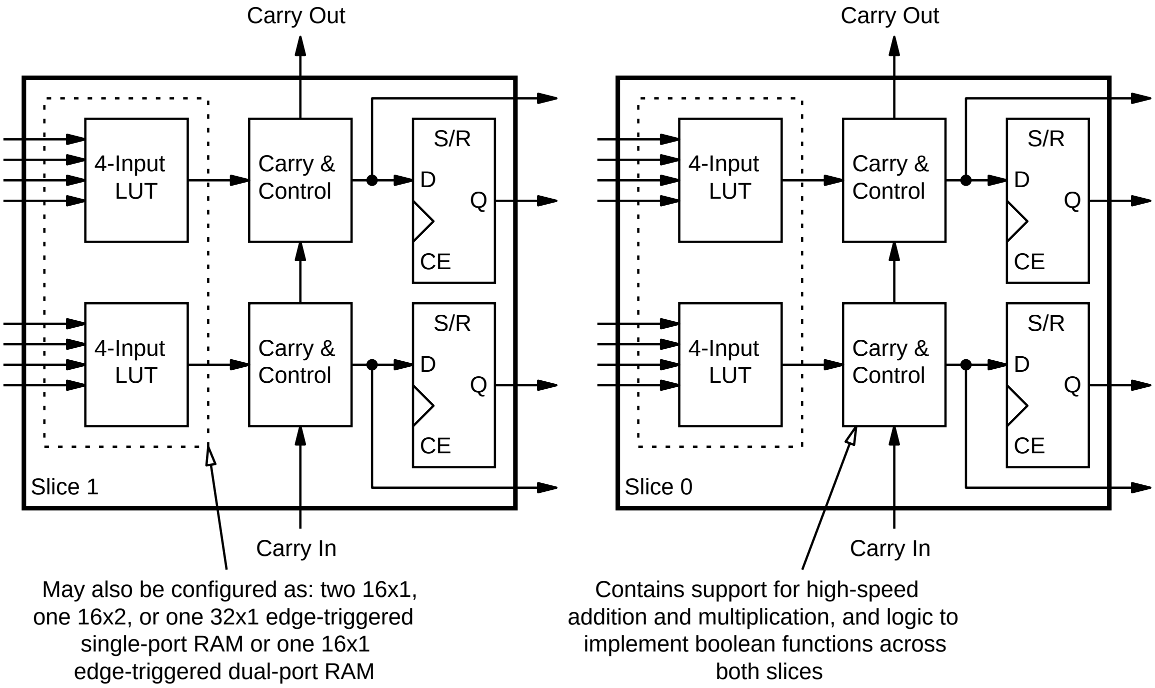
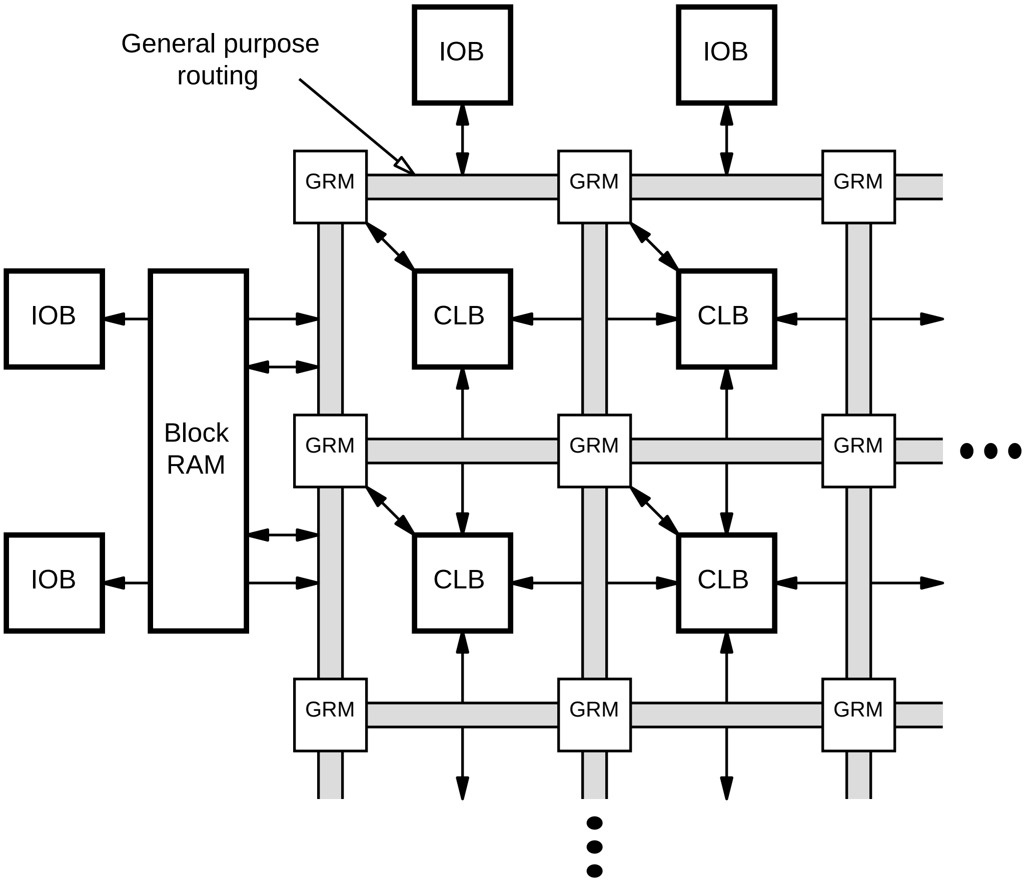

Carry Out
Carry Out
4-Input LUT
Carry & Control
S/R
D
Q
CE
4-Input LUT
Carry & Control
S/R
D
Q
CE
Slice 1
4-Input LUT
Carry & Control
S/R
D
Q
CE
4-Input LUT
Carry & Control
S/R
D
Q
CE
Slice 0
Carry In
Carry In
May also be configured as: two 16x1, one 16x2, or one 32x1 edge-triggered single-port RAM or one 16x1 edge-triggered dual-port RAM
Contains support for high-speed addition and multiplication, and logic to implement boolean functions across both slices

Figure 5.1: Block Diagram of Virtex CLB (Two Slices)

trolled by switch matrices called general routing matrices (GRMs). The programmable interconnect allows mappings that require local communication to be handled efficiently along with requirements for arbitrary, longer-distance, routing demands. In addition to the programmable interconnect, there are a few dedicated routing resources. One example is the carry-chains between CLBs that allow high-speed carry propagation through a series of slices, enabling high-speed adders and other arithmetic units to be designed in a chain of CLBs. Connections between the internal routing and the external world are made through Input/Output Blocks, or IOBs, which contain input/output registers and connect directly to a package pin. [Xil98a]

The Virtex FPGAs also include 4,096-bit block RAM units on the edges of the FPGA. These resources are ideal when large amounts of memory are required that would not use the small, distributed CLB-based memory efficiently. Finally, the Virtex also has advanced clock management resources built in, including a delay locked loop (DLL) that reduces clock skew and can divide (by up to 16) or multiply (by 2) external clocks for slower or faster internal clocking. Figure 5.2 shows a block diagram of a Virtex series chip. [Xil98a]



CLB = Configuable Logic Block
GRM = General Routing Matrix
IOB = Input/Output Block

Figure 5.2: Virtex Block Diagram

The highly replicated, register rich architecture of the Virtex makes it suitable for custom computing applications. Each slice can perform a two-bit computation or look-up, allowing a systolic structure of processors to be built out of the regular array of CLBs in a Virtex. There are cases when a finer grain structure may be more efficient, as the CLB structure may not be the most efficient medium for very small systolic-cell based arithmetic (for example, the bit-level systolic FIR filter design investigated in this thesis).

The largest Virtex, the XCV1000, has 12,288 CLB slices and 131,072 block RAM bits [Xil98a].

5.3 Computational Unit Implementation

To illustrate the capacity of a slice, two commonly used DSP computational units, an adder and a multiplier, are presented with their area in terms of CLB slices.

The Virtex has dedicated fast-carry chain resources built into each CLB. Two adder bits can fit into a single slice so that a *b*-bit adder consumes $^{b}/_{2}$ CLB slices. A 16-bit adder would require 8 slices. [Xil98a]

The Virtex also has dedicated multiplication resources so that two multiplication bits can fit into a single slice. An *a*-bit by *b*-bit multiplier requires approximately

$$\frac{b\log_2 b + (b-1)a}{2} \tag{5.1}$$

CLB slices.[1] A 16x16-bit multiplier would require about 152 slices.

[1] Based on an optimized multiplier built for Virtex by Xilinx.

Chapter 6

Algorithms and Techniques

6.1 Overview

Several different algorithms or filtering techniques were investigated for implementing a FIR filter in a FPGA device. These included FIR filtering using parallel multipliers and accumulators, a bit-level systolic array, distributed arithmetic, fast FIR algorithms, and frequency domain filtering. The specifics of each are discussed in this chapter.

6.2 FIR Filtering

A FIR filter computes the discrete convolution of an input $x[n]$ and a finite-length filter response $h[n]$. This convolution can be written as

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k], \tag{6.1}$$

where N is the length of the filter response $h[n]$, and is referred to as the number of taps in the filter.

6.3 Parallel Multipliers and Accumulators

The most direct realization of a FIR filter is to calculate the output $y[n]$ using parallel multipliers and accumulators (MACs). The parallel MAC structure is illustrated in Figure 6.1, and is derived directly from the FIR convolution in Equation (6.1) [OS89]. In this structure, each MAC computes the product of the delayed input and the tap's active coefficient. The outputs from each multiplier are then accumulated together to produce the filter's output.

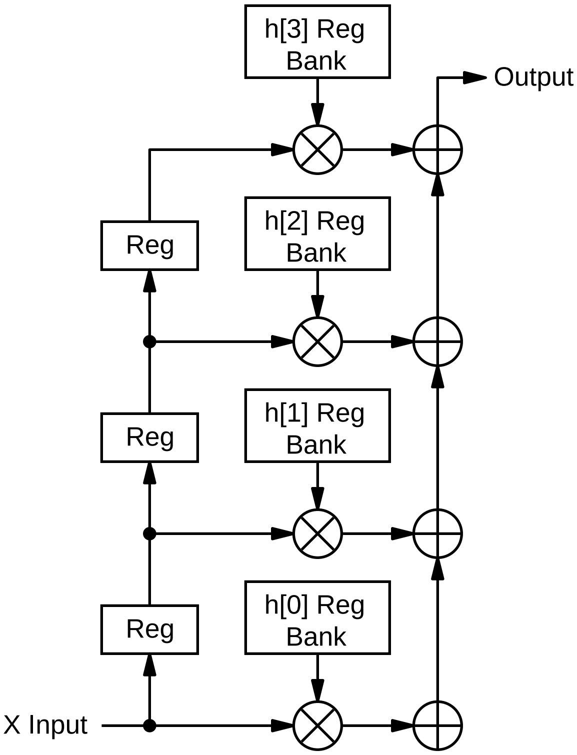


Figure 6.1: Direct Parallel Multiplier and Accumulator FIR Structure

The structure in Figure 6.1 has long combinatorial delays through the accumulation chain, so the summer tree network shown in Figure 6.2 or the transposed form shown in Figure 6.3 are often used in actual FIR computational hardware. Both forms produce the same output, but can have their accumulation chains pipelined to increase performance. The benefit of the transposed form is that each MAC communicates only with adjacent MACs, as Figure 6.3 shows. This allows the MACs to be placed in a linear systolic fashion, where adjacent MACs are placed next to each other in a line so that each MAC only has routes to and from its nearest neighbors. This maximizes the performance of the design while minimizing its area. The tree network requires long, complicated route lengths to the inputs of each stage of adders as the number of MACs that have been summed together for a given adder increases and the MACs grow farther apart. This prohibits a simple linear distribution of MAC cells and slows the design's performance due to the long routes.

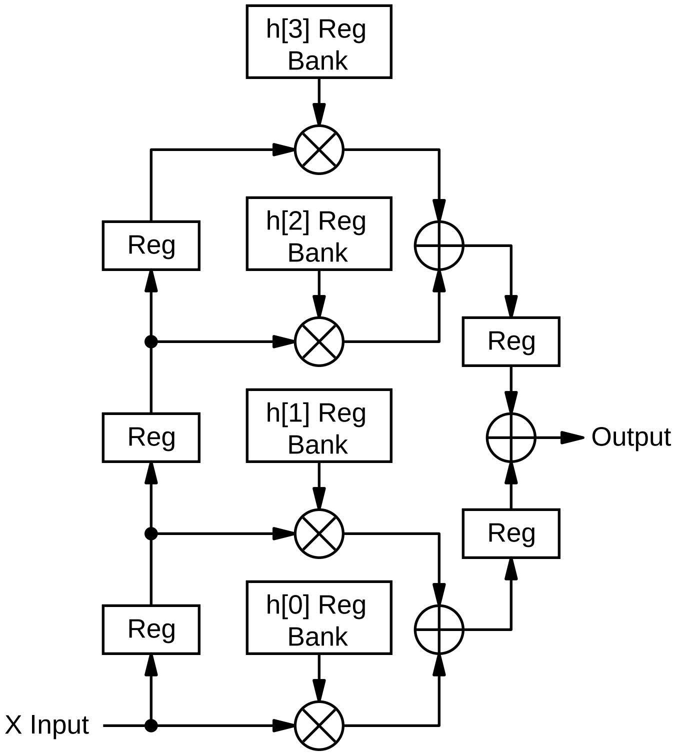


Figure 6.2: Parallel MAC Filter With Summer Tree

One problem with the transposed form of the parallel MAC filter is that it requires a large fan-out on the *X* input signals, as they must connect to every MAC. To reduce this fan-out while maintaining pipelining in the accumulation chain and allowing the MACs to be placed in a linear systolic fashion, an additional stage of pipelining in both the inputs and outputs of each MAC can be introduced into the direct filter as shown in Figure 6.4.

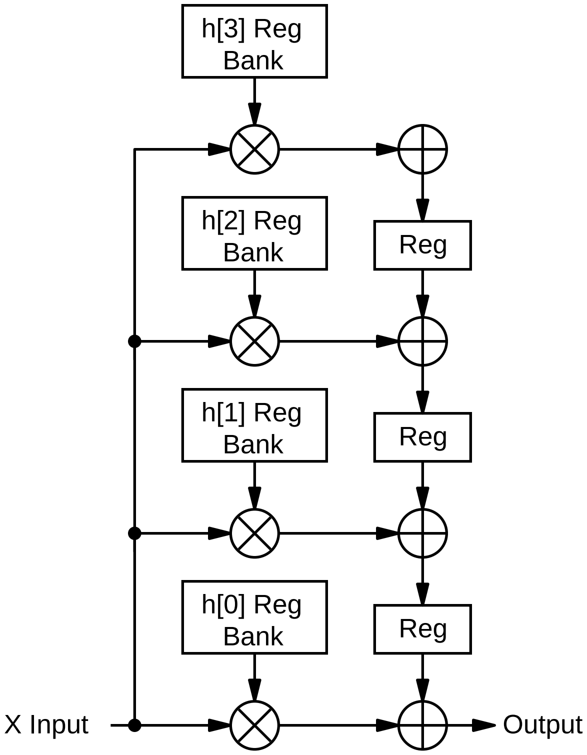


Figure 6.3: Transposed Form Parallel MAC Filter

A frequent method used to decrease the area of a parallel MAC approach to FIR filtering is to increase the number of taps computed per MAC. This is the technique used in the custom VLSI chip [Gre96, MM99, MMT00]. One variation of this technique is shown in Figure 6.5. In this figure, a single multiplier is re-used eight times to compute the product of eight *X* values multiplied by eight coefficients for each input into the filter. An eight-word deep RAM stores the eight coefficients for the tap, and a seven-word long shift register stores the *X* values.

This architecture is similar to the one used in the custom VLSI chip (see Section 3.2 and Figure 3.2), except that shift registers have been used to store multiple input values for each MAC instead of RAM storing the multiplier's outputs. The result is the same except that storing multiple inputs per tap

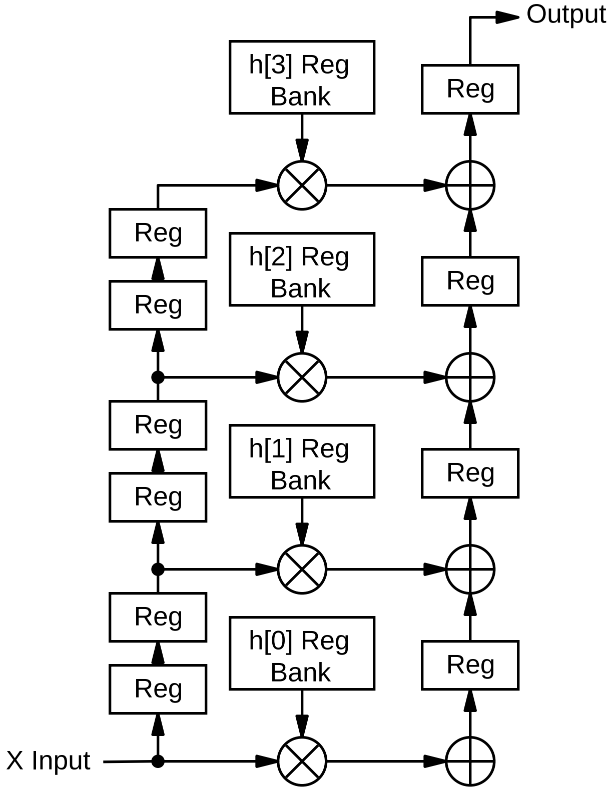


Figure 6.4: Direct Form Pipelined MAC Fir Filter

requires less memory since the inputs are only 16-bits long versus the 24-bit multiplier outputs.

The shift registers are loaded with a new value at the beginning of each eight-clock cycle. The shift registers are then fed their outputs back into their inputs for the next seven clock cycles. This moves the input that was shifted into the register at the beginning of the eight clock cycles to the second shift register position at the beginning of the next eight clock cycles so that the next new value is loaded into the first position. After eight clock cycles, this input becomes the new value to the next MAC's shift registers. At the same time, the shift registers are arranged so that eight consecutive input values are supplied to the multiplier to be multiplied by eight consecutive coefficient values. These will be added together by the final accumulator shown in Figure 6.5.

The output of the first two MACs' shift registers and coefficient registers (i.e. the multiplier inputs) are shown in Table 6.1 for two-tap MACs. With eight-tap MACs, the movement of inputs through the shift registers produces the same effect with 64 multipliers as if 512 multipliers had been used with a clock rate equal to the sample rate.

	Mac 0				Mac1				
Clock Cycle	MAC Input	Shift Register		Coeff Output	MAC Input	Shift Register		Coeff Output	Chip Output
		In	Out			In	Out		
0	$x[0]$	$x[0]$		$h[1]$				$h[3]$	$x[0]*h[0]$
1		$x[0]$	$x[0]$	$h[0]$				$h[2]$	
2	$x[1]$	$x[1]$	$x[0]$	$h[1]$				$h[3]$	$x[0]*h[1]+$ $x[1]*h[0]$
3		$x[1]$	$x[1]$	$h[0]$				$h[2]$	
4	$x[2]$	$x[2]$	$x[1]$	$h[1]$	$x[0]$	$x[0]$		$h[3]$	$x[0]*h[2]+$ $x[1]*h[1]+$ $x[2]*h[0]$
5		$x[2]$	$x[2]$	$h[0]$		$x[0]$	$x[0]$	$h[2]$	

Table 6.1: Two Taps Per MAC FIR Filter Example

6.4 Bit-Level Systolic Array

An approach which has proven to be a very efficient VLSI implementation at MIT Lincoln Laboratory [Son] is a fully-efficient bit-level systolic structure by Chin-Liang Wang [WWC88]. With this technique, single-bit processors compute each tap's multiplication partial products and accumulate tap outputs together in a systolic array. As inputs propagate through the array, filtered outputs are produced. The systolic nature of this approach lends itself well to

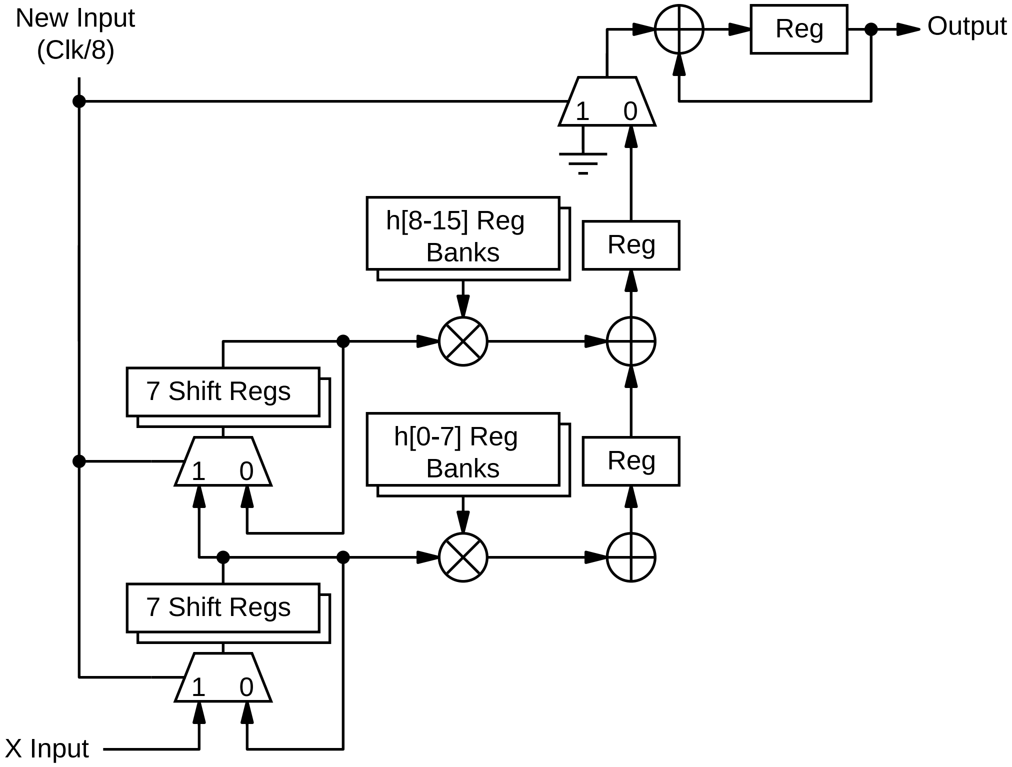
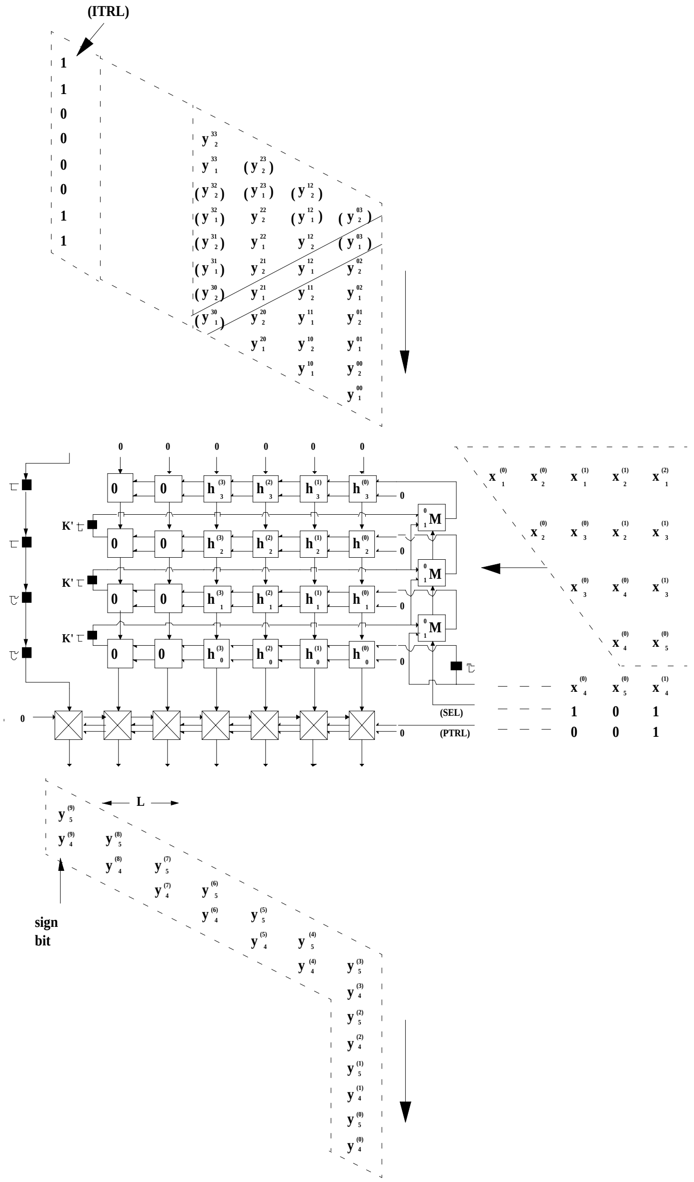


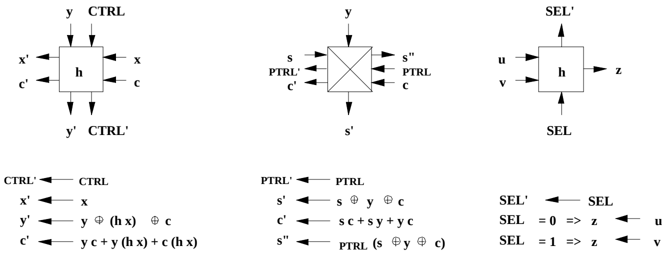
Figure 6.5: Eight Taps Per MAC FIR Filter

VLSI, and would be ideal for a FPGA if it was area-efficient, as it would limit the routing requirements in the FPGA to local connections between CLBs.

The configuration of this array and the logic required to implement each cell is shown in Figure 6.6. Since this technique did not turn out to be efficient in a FPGA architecture (see Section 7.3), the derivation if its structure will not be included here (see [Son, WWC88] for more information). It is presented to show the differences between architectures optimized for a FPGA's coarse-grained structure versus architectures optimized at the transistor level for a VLSI approach.



Fully Efficient Bit-Level Systolic Array Architecture (4 Taps by 4 Bits)



Array Cells

Figure 6.6: Bit-Level Systolic Array

6.5 Distributed Arithmetic

The following section describes DA, and is drawn from information presented in [MM99, Whi89, Gos96a, Gos95, Gos96b, New95, Xilb].

DA works by distributing the bit arithmetic of the sum-of-products (also called the vector dot product) used to calculate the FIR filter output given in Equation (6.1). This equation will be re-written as

$$y(n) = \sum_{k=0}^{N-1} A_k x_k(n), \tag{6.2}$$

where $A_k = h[k]$.

A FIR filter is typically implemented with some variation of Figure 6.1 or Figure 6.3, where a summation of the results of N multipliers each calculating an $A_k x_k(n)$ product produce the output for a given n.

The number format used in the custom VLSI chip and in the FPGA design is 2's complement fractional fixed-point. In this format, the binary point is to the right of the most significant bit so that the most significant bit of a number represents -1, and each subsequent bit represents a power of $\frac{1}{2}$. Using this format, the variable x_k may be written as

$$x_k = -x_{k0} + \sum_{b=1}^{B-1} x_{kb} 2^{-b}, \tag{6.3}$$

where x_{kb} is the bth bit of x_k, and B is the number of bits in the input variable.

Substituting Equation (6.3) into Equation (6.2) gives (n has been dropped, as we are only concerned with a single given output sample)

$$y = \sum_{k=0}^{N-1} A_k \left(-x_{k0} + \sum_{b=1}^{B-1} x_{kb} 2^{-b} \right), \tag{6.4}$$

which when rewritten, gives

$$y = -\sum_{k=0}^{N-1} x_{k0} A_k + \sum_{b=1}^{B-1} 2^{-b} \left(\sum_{k=0}^{N-1} x_{kb} A_k \right). \tag{6.5}$$

Explicitly writing out the summations results in the DA equation

$$\begin{aligned} y = & -[(x_{00}A_0 + x_{10}A_1 + \ldots + x_{(N-1)0}A_{N-1})] \\ & + [x_{01}A_0 + x_{11}A_1 + \ldots + x_{(N-1)1}A_{N-1}]2^{-1} \\ & \ldots \\ & + [x_{0(B-1)}A_0 + x_{1(B-1)}A_1 + \ldots + x_{(N-1)(B-1)}A_{N-1}]2^{B-1}. \end{aligned} \tag{6.6}$$

Each multiplication of a x_{kb} term and an A_k term is the product of a coefficient word with an input bit. This can be implemented by using an AND gate between each bit of the coefficient word and the input bit. Each scaling factor (2^{-i}) can be implemented by shifting the data to be scaled right i bits. Equation (6.6) therefore becomes the summation of the scaled summation of a series of AND gates. This operation could be performed in parallel (as described in Section 6.3), or bit-serially, where each clock cycle a single bit from every x_k is multiplied by the corresponding A_k, forming one bracketed term in Equation (6.6). These partial products are then accumulated together with the appropriate scaling to produce a final multiplier output. An example of bit-serial multiplication for a single coefficient and input is shown in Figure 6.7.

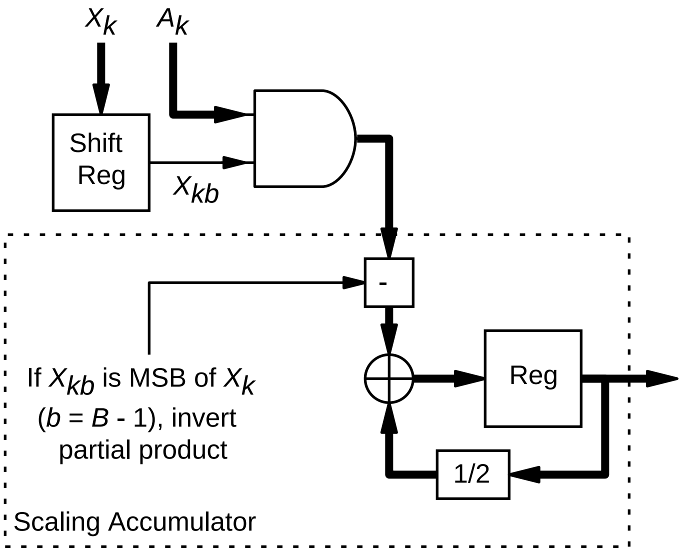


Figure 6.7: Bit-Serial Multiplier

As Figure 6.7 illustrates, each clock cycle a single partial product consisting of one bit of the input x_k multiplied by the coefficient A_k is produced. This partial product is then added to an accumulating sum of partial products, which has been shifted right one bit (multiplied by ½). This operation produces the following result for a four-bit input (with each term in parenthesis being computed each clock cycle):

$$y_k = (((x_{k3}A_k)2^{-1} + x_{k2}A_k)2^{-1} + x_{k1}A_k)2^{-1} - x_{k0}A_k, \qquad (6.7)$$

which when simplified, gives

$$y_k = x_{k3}A_k2^{-3} + x_{k2}A_k2^{-2} + x_{k1}A_k2^{-1} - x_{k0}A_k, \qquad (6.8)$$

and finally, results in the product of the input and the coefficient after four clock cycles:

$$y_k = x_kA_k. \qquad (6.9)$$

To maintain full-precision, the accumulator must be able to hold the entire multiplied result. The number of bits required is the number of bits in the input data + the number of bits in the coefficients.

The MAC structure in Figure 6.1 can be implemented with the bit-serial multiplier in Figure 6.7 as shown in Figure 6.8.

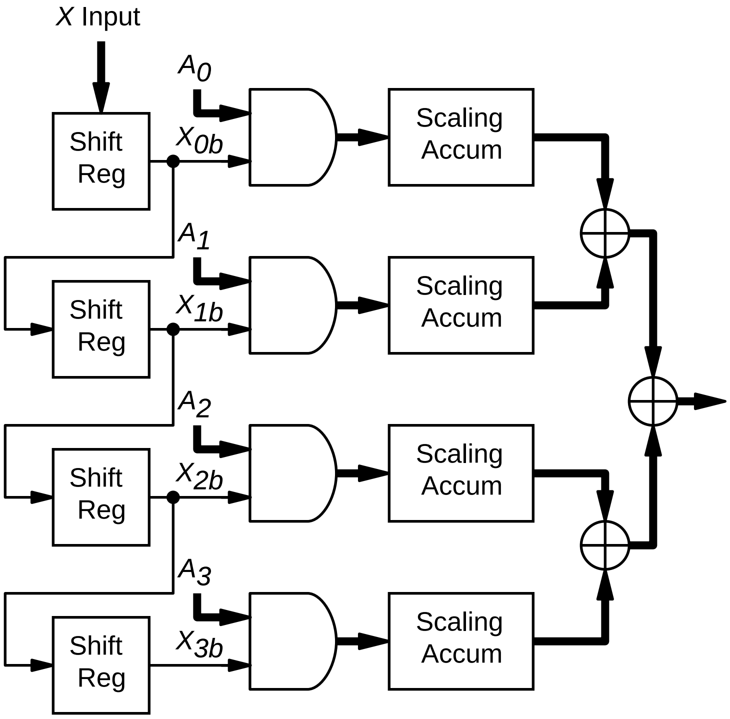


Figure 6.8: Four MAC Filter Using Bit-Serial Multiplier

In Figure 6.8, a parallel input to the FIR is converted into a serial stream of bits. Each clock cycle, one bit of the input is presented to the first scaling accumulator, and placed into a serial shift register for the next tap, so that each tap's input sample is presented to each scaling accumulator in a serial fashion. Each tap takes *B* clock cycle to produce a product, which are then summed together to produce an output sample. However, as the bracketed terms in Equation (6.6) show, the partial products computed by each AND gate can be summed together first, then accumulated with scaling. In this method, one bracketed term in Equation (6.6) is computed each clock cycle, so B clock cycles are still required, yet each tap requires less hardware, since only one master scaling accumulator is now necessary. The new FIR structure is shown in Figure 6.9.

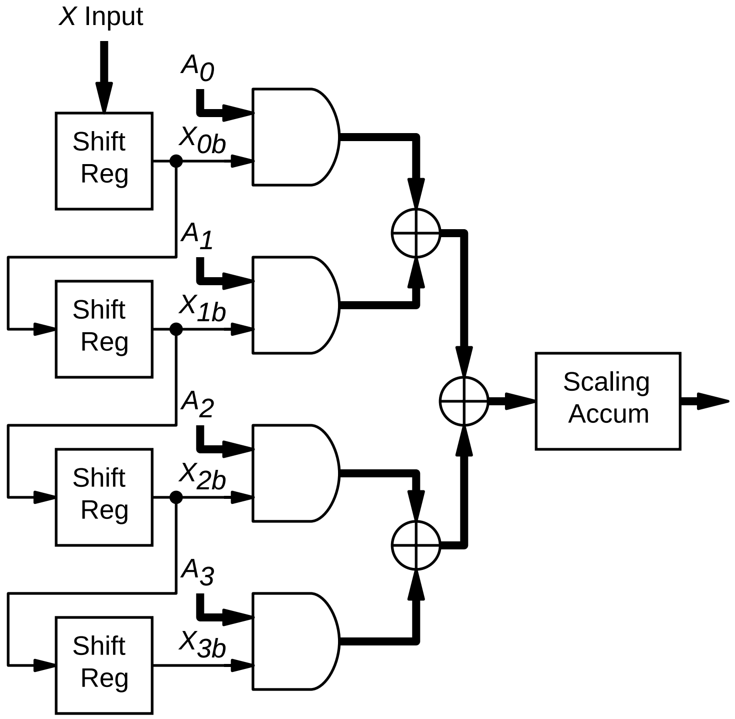


Figure 6.9: Serial Distributed Arithmetic FIR

To maintain full precision in this case, the scaling accumulator is now required to hold the number of bits in the input + the number of bits in the coefficients + the number of bits added due to word growth through the adder stages (1 bit per stage).

If the coefficients for the filter are constant, then the output of the summer tree depends solely on the single-bit inputs to each tap. With this being the case, the storage registers for the coefficients, the AND gates, and the summer tree can all be replaced by a single look-up-table addressed by the single-bit shift register outputs as shown in Figure 6.10.

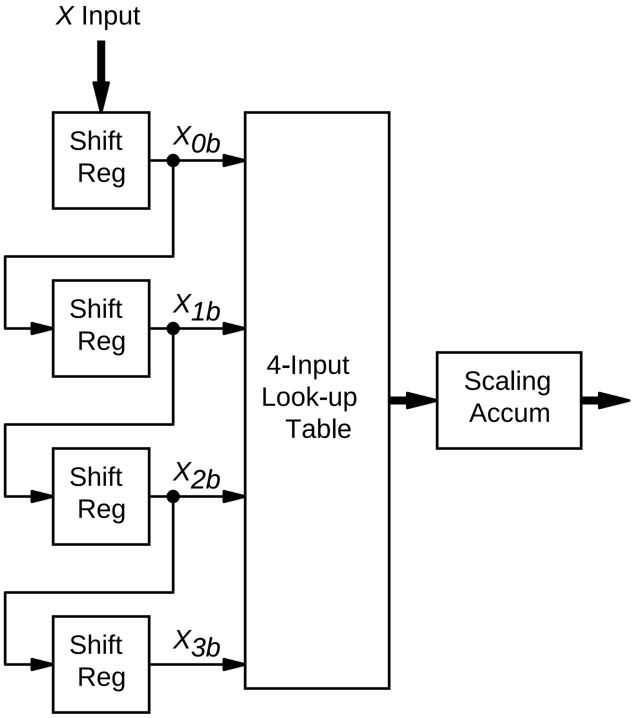


Figure 6.10: Serial Distributed FIR Using LUT

With four taps as shown in Figure 6.10, a LUT with 16 entries is required. Each 4-bit address into the LUT can be thought of as being a sum of coefficients: if a particular address bit is high, then that address' sum should include the corresponding coefficient. Table 6.2 shows the 16 locations in the LUT, and what values they should hold.

LUT Address	LUT Output
x_{3b} x_{2b} x_{1b} x_{0b}	Sum
0 0 0 0	0
0 0 0 1	A_0
0 0 1 0	A_1

Table 6.2: Contents of 4-Tap LUT

LUT Address	LUT Output
0 0 1 1	$A_0 + A_1$
0 1 0 0	A_2
0 1 0 1	$A_0 + A_2$
0 1 1 0	$A_1 + A_2$
0 1 1 1	$A_0 + A_1 + A_2$
1 0 0 0	A_3
1 0 0 1	$A_0 + A_3$
1 0 1 0	$A_1 + A_3$
1 0 1 1	$A_0 + A_1 + A_3$
1 1 0 0	$A_2 + A_3$
1 1 0 1	$A_0 + A_2 + A_3$
1 1 1 0	$A_1 + A_2 + A_3$
1 1 1 1	$A_0 + A_1 + A_2 + A_3$

Table 6.2: Contents of 4-Tap LUT (Continued)

To keep the output of the LUT at full precision, the LUT should be two bits larger than the size of the coefficients to accommodate for word growth through the additions in Table 6.2.

The 16x1 RAM units within the Xilinx CLBs are ideal candidates for this sort of DA scheme. One bit of a single 4-input LUT can fit into one of these units with no unused logic.

For FIR filters larger than 4-taps, the filter can be broken into four tap groups, each constructed as shown in Figure 6.10. For example, a 16-tap FIR is shown in Figure 6.11. To eliminate overflow, each adder stage must grow by one bit, and the scaling accumulator must also grow accordingly in size (the scaling accumulator could drop the lower bits in its accumulation if less precision is required).

Although larger LUTs could be used with less adders, LUTs larger than four inputs do not save space. For example, a five-input LUT would require 32-entries and take up two 16x1 RAM units (an entire slice). However, if these two 16x1 RAM units were used separately, they could each be addressed by four taps, allowing an entire slice to handle eight taps. The extra adder needed to sum the two four-input LUTs together would not significantly increase the area enough to justify a five-input LUT.

6.5.1 Parallel Distributed Arithmetic

A benefit of distributed arithmetic is that it easily allows a trade-off to be made between the filter's area and performance. By doubling the filter's area, the filter's throughput or sample rate can be doubled without changing the clock rate that the individual filter components operate at. In the serial distributed arithmetic (SDA) designs discussed above, a clock rate *B* times the sample rate is required, as one clock cycle is needed to look up a partial product for each bit of *x*. However, by taking advantage of a feature inherent in the DA equation, Equation (6.6), fewer clock cycles can be required per input sample. Presently, one term in the equation has been computed per clock cycle. However, any number of terms can be computed per clock cycle (referred to as parallel distributed arithmetic, or PDA). For example, if two terms are computed per clock cycle, then $^{B}\!/_{2}$ clock cycles are required to compute an output.

To compute two terms per clock cycle, two identical SDA FIR filters as described above must be constructed. Each filter will compute one term in Equation (6.6) so that two terms are computed per clock cycle. One filter will

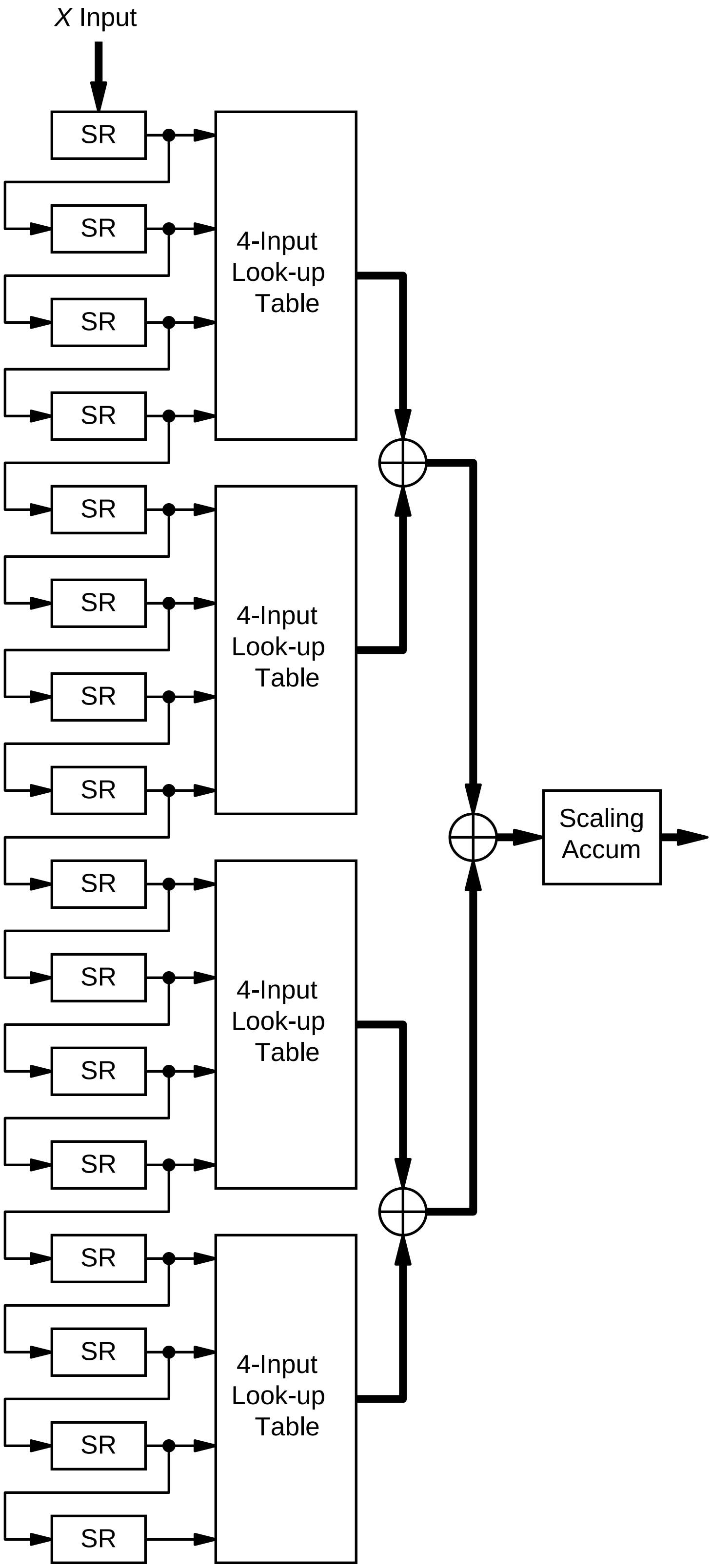


Figure 6.11: 16-Tap Serial Distributed FIR

compute outputs for even input sample bits, and the other filter will compute outputs for odd input sample bits. For example, on the first clock cycle, the first filter will compute the output term associated with x_{k0} while the other filter computes the output term associated with x_{k1}. These outputs are then added together with the first filter's output (the bit 0 term) scaled by $^1\!/_2$, and then sent to the scaling accumulator. Each clock cycle, the scaling accumulator scales its registered accumulation by $^1\!/_4$ to accommodate for the fact that it is handling two partial products per clock cycle instead of one. The 2-bit PDA approach requires twice as much area as the serial approach, but has twice the performance, and is illustrated in Figure 6.12.

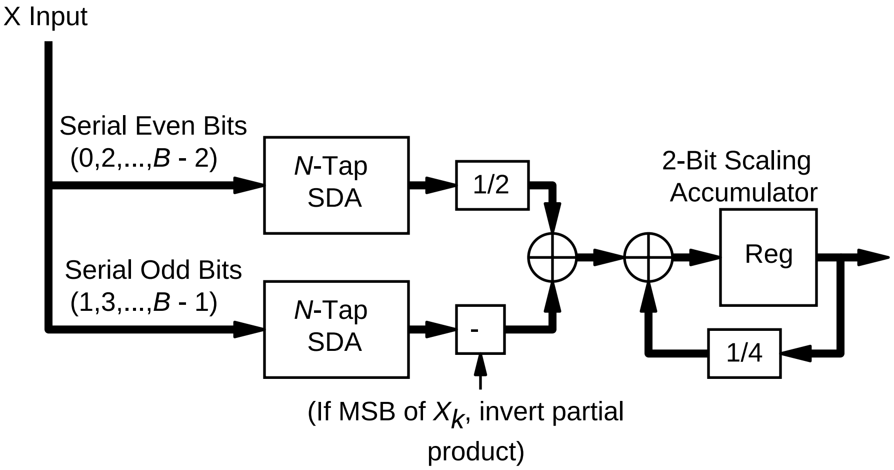


Figure 6.12: 2-Bit Parallel Distributed Arithmetic FIR

6.6 Fast FIR Algorithm

The class of fast FIR algorithms (FFA) attempt to increase the parallelism of the FIR structure without a linear increase in area [PP97, CKJ+98]. Traditionally, to double the throughput of a FIR filter without increasing the clock rate of the filter itself, the filter area would have to be doubled.

Doubling the throughput of a FIR filter without changing its internal clock rate means that two outputs are to be calculated each clock cycle. These two outputs will be referred to as $y[2j]$ and $y[2j+1]$. Producing two outputs per clock cycle would require two inputs per clock cycle as well, $x[2j]$ and $x[2j+1]$. This leads to the following set of equations:

$$\begin{aligned} x_0[j] &= x[2j] \\ x_1[j] &= x[2j+1] \\ y_0[j] &= y[2j] \\ y_1[j] &= y[2j+1], \end{aligned} \tag{6.10}$$

where x_0 and y_0 represent the even inputs and outputs, and x_1 and y_1 represent the odd inputs and outputs.

Two polyphase decompositions of the filter will be required, one containing the even samples of the original filter, the other the odd:

$$\begin{aligned} h_0[k] &= h[2k] \\ h_1[k] &= h[2k+1], \end{aligned} \tag{6.11}$$

where $h[n]$ is the original filter, $k = 0, 1, \ldots, N/2-1$, and N is the length of the original filter.

The above equations give the following z-transforms:

$$\begin{aligned} X &= X_0 + X_1 z^{-1} \\ H &= H_0 + H_1 z^{-1} \\ Y &= Y_0 + Y_1 z^{-1}, \end{aligned} \tag{6.12}$$

which leads to the following two-parallel polyphase representation of the FIR filter:

$$\begin{aligned} Y &= X \cdot H \\ &= (X_0 + X_1 z^{-1})(H_0 + H_1 z^{-1}) \\ &= X_0 H_0 + (X_0 H_1 + X_1 H_0) z^{-1} + X_1 H_1 z^{-2} \\ Y_0 &= X_0 H_0 + X_1 H_1 z^{-2} \\ Y_1 &= X_0 X_1 + X_1 H_0. \end{aligned} \tag{6.13}$$

Equation (6.13) says that to double the throughput of the overall FIR filter $h[n]$, two of each of the length ${}^{N}\!/_{2}$ polyphase filters would be required as shown in Figure 6.13, resulting in an overall filter with twice as many taps as the original filter (four length ${}^{N}\!/_{2}$ filters).

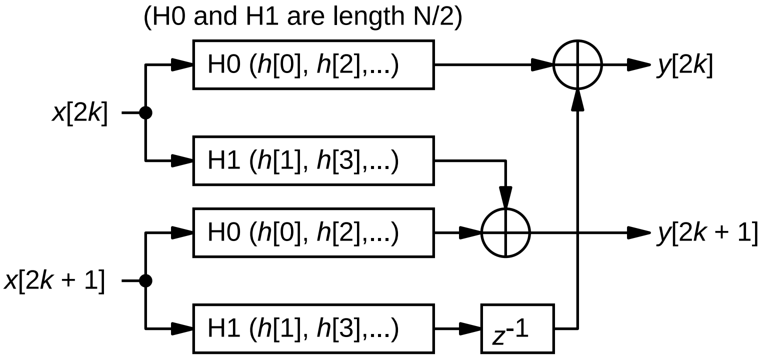


Figure 6.13: Traditional Two-Parallel FIR Filter Implementation

Two input samples are collected at a time and passed into the filter structure as illustrated in Figure 6.13, which produces two output samples. Each filter block shown in the figure is only running as fast as the original filter, however, so the throughput has been doubled. The FFA approach takes advantage of a rewriting of the polyphase equations derived from Equation (6.13):

$$\begin{aligned} Y &= X_0 H_0 + (X_0 H_1 + X_1 H_0) z^{-1} + X_1 H_1 z^{-2} \\ &= X_0 H_0 + [(X_0 + X_1)(H_0 + H_1) - X_0 H_0 - X_1 H_1] z^{-1} + X_1 H_1 z^{-2}, \end{aligned} \tag{6.14}$$

which implies that

$$\begin{aligned} Y_0 &= X_0 H_0 + X_1 H_1 z^{-2} \\ Y_1 &= (X_0 + X_1)(H_0 + H_1) - X_0 H_0 - X_1 H_1. \end{aligned} \tag{6.15}$$

The structure that implements Equation (6.15) is shown in Figure 6.14 for the same overall filter inputs and outputs. This filter only requires 1.5 times as many taps as the original, non-parallel, filter, although the coefficients for the middle FIR element in the figure must be pre-computed before being loaded into the FIR element. This is not an issue for most applications, as such a computation can be performed external to the filter.

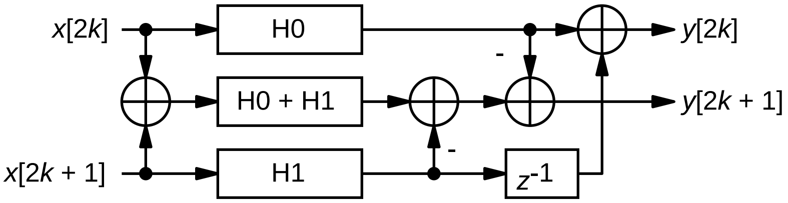


Figure 6.14: Two-Parallel FFA Implementation

Each of the three filter blocks shown in Figure 6.14 may have the FFA algorithm applied to it, resulting in the filter shown in Figure 6.15. This filter can process four inputs and outputs per clock cycle, with an area increase of 2.25 times the original filter size versus four times for a normal polyphase implementation, as the four-parallel FFA requires $^{9N}/_{16}$ taps instead of $16N$ taps. This process may be carried out recursively, increasing the throughput of the filter without a linear increase in area.

6.7 Frequency Domain Filtering

Instead of using convolution to calculate the output response for a FIR filter (as the techniques discussed above used), the filtering can be performed in the frequency domain. Convolution in the time domain is simply a multiplication

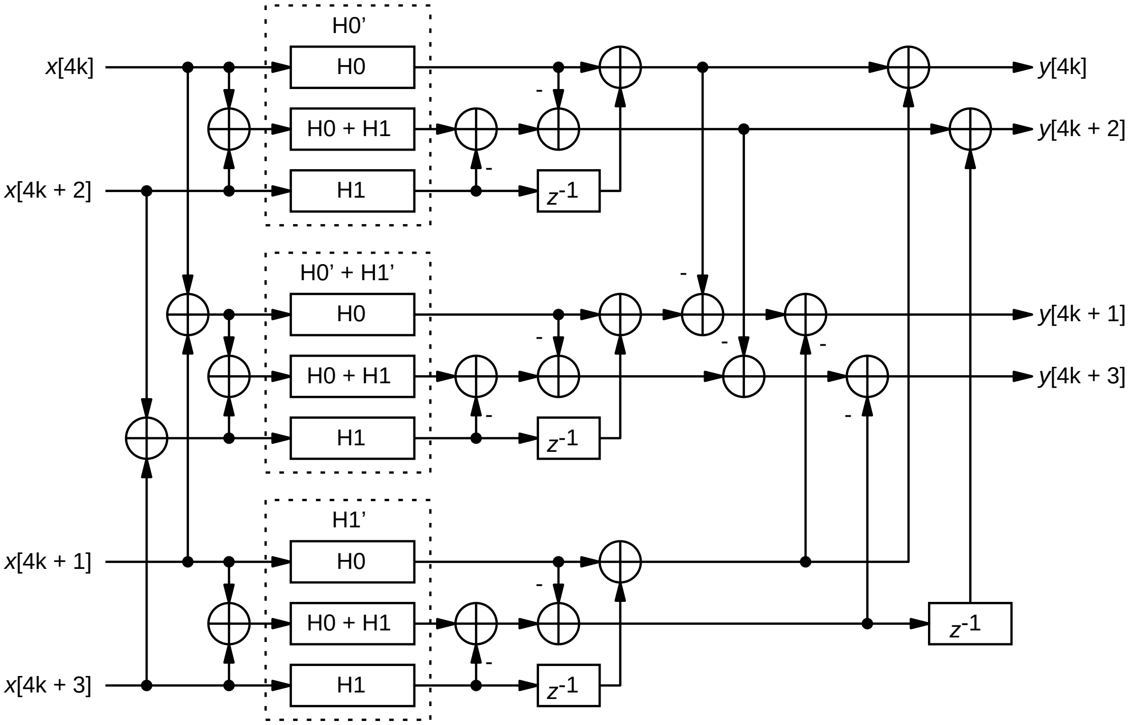


Figure 6.15: Four-Parallel FFA Implementation

operation in the frequency domain, so such an operation requires a transformation from the time domain into the frequency domain by a fast Fourier transform (FFT), a point multiplication of the input signal's spectrum by the filter's spectrum, and a transformation back into the time domain by an inverse fast Fourier transform (IFFT). The benefit of this technique is that it requires much less computational hardware than any of the approaches discussed so far using convolution. A FFT's computational requirements scales on the order of $\log_2 N$ versus N for convolution approaches.

The FFT is derived from the discrete Fourier transform (DFT), which is used to transform discrete time waveforms into discrete frequency spectrums [OS89, GW75]. The DFT is defined by

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \tag{6.16}$$

where

$$W_N^z = e^{-j\frac{2\pi z}{N}}, \tag{6.17}$$

$x[n]$ is a complex data sample at time n, $X[k]$ is a complex frequency sample at frequency $^k/_N$, and N is the number of frequency samples to calculate. W_N^z is sometimes referred to as a “twiddle factor.” The DFT requires on the order of N^2 computational requirements, so a more efficient method of computing the DFT is required. If N is an integer power of r, i.e. $N = r^v$, then an especially easy representation of the DFT appears, the radix-r FFT, where Equation (6.16) can be calculated iteratively in v stages:

$$a_{u,v}^m = \sum_{l=0}^{r-1} W_{r^m}^{-ul} a_{p,q}^{m-1}, \tag{6.18}$$

with

$$\begin{aligned} p &= u \bmod r^{m-1} \\ q &= v + lC_m \\ C_m &= \frac{N}{r^m} \\ a_{u,0}^0 &= x[n], \end{aligned} \tag{6.19}$$

$m = 1 \ldots M$, $u = 0 \ldots r^m - 1$, and $v = 0 \ldots C_m$ [GW75]. This is referred to as “decimation in frequency.” For $r = 2$, the algorithm is especially simple. At each stage, the algorithm passes through the entire array of N complex numbers, two at a time, generating a new array of N numbers. The basic numerical computation operates on a pair of numbers at a time, and is referred to as a "butterfly." The decimation in frequency FFT structure is shown in Figure 6.16 for $N = 8 = 2^3$, and a butterfly is shown in Figure 6.17. The numbers below each node in Figure 6.16 represent the z term in the twiddle factor W_N^z for that butterfly. A radix-4 FFT also exists, where four outputs are computed per butterfly for four inputs [OS89, GW75].

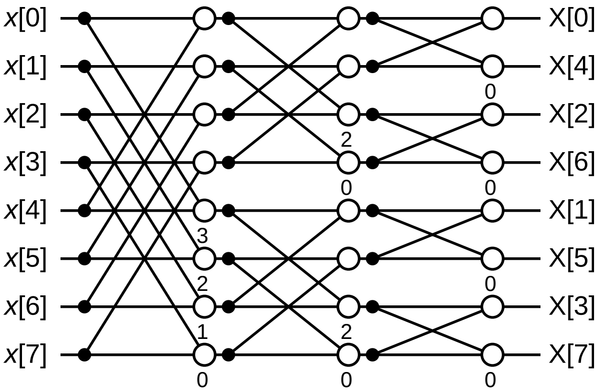


Figure 6.16: Eight-Point Decimation-In-Frequency FFT

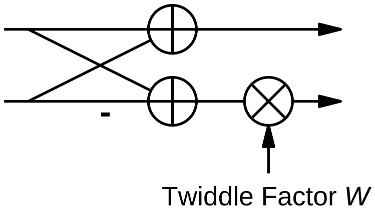


Figure 6.17: Decimation-In-Frequency Butterfly

6.7.1 Pipelined FFT

A nice feature of the FFT is that it can be easily pipelined by stage, as each stage needs only data from the proceeding stage. Each vertical grouping of butterflies in Figure 6.16 is referred to as a stage. Only one butterfly needs to be calculated in each stage at a time, although (to maximize the sample rate), each stage must have a butterfly calculated each clock cycle. Therefore, the

FFT can be built in a pipelined fashion, with each stage handled by a single pipelined butterfly.

Each pipelined butterfly needs shift register storage 2^m words long (where m is the stage number, with 0 being the right-most stage) to align its inputs and outputs correctly. For example, the first stage has butterflies that process inputs four samples apart in time. A shift register four words long is required to store the first four inputs, then output those four inputs to the top of that stage's butterfly as the next four inputs arrive at the bottom of the butterfly to compute the correct butterfly outputs. The top output of the butterfly are sent to the next stage while the bottom outputs are put into the shift register, which are then shifted out after the four butterflies have been computed. An example of a pipeline module is shown in Figure 6.18, and an eight-point pipelined FFT architecture is shown in Figure 6.19. [GW75]

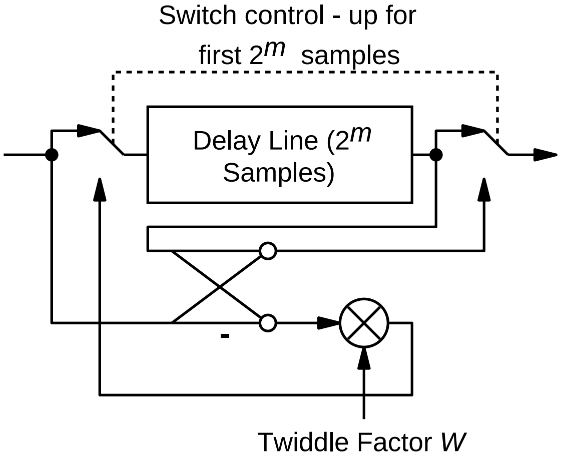


Figure 6.18: Pipelined FFT Butterfly Module

The twiddle factors can be arranged so that they may be sent to all of the modules from a common memory if they are retrieved at the correct time, reducing memory requirements. [GW75]

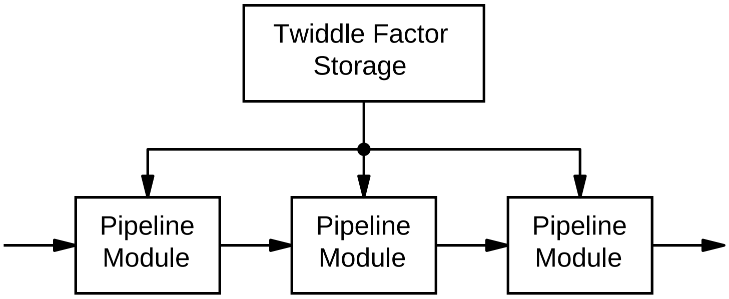


Figure 6.19: Eight-Point Pipelined FFT Architecture Block Diagram

6.7.2 FFT Convolution

Performing convolution with a FFT (i.e. transforming to the frequency domain, multiplying, and transforming back to the time domain) requires a FFT at least twice as large as the length of the filter to avoid time-aliasing in computing the DFT of the filter coefficients [OS89]. If a filter is N taps long, the FFT of the filter will have to be $2N$ points (with zero padding used to extend the N taps to $2N$ inputs for a $2N$-point FFT). The convolution is performed by retrieving a block of N inputs, performing a $2N$-point FFT (with zero-padding filling out the inputs) on them, multiplying them by the $2N$ filter frequency components previously transformed to the frequency domain by FFT, and performing an IFFT on the multiplication outputs. However, since only N inputs were taken and N outputs should be produced from the filter for a given block, and the $2N$-point IFFT produces $2N$ outputs, only the last N IFFT outputs should be used as filter outputs, as the first N IFFT outputs do not represent correct values of the convolution of the filter and the block of inputs [OS89]. This process is termed *overlap-save*.

6.7.3 Noise Analysis

A fixed-point FFT requires a considerable amount of rounding, as each stage has a multiplier that increases the stage's bit-length drastically. Rounding is required to reduce the stage's output to a manageable size. The noise analysis for a fixed-point FFT is complex, and is described in detail in [OW75, OS89] The important result is that each stage's butterfly's outputs require a scaling factor of $^1\!/_2$ to keep their adders from overflowing, which also reduces the final total amount of noise at the output. A MATLAB model of a fixed-point FFT with scaling at each butterfly was constructed to measure the final frequency domain filtering process's precision for given FFT, coefficient frequency spectrum, and IFFT word lengths.

Chapter 7

Implementation

7.1 Verification

To verify the functionality of the designs that were actually implemented below, a custom 512-tap FIR filter was constructed in C++ that could output full-precision fixed-point 2's complement results.[1] The output of this simulation was compared, bit-for-bit, with the output of each simulated FPGA design to ensure that each design was fully operational.

7.2 Parallel Multipliers and Accumulators

In the custom VLSI chip, each multiplier was re-used eight times per input sample, as shown in Figure 6.5. This meant that each tap would compute eight products and accumulate them together every input sample for eight separate coefficients. In this manner, 64 MACs were all that were needed to compute a 512-tap filter. However, each MAC needed to operate at a clock rate eight times that of the input data rate, so for a 5 MHz sample rate, a 40 MHz clock was required.

In an Xilinx Virtex series FPGA, a 16-bit by 18-bit parallel multiplier would require 163 slices (measured, not calculated). Therefore, the multipliers alone in a parallel MAC structure similar to Figure 6.5 would require 10,432 slices, which is near the Virtex XCV1000's capacity (12,288 slices). The addi-

[1] Courtesy of Michael Killoran, MIT Lincoln Laboratory

tional registers and control logic required for this design would push this number higher.

To decrease the area required by the MAC structure, the number of taps each MAC calculates could be increased. For example, each MAC could compute the product of 16 coefficients and the input word per input. This would halve the multipliers' area to 5,216 slices, but would double the clock rate to 80 MHz. The clock doubling could be handled internally by one of the Virtex Digital Locked Loops (DLL).

If each MAC was responsible for 16 taps, the coefficient storage for a single bank for a single MAC would require 9 slices since a slice contains two 16x1 RAM blocks, and one RAM block could hold a single bit for all 16 taps. 18-bit coefficients would therefore require 18 RAM blocks which can be contained within 9 slices. Two coefficient banks are required by the design specifications, so 18 slices are needed per MAC for coefficient storage. With 32 MACs, this means that 576 slices will be required for coefficient storage.

A 32-MAC, 512-tap filter was built using the architecture shown in Figure 6.5. The details of an individual MAC are shown in Figure 7.1. Double banking has been handled by two coefficient storage RAMs per MAC. The active bank's coefficient RAM is addressed by a looping counter that selects the coefficient to be multiplied by the *X* value output from the shift register. The loadable bank's coefficient RAM is addressed by the current coefficient load address, is write-select enabled, and is clocked by a coefficient load signal so that new coefficients may be loaded into it while the active bank is operating. A pipelined parallel multiplier produces the product of the current *X* value and the current coefficient. A rounder unit reduces the multiplier's output bit width (which is the size of the input plus the size of the coefficient) to a smaller width. As this architecture is the same as that used in the custom VLSI design, the MAC output width was chosen to be 24-bits like it was in the custom design to maintain the same precision.

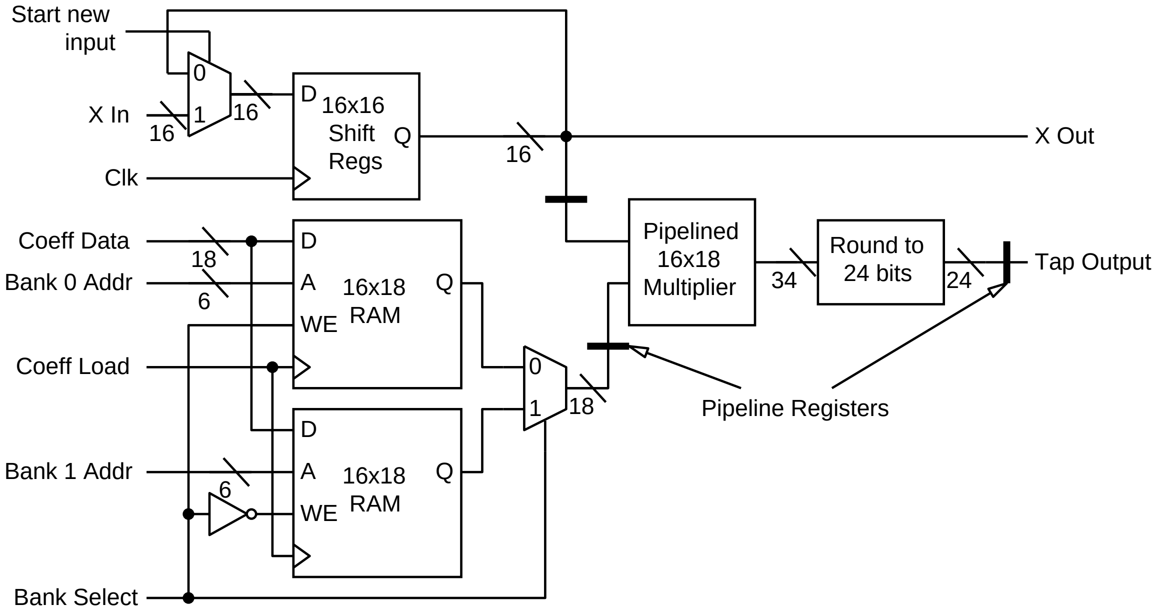


Figure 7.1: 16-Tap Parallel MAC Unit

The full, operational, 32-MAC filter (with accumulator, summers, and glue logic) required 7,816 slices.

7.3 Bit-Level Systolic Array

In the structure shown in Figure 6.6, each cell's output is required to be registered in order to pipeline the array. The three outputs from each main cell each require a register, so the cell requires at least one and a half CLBs. Each main cell also must compute two functions of four inputs (each of which can fit into a Virtex LUT) and needs to store one bit of a coefficient (without dual-banking as required by the design specifications). A CLB LUT may be used to store this bit. Therefore, each main cell requires three registers and three LUTs. Two main cells may be contained within three slices, which would contain six LUTs and six registers, without dual-banking.

According to Wang [WWC88], the number of main cells required for the systolic array is $(B + L) \cdot N$, where B = the number of bits in the input, N = the number of taps, and $L = \log_2 N$. For a 512-tap, 18-bit filter, the number of main cells required for the systolic array is 13,824. This would require 20,736 slices to implement. Since this structure is bit-level pipelined, one output is produced every B clock cycles, or every 16 clock cycles for the 16-bit design. Therefore, for a 5 MHz input and output sample rate, the array will have to operate at 80 MHz, which is above the design requirements, although this may be handled by the Virtex DLL circuitry. In addition to the main array, there are other cells required for the design that would also increase the area.

This architecture was used for a full-custom, transistor mask-level design, 1024-tap FIR filter that could operate with a clock rate up to 640 MHz, and a sample rate up to 20 MHz with a 0.6 μm fabrication process. The reason that this design was so efficient for a full-custom chip versus a FPGA is that it requires a very fine-grained architecture. The basic cell in the bit-level systolic array is only three registers and a few logic gates, which takes up very little area on a custom chip. However, for a coarse-grained FPGA such as the Virtex, the simplicity of a single cell is actually a drawback, area-wise. A better approach for a FPGA design is to use cell sizes that more appropriately map into the FPGA’s architecture, such as those used in distributed arithmetic.

7.4 Distributed Arithmetic

The distributed arithmetic approach as presented in Section 6.5 can easily be expanded to 512-taps, but two problems remain. First of all, the DA coefficients are constant, but the design requirements demand two swapable, loadable coefficient banks. Second, a SDA filter requires *B* clock cycles to process a single input sample. For a 5 MHz input, this means a 16-tap filter must run at 80 MHz, which is twice as fast as the design specifications allow.

7.4.1 Distributed Arithmetic Four-Tap Group

To solve the first problem, two banks of LUTs are used for each four-tap group. One bank is used as the active LUT—this is the LUT that is addressed by the shift-register outputs—and one bank is loadable by the user. Therefore, the user can load all of the LUT values into one bank of the FIR filter in the background while the old LUT values stored in the other bank are active. By toggling a bank select line, the two banks are switched so that the previously loadable bank is now active, and the previously active bank is now loadable.

Figure 7.2 shows the complete block diagram for a single four-tap group including the shift registers and the double-banking LUTs as described above. The four-tap group accepts a serial data input (from the last tap's shift register of the previous group), and produces a serial data output for the next group's first tap's shift register. A bank selection line selects (through multiplexors) which bank of LUTs are active, and which are used for loading new coefficients. Each bank is 20-bits long due to two-bit word growth in computing the LUT contents. The active bank is addressed by the four shift register outputs. The bank's output is the group's output. The loadable bank is addressed by an external set of coefficient address lines that select which of the 16 bank addresses is being written. A group coefficient load enable line selects whether this group is to have its coefficients updated versus another group, and a bank write clock line writes the data into the correct bank (the bank being loaded). A separate clock was used for each bank's LUT write clock to minimize the amount of logic local to a group.

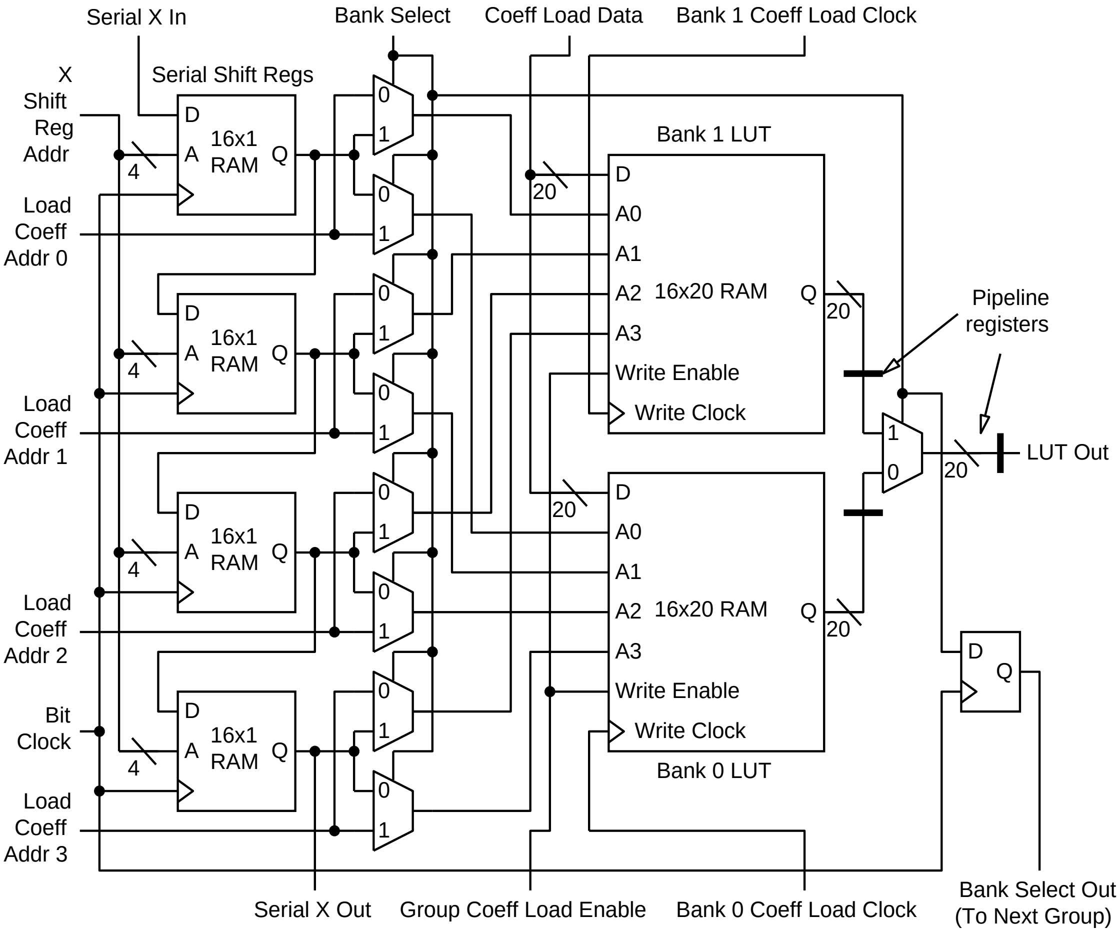


Figure 7.2: Four-Tap Group

Pipelining was inserted in the four-tap group so that the combinational delay between pipeline registers has been kept to a minimum to increase performance. In addition, as shown, the bank selection line has been pipelined between four-tap groups. This prevents a single bank selection line from having to drive all the multiplexors in every four-tap group, which would lead to a very high fan-out and a slow signal, decreasing the overall system performance. One drawback is that changing coefficient banks will take 128 clock cycles during which the new bank selection signal is propagated through its pipelining registers. Any outputs produced during that time will consist of outputs from both coefficient banks, and will be incorrect responses from either bank's filter. This drawback is addressed below with the linear-network summer tree. In addition, coefficients in a given four-tap's stand-by registers cannot be altered after a bank selection switch until the new bank selection signal has propagated to that four-tap, or else the wrong bank would be updated.

A single four-tap group requires 36 slices.

A 512-tap filter using an expanded version of Figure 6.11 with the four-tap group in Figure 7.2 was built. This filter was designed with full-precision, meaning that every adder stage grew by one bit (so no rounding was required), and the scaling accumulator was large enough to hold the entire 43-bit result. This 512-tap filter required 6,203 slices.

7.4.2 Linear Summer Network

The summer tree used to create the full 512-tap partial product is a large bottleneck in the design discussed above. The large summer tree requires long routing lengths to provide the inputs for the last few adders, and is not geometrically easy to fit into a FPGA without wasting area or introducing even longer wire lengths. A linear design was created, where each four-tap group has a summer associated with it that adds the previous four-tap group's output to its own output. This sum is pipelined, and sent onto the next tap. Due to the added stage of pipelining between the summers at each four-tap group, a stage of pipelining must be inserted between each group's serially cascaded input value. This will keep the outputs and inputs correctly synchronized. The linear summer technique is illustrated in Figure 7.3.

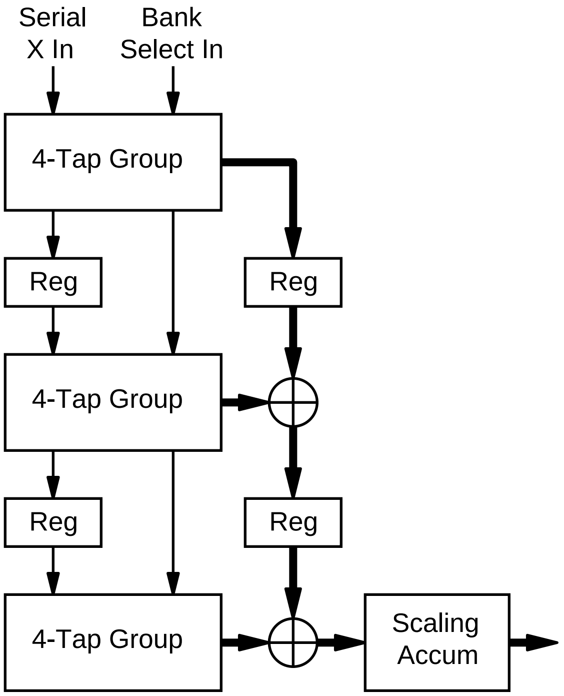


Figure 7.3: Linear Summer Network for SDA

With the linear technique applied, the design is more efficient as each four-tap group has one summer attached to it that needs to communicate with only adjacent groups, so all of the groups may be stacked together. There are no long routing lengths required in this design like there are in the summer tree technique. To minimize wasted area, each summer is only as many bits long as required to protect against overflow. For example, the summer for the second four-tap group need only be 21 bits long because it is adding the 20-bit result of the first group to the 20-bit result of the second group. The third group's summer needs to be 21-bits long, as it is adding three 20-bit results.

The forth group's summer needs to be 22-bits long, as it is adding four 20-bit results. The number of extra bits per summer can be found by taking the integer portion of $\log_2 i$, where i is the four-tap group's number.

A benefit of the combination of the linear network and pipelining the bank selection line as discussed Section 7.4.1 is that, upon the execution of a bank switch (inverting the bank selection signal), the four-tap groups sequentially switch their coefficient banks from stand-by to active each clock cycle. Outputs being formed by the four-tap groups and being passed along through the linear summer network before the banks were switched will continue to have partial products generated using the old coefficients added to them as they move down the summer network's pipelining chain. Since the outputs and the bank selection signal propagate through the network at the same rate, the first output after the bank selection switch will only have partial products generated with the new coefficients added to it. The coefficient banks in a given four-tap group will swap at the same time this output enters the group, resulting in the correct partial product being summed to the output by the group. This means that no incorrect data will be generated during a bank switch; rather, the filter's output will seamlessly change from one filter response to the next.

This technique has two small drawbacks. First of all, it is slightly larger than the summer tree technique. The 512-tap SDA linear adder tree required 6,378 slices. Although the number of adders is the same for both techniques, the adder tree requires less adders as the adder bit size grows, whereas the linear network requires more. For example, the adder tree only requires two 24-bit adders, whereas the linear network requires 63 24-bit adders. However, as long as the design still fits within a Virtex device, this is acceptable. The second drawback is that the output has a latency of 128 clock cycles due to the pipelining of each four-tap group's output. In most signal processing applications, small latencies such as this are not detrimental.

7.4.3 Achieving 40 MHz Performance For 5 MHz Data Rate

The serial distributed arithmetic design as described above requires a clock rate 16 times faster than the input data rate. For a 5 MHz data rate, this means the serial filters must run at 80 MHz. The design specifications require a clock eight times faster than the input rate, or 40 MHz. Two solutions exist to solve this problem. The first is to operate the internal SDA filter at 80 MHz, using the Virtex DLL to multiply the external 40 MHz clock up to a 80 MHz internal clock rate. The second is to use 2-bit PDA, with two 512-tap SDA filters placed on the chip as shown in Figure 6.12. A clock rate eight times faster than the sample rate would be required for this design, and the internal filters would only have to operate at 40 MHz, yet the design would require twice as much area as a SDA design. The trade-off between the two techniques is speed versus area. The DLL design requires the internal filter to work twice as fast as the 2-bit PDA design, whereas the 2-bit PDA design requires twice as much area. Unfortunately, the 2-bit PDA design requires 12,756 slices for the 512-tap linear-network filter alone (without the scaling accumulator and additional glue logic), which is larger than the XCV1000. For this reason, the DLL SDA design was chosen.

7.5 Fast FIR Algorithm

A new, unique application of the FFA algorithm was developed and analyzed for the filtering problem in this thesis, although it is not limited to FPGA implementations.

The parallel MAC approach to the filtering problem described in Section 6.3 can be combined with the FFA algorithm in Section 6.6 to derive small filtering structures. In the parallel MAC approach implementation (see Section 7.2), 64 MACs were used to calculate a 512-tap FIR response by having each MAC perform eight multiplications per input word, so that each MAC handled eight filter taps. This could be increased to 16 taps per MAC, so that 32 MACs would be needed with an 80 MHz clock rate (provided the external clock is multiplied to 80 MHz from 40 MHz by the DLL).

If each MAC was, in turn, now responsible for 32 taps, only 16 MACs would be required, but a 160 MHz clock would be needed. The new FFA approach would take the polyphase decomposition of the filter as described in Section 6.6. Each polyphase filter (i.e. H_0 or H_1) would be half the size of the original 512-tap filter, or 256-taps. Applying the FFA algorithm would require three 256-tap filters as shown in Figure 6.14. Each of these smaller filters would now have to run at only half the original filter's sample rate, or 2.5 MHz to maintain an overall sample rate of 5 MHz (the FFA approach allows an overall throughput twice that of the individual filters' sample rates). Each polyphase filter could be implemented with eight MACs, where each MAC handled 32 taps. The benefit is that the sample rate of each filter is now only 2.5 MHz, meaning that each MAC would only have to run at 80 MHz to compute the results from 32 taps per input. With no change in clock rate, the total number of MACs would be decreased to 24 from the original 32.

One major change is that the middle filter ($H_0 + H_1$ in Figure 11) would be multiplying 17-bit inputs by 19-bit coefficients because of the one-bit word growth through the addition of $x[2k] + x[2k + 1]$ before the filter and the one-bit word growth in the addition of $h_0[k] + h_1[k]$ to compute the filter's coefficients. Therefore, the middle polyphase filter would be slightly larger than the other two filters. The overall FFA filter will also be slightly larger due to the five extra summers and the delay element, but this extra logic is insignificant when compared to a single MAC. Finally, the size of each MAC will grow as each MAC will have to have registers to store 32 coefficients and inputs instead of 16.

If each MAC was responsible for 64 taps (doubling the clock rate to 160 MHz again), and the FFA algorithm was applied again (i.e., a 4-parallel FFA implementation as in Figure 6.15), the new filter would require 9 polyphase filters of length 128 taps. Each polyphase filter would now be comprised of two MACs, and would need to only operate at 1.25 MHz, resulting in an overall frequency rate of 80 MHz. The total number of MACs would now be 18 instead of 24 or 32. Again, however, several of the polyphase filters will have to handle larger inputs and coefficients (specifically, one will multiply 18-bit inputs by 20-bit coefficients, and four will multiply 17-bit inputs by 19-bit coefficients), more adders will be required, and each MAC will have to store 64 coefficients and inputs, so the net gain in area will be smaller than it appears by simply looking at the number of MACs saved.

Each individual polyphase filter is similar in design to the filter shown in Figure 6.5. The 64-tap MAC used in the 4-parallel FFA is similar to the one shown in Figure 7.1, except that the input shift register and the coefficient banks are each 64 words long. The output of each MAC was chosen to be 24-bits as it was in the 32-MAC filter in Section 7.2. All of the summations from the output of each MAC to the output of the overall filter were sized to maintain full precision. Because of the larger input and coefficient bit widths in several of the filters, however, it was unknown whether having 24-bit MAC outputs would maintain the same numerical precision for the overall filter as for the custom VLSI chip. To verify this, a MATLAB simulation of the FFA filter was constructed with fixed-point arithmetic and 24-bit rounding at the output of each MAC filter. This simulation showed that the FFA approach with 24-bit MAC outputs created a final FIR filter output with 17.7 bit precision, slightly less than the custom VLSI approach. This precision was deemed acceptable, as increasing the bit-width of each MAC output would drastically increase the filter size.

A 16-bit input, 18-bit coefficient MAC in the FFA filter required 163 slices. The overall size of the FFA filter was 6,900 slices. One drawback to this design is that the overall filter is not regular or systolic like the distributed arithmetic design. It also has a large number of high fan-out signals (the active coefficient RAM bank address lines and the *start_new_input* control signal). However, only the logic internal to each MAC is required to run at the full 80 MHz internal clock rate. This logic is regular (as Figure 6.5 shows), can be placed in a somewhat systolic fashion (although there are still a number of high fan-out signals), and is highly pipelinable. The FFA summer network must only run at $\frac{1}{4}$ the *sample rate* (1.25 MHz), as this network calculates, in parallel, four filter outputs simultaneously and needs to produce its outputs only once every four sample clocks.

Therefore, while the overall filter cannot be placed as easily within a FPGA as the distributed arithmetic design while trying to minimize route lengths, the parts of the filter that do need to run fast can be placed efficiently. The main performance bottleneck is being able to drive the high fan-out signals that connect to every MAC fast enough to meet the performance of the internal polyphase filter logic.

7.6 Frequency Domain Filtering

For the frequency domain filtering implementation, a 1024-point FFT and IFFT are required (see Section 6.7.2). However, since the input data is real, both the real and imaginary parts of the FFT may be used to hold the real input data [OW75, OS89]. Therefore, a 1024-point FFT can be calculated with a 512-point FFT structure. A radix-2 FFT and IFFT were chosen as a radix-4 FFT, which is easy to implement [OS89], would require a 1024-point FFT structure, and would be bigger than a 512-point radix-2 FFT.

The next step for the frequency domain filtering implementation was to determine the bit-size required for the FFTs in order to meet the custom VLSI chip's output precision. A MATLAB simulation of filter using a fixed-point radix-2 FFT and IFFT was created and run with different values for the FFT bit-width, filter frequency spectrum bit-width, point multiplication output rounding width, and IFFT bit-width. The minimum bit-width to maintain approximately 18 bits of output precision was determined for each variable separately, and is shown in Table 7.1.

Parameter	Bit-Width	Output Precision
FFT bit-width	30	17.704
	31	18.802
Filter spectrum bit-width	28	17.973
	29	19.589
Point multiplication rounded output width	34	17.487
	35	18.465
IFFT bit-width	35	17.536
	36	18.512

Table 7.1: Frequency Domain Filtering Output Precision

The bold bit-widths in Table 7.1 were used together in a final simulation of the frequency domain filtering technique. The output precision was measured to be 17.881 bits, which was determined to be closed enough to the custom VLSI output precision, as adding a bit to any of the parameters above would drastically increase the technique's area.

With the bit-widths above, a preliminary area calculation was made for multipliers and memory. Using the equation in Section 5.3, the 31-bit x 31-bit multipliers in the FFT would require about 542 slices. The point multiplication would be multiplying the FFT's 31-bit result by a 29-bit filter spectrum value, which would require about 512 slices. The IFFT multipliers would be multiplying 36-bit numbers by 36-bit numbers, and would require about 723 slices.

Assuming a 512-point FFT was being used as described above, the FFT and IFFT would each have 9 stages, so 18 butterfly pipeline stages would be required total. With four multipliers per stage to compute each stage's complex twiddle factor multiplication and four multipliers for the complex point multiply, the multipliers would require 47,588 slices. Since the multipliers are full-parallel, the clock rate for this implementation would be equal to the sample rate.

If the clock rate was quadrupled and each multiplier was reused four times per input sample (i.e. one multiplier per complex multiply), the multipliers would require 11,897 slices. Reusing the multipliers any more times would be difficult, as a single multiplier would have to perform the multiplications for multiple pipeline stages of the FFT or IFFT. Assuming that this was easy to implement and the clock rate was made to be 16 times the sample rate (as it was in the FFA and DA implementations), the multipliers would now require 2,974 slices.

Each pipeline stage requires 2^m words of storage for its delay line (see figure Figure 6.18), where *m* is the stage number. This means that 511 complex words of storage are required for the FFT and 511 complex words are required for the IFFT. 256 complex twiddle factors are required for each FFT, but may be shared between the FFT and IFFT. Two banks of 1,024 complex words are required for the frequency spectrum storage so that one bank may be loaded while the other is active. Each complex word requires two RAM words of storage. With 31-bit words for the FFT delay storage, 29-bit words for the frequency spectrum storage, 36-bit words for the IFFT delay storage, and 36-bit words for the twiddle factor memory (to accommodate the 36-bit IFFT requirement), the total filter requires 205,690 bits of memory storage.

One advantage of this implementation is that much of the memory storage consists of large blocks of RAM where only a single location needs to be accessed at a time. This means that the Virtex block RAM could be used for this application. The Virtex XCV1000 has 131,072 bits of block RAM. Assuming that all of this memory could be used, 74,618 bits would remain and could be accommodated by the distributed RAM within the CLBs. 74,618 bits of RAM would require 2,332 slices.

Therefore, the frequency multiplication technique for a 512-tap filter would require 5,306 slices for memory and multipliers *alone*. The control logic for this implementation is complex and would be area-intensive, especially with a single multiplier being re-used 16 times per input sample. In addition, adders, pipeline registers, twiddle factor distribution logic, and the control logic to implement the overlap-save method necessary to filter the continuous input stream would increase the area dramatically. As a reference, in the distributed arithmetic design, which had much simpler control and datapath logic, the four-tap groups required 4,608 slices out of 6,378 slices, leaving 1,770 slices for control and summation. Even with only 1,500 slices of control and datapath logic, the FFT algorithm would require more slices than the distributed arithmetic approach and about the same as the FFA algorithm.

Although the pipelined FFT algorithm is (hence the name) highly pipelinable, a design reusing a single multiplier sixteen times would not be regular and would require long route lengths due to the complex nature of its control, reducing its performance compared to the other, more regular designs discussed earlier. Therefore, the FFT algorithm was ruled as being larger and slower than the DA or FFA designs, and was not implemented.

One point to note with the FFT algorithm is that moving from a 512-tap filter to a 1024-tap filter would require a smaller area increase than moving from a 512-tap filter to a 1024-tap filter using a FIR approach, as such a move requires the addition of one stage to the FFT and IFFT and twice as much twiddle factor and spectrum storage memory, whereas the FIR techniques would require a doubling of area. The FFT algorithm's area benefits would become even more obvious as the filter's size increased further, as each doubling of taps requires a doubling of area but a small increase in frequency spectrum filtering area.

Chapter 8

CELL_MAKER Custom Layout Tool

8.1 Motivation

One of the benefits of using VHDL to describe a gate-level design such as a FPGA or custom VLSI design is the ability to create parameterizable modules that can be instantiated in higher levels of the design hierarchy. For example, a delay line could be created in VHDL with a variable number of bits and a variable number of delay stages. When this delay line module is instantiated, the exact sizing of a particular instance is specified at compile time in the VHDL as part of the instantiation code. It is also often easier to write, simulate, debug, and modify VHDL than schematics. VHDL also gives the user the ability to write high-level, portable code that can be synthesized into a particular architecture, although for reasons discussed below, this benefit of VHDL is not being used for the designs in this thesis.

The problem with using VHDL for FPGA designs is that there is no way presently (with the Xilinx Foundation software using Synopsys FPGA Express for synthesis) to communicate the desired layout of a VHDL design to the placement software. In the designs discussed in Chapter 7, a single small cell was often replicated many times in a systolic fashion. For example, in the DA approach, the 4-tap group was replicated 128 times. Each group only connected to the two adjacent groups, creating a linear systolic network. Because of this, care and time could be taken to place the components that make up one group, and then this placement could be replicated to all the instantiations of the 4-tap group. Ensuring that consecutive groups were placed adjacent to each other would result in the most efficient design.

Unfortunately, VHDL has no method of describing this process, and the placement tools do not have the knowledge that the design was created in a systolic fashion. As a result, in the linear DA design for example, the components from different four-tap groups and the rest of the logic were interspersed among each other in a seemingly haphazard fashion. Many signals that should have been short if the systolic approach was taken ended up long, as components that should have been near each other were far apart on the chip. For example, the connection from the last four-tap group to the scaling accumulator spanned the entire width of the chip, as the two cells were not placed next to each other as they should be for maximum performance. The delay along this route was 6 ns alone. Constraining the design's placement would drastically improve its performance.

Although some tools (e.g. Synopsys) allow attributes (such as placement constraints) to be passed from VHDL to the synthesized netlist for instantiated components, this feature is very limited. Components created as part of a generate statement (for example, a bank of registers x-bits long could be created by using a generate statement to duplicate a single register x times) are created during the synthesis process, and cannot have attributes attached to them. In addition, writing a long string of attribute statements can be tedious and error-prone.

A tool, CELL_MAKER, was developed for this thesis to read in VHDL code, extract basic placement constraints added as comments by the user, and create a user constraints (UCF) file describing the placement constraints for every component in the VHDL code. This tool allowed a cell that is replicated many times to be placed once, and the replication strategy (e.g. linear systolic) to be specified in order to create the best placement. The only limitation is that instantiated library components from the Xilinx FPGA library (e.g. registers, RAMs, adder primitives, etc.) must be used within the VHDL instead of high-level, synthesizable code (e.g. using the + operator for addition). The reason is that the tool cannot infer the components that would be generated by high-level code. All the components must be explicitly instantiated so that placement constraints can be attached to them unless the synthesizable code can be placed within a single CLB or slice. A special provision for that situation has been provided within CELL_MAKER.

8.2 Operation

Placement constraints are specified as comments in the VHDL code for CELL_MAKER. A placement constraint comment is placed after the component name and before the generic map or port map statement in an instantiated component. A constrained instance takes the following form, where the constraints are shown in bold (anything after a “--” in VHDL is considered a comment):

```
<instance> : <component>
        -- cell_const <constraint_type> <constraint>
        [ -- cell_const param <parameter> => <cm_expression> ]
        [ -- (more param statements...) ]
    generic map (
    ...)
    port map (
    ...);
```

The code above shows the two statements that are available when constraining an instance. The constraint_type statement must always be first and can only occur once. This command constrains the placement of the instance, and is described in Section 8.2.1. The second statement is optional and may occur several times. This is used for passing parameters into the instantiated component, and is described in Section 8.2.2.

8.2.1 RLOC Placement Constraints

The basic form of a placement constraint is the same as the Xilinx LOC constraint, which takes the form (for Virtex): “R <row> C <column> [S <slice>],” where <row>, <column>, and <slice> are replaced with “cm_expressions.” This new form of an expression is described in Section 8.2.4. Rows are numbered from top to bottom within the chip, and columns are numbered from left to right. The left slice in a CLB is slice 1, and the right slice is slice 0.

Three constraint types are available: rloc, rloc_clb, and rloc_clb_all. Although these constraints use the word “rloc,” they will ultimately be turned into Xilinx LOC constraints by the custom tool, not Xilinx RLOC constraints. Rloc_clb is intended to be used on instantiated FPGA components such as flip-flops or adder carry chain multiplexors that can be located within a specific CLB or slice. The other two constraints are used on instantiated VHDL components, i.e. components that are written in VHDL and have an entity and architecture that will be synthesized into the design and contain more VHDL components or instantiated FPGA components.

The rloc constraint is used on a VHDL component with instantiated VHDL or FPGA components within it, and is of the form “R <row> C <column>” (i.e. no slice is specified). When the custom tool encounters a rloc constraint on a component, it adds the row and column numbers specified in the rloc constraint to all of the row and column numbers in constraints within the component. For example, if a component has a particular instance located at row 4, column 3, and that component has been instantiated in a higher level of the VHDL hierarchy with a constraint of row 10, column 20, the instance within the component will now be placed at row 14, column 23. In this manner, the relative locations of components within a VHDL entity can be specified relative to row 0, column 0 (e.g. row 4, column 3 in our example). When that entity is instantiated in a higher level of hierarchy, the entire entity can be located as if it were one cell at a particular location with its row 0, column 0 located wherever the constraint on its instantiation specifies it should be located (row 10, column 20 in this example). All of the components in the entity will be located relative to that location as they are described within the entity as if that location were row 0, column 0.

The final placement locations for individual FPGA components is determined by tracing through the VHDL hierarchy from the top-most entity down to the individual FPGA components, adding the rloc row and column values for each instantiated component as the program progresses through the hierarchy.

Rloc_clb_all is a special constraint that is used only on VHDL components that contain logic or FPGA components that can fit within a single CLB or slice, and may contain a slice number. This constraint specifies that all of the logic within the component should be placed within the specified CLB or slice. This is useful when it is easier to describe a logic function with VHDL commands (for example, a multiplexor or boolean logic) than with instantiated components, but the logic needs to be located within a particular CLB or slice. No location constraints should be used within the component's VHDL code as the tool will not look at them. In addition a special flag must be included on any entity that will have rloc_clb_all constraints attached to it as described below.

8.2.2 Parameter Passing

CELL_MAKER allows constants from an entity's generic block to be used within a constraint's expression. If there are parameters that are only to be used by constraints and not within the VHDL code itself that need to be passed into an entity, they may be passed along to an instantiated component with "cell_const param" statements. The parameters that may be passed to an entity are described in its entity declaration as shown in the following code:

```
entity <entity_name> is
        -- [cell_const paramdef <parameter> [ :=
    <default_cm_expression> ] ]
        -- [ (more paramdef statements... ) ]
        -- [ cell_const flag clb_all_entity]
    generic (
    );
    map (
    );
end <entity_name>;
```

The declaration of a parameter allows the parameter to have a default value if it is not given a value when it is instantiated. If a parameter has a default value, and is given a value during instantiation, the default value will be ignored. If it does not have a default value, it will require a value upon instantiation. The default value may use any parameters declared earlier within the entity block in its expression.

The clb_all_entity flag statement must be included if this entity is to have a rloc_clb_all constraint attached to it instead of the normal rloc constraint attached to VHDL component. This is purely to ensure the user attaches the correct constraint and gets the correct behavior out of the system. An entity with this flag would have no parameters, as it cannot have any constraints within its code that could use the parameters.

8.2.3 Generate statements

One of the driving motivations for CELL_MAKER was the desire to place large, systolic, designs efficiently. The tool facilitates this by allowing the range variable within a “for” generate statement to be used within constraints or expressions passed as parameters into instantiated components. The example below illustrates how this can be used to replicate a component easily while constraining its location.

```
reg_block : for i in 0 to 15 generate
    reg_instance : register_component
            -- cell_const rloc_clb R -i/2 C 0 S 1
        port map ( ... );
end generate;
```

In this example, two registers are placed in each CLB, with the reg_instance corresponding to i = 0 placed in row 0, column 0, slice 1, and the reg_instance corresponding to i = 15 placed in row -7, column 0, slice 1. This register bank is therefore a vertical bank of registers with the least significant bit at the bottom of the bank.

One work-around for Synopsys FPGA Express had to be made in order to get this tool to work correctly. Normally, when a component is instantiated within a generate statement, FPGA Express gives each replicated instance a name of the type “<instance_name>_<i>,” where instance_name is the name of the instance in the VHDL code, and i is the generate statement’s range variable’s value for that instance. However, when the instantiated component is an entity written in VHDL with generic constants, this naming scheme changes to a much less intuitive system that was uncovered through experimentation, and is beyond the scope of this paper. However, CELL_MAKER was modified to compensate for this special condition.

8.2.4 Cm_expressions and Conditional Constraints

Two more features were built into CELL_MAKER to make constraining components easier. The first was the definition of a “cm_expression.” This expression can use the following VHDL operators with normal VHDL precedence to compute a value:

```
(, ), -, +, *, / (integer division), MOD.
```

In addition to normal equations, a cm_expression may take the following form:

```
<expr1> when <expr2> else <cm_expression>
```

In this form, expr1 is used as the expression’s value when expr2 evaluates to true. If expr2 is false, then cm_expression is evaluated and used as the expression’s value (the “else” cm_expression may contain more “when” statements). Expr2 may use the following VHDL operators:

```
and, or, not, =, /= (not equals), <, >, <=, >=.
```

Finally, CELL_MAKER has been given the ability to include rloc_clb constraints conditionally. The usage of this feature is:

```
-- cell_const rloc_clb (R <row> C <col> [S <slice>]) when <expr>
```

If expr evaluates as true, then the constraint is attached to the instance, otherwise it is not and the instance will be unconstrained.

8.2.5 Output

The final output of CELL_MAKER is a Xilinx user constraints file (UCF) file, with lines of the form:

```
INST <instance_hierarchy_name> LOC = CLB_<final_location>;
```

A line is included for each instantiated component that has no VHDL code (i.e. a FPGA component) and a rloc_clb constraint that evaluates true if it is a conditional constraint (conditional constraints that evaluate false are still included in the file, but are commented out), and for each block of logic and components in a VHDL entity with a rloc_clb_all constraint attached to the entity.

CELL_MAKER can also include a user-created UCF file at the beginning of its generated UCF file if the user wishes to add additional timing constraints, pin-locking constraints, or manually-entered placement constraints.

8.2.6 CELL_MAKER Example

A small example with three VHDL entities is shown below to demonstrate CELL_MAKER's functionality (this example is to show CELL_MAKER's usage, and would result in a bizarre, non-ideal layout for a real design):

```
entity example1 is
        -- cell_const paramdef clb_slice
        -- cell_const paramdef def_ex:= 1
    generic ( X : in integer) ;
    port ( ... );
end example1;

architecture rtl of example1 is
begin
    ex_block : for i in 0 to X - 1 generate
        ex_inst : FPGA_component
                -- cell_const rloc_clb R -i*2 C 0 when i<1 else 3 S clb_slice
            port map ( ... );
    end generate;
    ex_cond_inst : another_FPGA_component
            -- cell_const rloc_clb (R 0 C 1 S 0) when def_ex + X = 2
        port map (... );
end rtl;

entity example2 is
        -- cell_const flag clb_all_entity
    port (
        signal a, b : in std_logic;
        signal c : out std_logic);
end example2;

architecture rtl of example2 is
begin
    c <= a and b;
end rtl;

entity example3 is
        -- cell_const paramdef top_param := 30
    port ( ... );
end example3;

architecture rtl of example3 is
begin
    example1_inst : example1
            -- cell_const rloc R 10 C top_param
            -- cell_const param clb_slice => 1
        generic map ( X => 2 )
        port map ( ... );

    example2_inst : example2
            -- cell_const rloc_clb_all R 20 C 40 S 0
        port map ( a => a, b => b, c => c );
end rtl;
```

The UCF file output from CELL_MAKER with example3 the top-most entity is:

```
INST example1_inst/ex_inst_0 LOC=CLB_R10C30.S1;
INST example1_inst/ex_inst_1 LOC=CLB_R8C33.S1;
# COND: INST example1_inst/ex_cond_inst LOC=CLB_R10C31.S0;
INST example2_inst/* LOC=CLB_R20C40.S0;
```

After the VHDL code has been synthesized with FPGA Express, it will be turned into a EDIF file that contains a net-list of the design, library symbols representing the bottom-most instantiated components within the VHDL hierarchy (the FPGA components), and library symbols representing high-level VHDL statements. This EDIF file along with the UCF file created by CELL_MAKER are then passed into the Xilinx tools to create a FPGA with functionality and placement described by the user. The "# COND" in line 3 indicates that the constraint on ex_cond_inst within example1_inst was a conditional constraint that evaluated as false (# is a comment character in a UCF file). If it was evaluated as true, there would be no "# COND." Line 4 indicates that all logic and components within example2_inst should be located in slice R20C40.S0.

8.3 CELL_MAKER on Linear Systolic DA Design

As mentioned earlier, the linear systolic design was not placed well within the Virtex XCV1000, resulting in long route lengths that decreased performance. The final placed and routed version of this design's layout is shown in Figure 8.1, and had a maximum clock rate of 86 MHz (see Section 9.1). This design had the best performance out of 20 different placements run by the Xilinx tools via their multi-pass place and route feature, yet was still slower than the summer tree design (see Section 9.1). As Figure 8.1 shows and as was discussed in Section 8.1, the placement strategy did not take advantage of any of the systolic, regular, design features built into the linear DA design.

The linear systolic DA design from Section 7.4 was constrained with the CELL_MAKER constraints described above so that the four tap groups were placed in a linear chain as shown in the top of Figure 8.2. A parallel-to-serial shift register at the input of the first four-tap group change the 16-bit *X* input

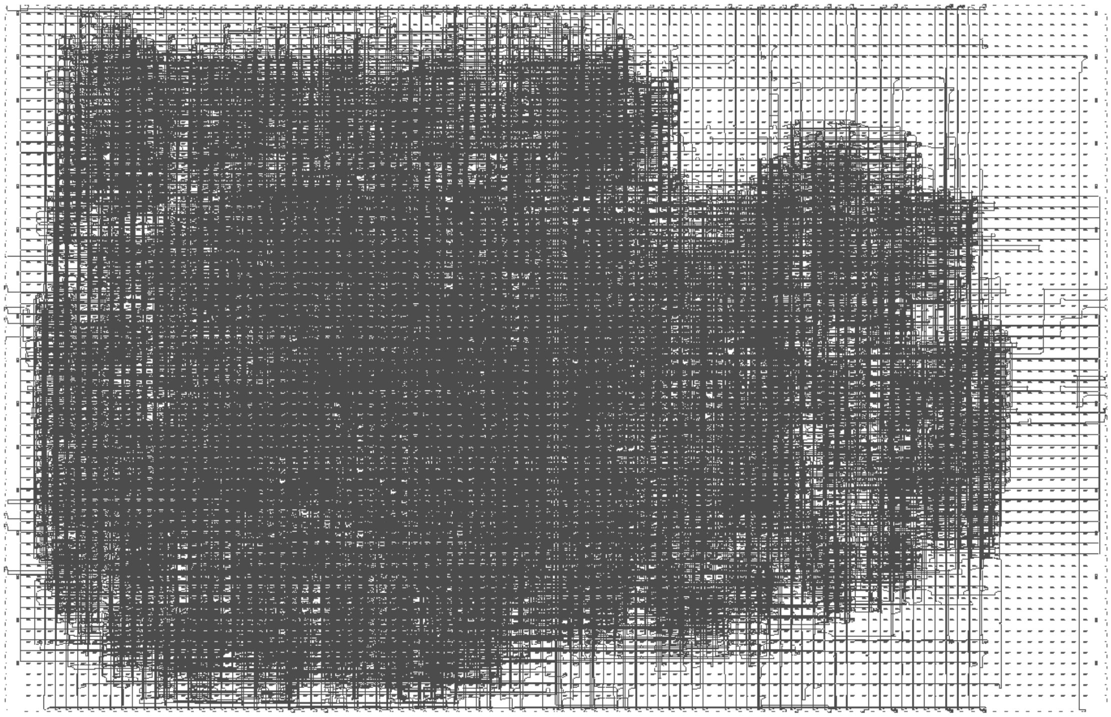
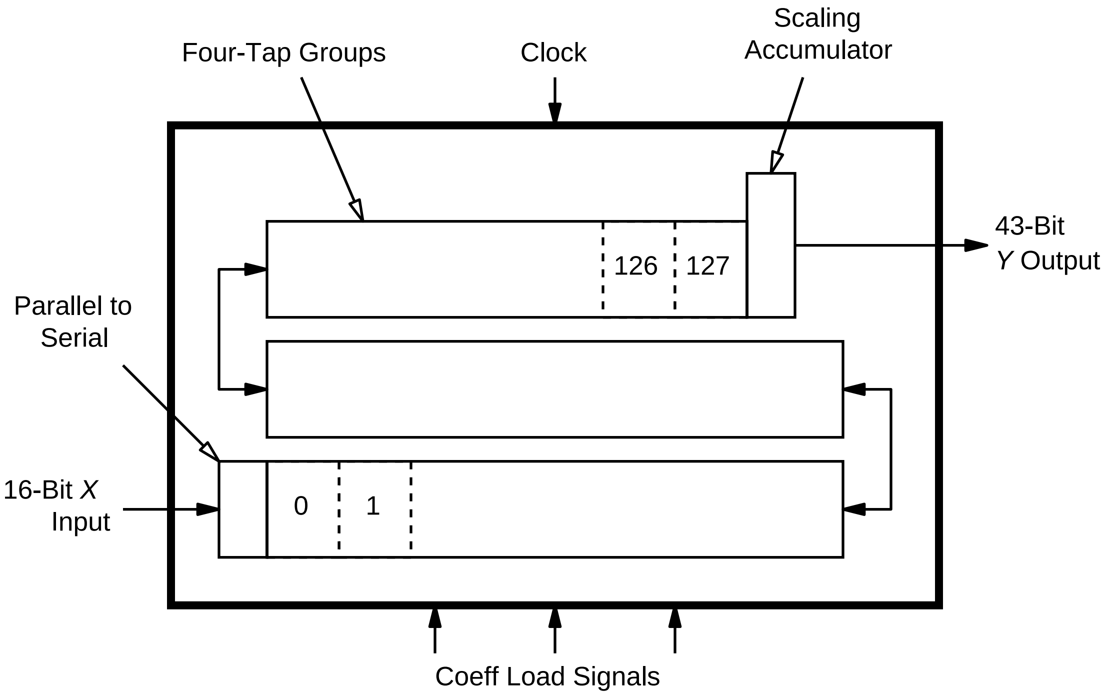


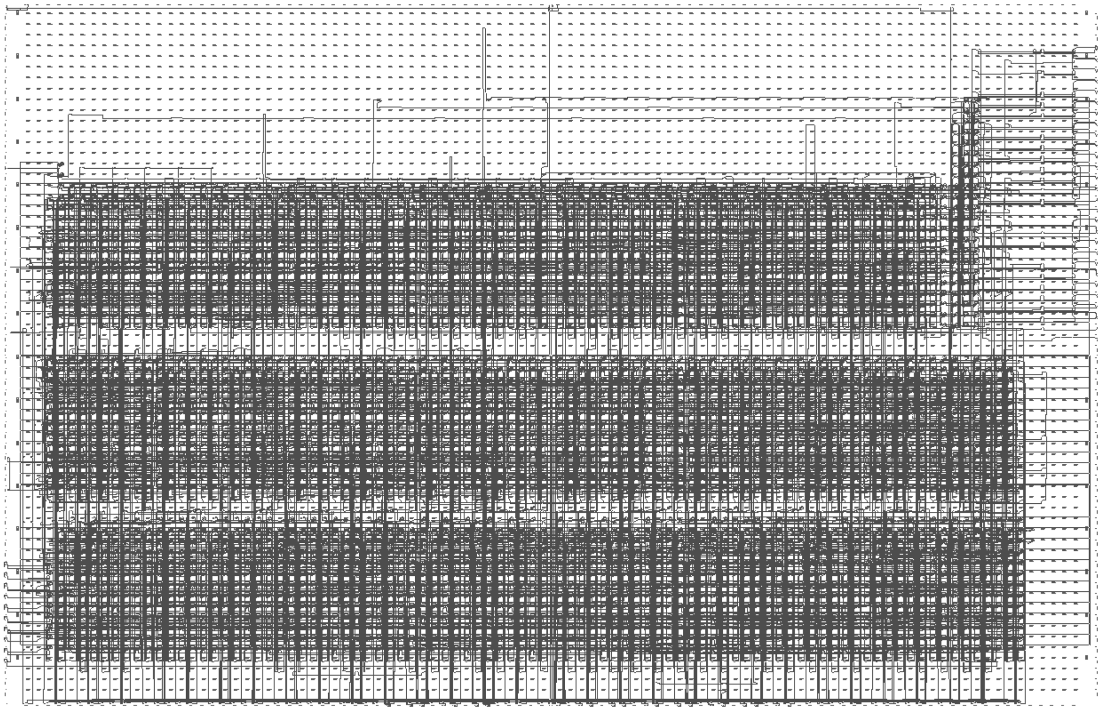
Figure 8.1: Placed and Routed Linear Systolic DA Design

into serial data for the DA algorithm, and a scaling accumulator at the output of the last four-tap group produced full 43-bit outputs.

CELL_MAKER was run on the linear systolic DA design with constraints added to the VHDL code, and the UCF file was applied to the Xilinx tools. The final placed and routed design with CELL_MAKER constraints is shown in the bottom of Figure 8.2. The performance increased from 86 MHz to 118 MHz (see Section 9.1) with the improved, systolic layout, a 37% improvement due to layout alone. The performance bottleneck was no longer unnecessarily long route lengths, as adder chain delays and connections between adjacent cells now limited the design.



Linear Systolic DA CELL_MAKER layout



Linear Systolic DA Final Placement and Routing After CELL_MAKER

Figure 8.2: CELL_MAKER Applied to Linear Systolic DA Design

Chapter 9

Results

9.1 Implementation Results

The area and performance results for the different designs presented in Chapter 7 are shown in Table 9.1. Performance results were obtained from the Xilinx Timing Analyzer, which reports worst-case timing statistics. All designs were implemented in a -6 speed grade Virtex XCV1000.

Design	Area (CLB Slices)	Performance (MHz)	Max Sample Rate (MHz)[a]	Performance/ Area ($x10^{-3}$)[b]
32 Parallel MACs	7,816	85.157	5.322	10.895
Bit-Level Systolic Array	> 20,736[c]	N/A	N/A	N/A
Summer Tree DA	6,253	92.842	5.803	14.846
Linear Systolic DA	6,446	85.955	5.372	13.335
Linear Systolic DA w/ CELL_MAKER Constraints	6,446	117.509	7.3443	18.230
4x4 FFA	6,900	85.164	5.323	12.343
Frequency Domain	> 6,800[d]	N/A	N/A	N/A

Table 9.1: Area and Performance Results

[a] Each design requires a clock rate 16 times faster than the sample rate, so max sample rate = performance / 16

[b] Ratio of performance to area of design

[c] 20,736 slices are required for the bit-level systolic array's main array cells only

[d] A minimum of 5,306 slices are required for multipliers and RAM if each multiplier is re-used 16 times which would require complex control logic. A low-bound estimate of 1,500 slices was used for all other logic in the design.

All of the designs that were implemented (the designs in Table 9.1 with performance results) exceeded the design specifications for the custom VLSI chip described in Chapter 4. Each design was able to fit within a single

XCV1000, and may be able to fit within a XCV600, which has 6,912 CLB slices. Each design was able to run with a coefficient load clock of at least 25 MHz, meaning that they all had dual banked coefficients that were able to be loaded within 1 ms. The implemented designs all had close to 18-bit output precision, and performed 512-tap FIR filtering on 16-bit inputs and 18-bit coefficients.

In Table 9.1, the column “performance/area” gives a figure of merit for each design taking into account both its performance and area by simply taking the ratio of the two measurements. The fastest design and the design with the best performance for a given area was the linear systolic design placed with CELL_MAKER constraints. The summer tree DA design was the smallest design. The summer tree design may have benefited with constraints applied to it as well, but a placement strategy was not able to be derived that could limit route lengths to adjacent cells as was done in the linear tree design. Therefore, a constrained summer tree design would never be able to meet the performance of the constrained linear systolic design.

With a sample rate of 7.3 MHz, and 512 multiply and add operations (two operations) performed each clock cycle, the linear tree DA design can perform 7.475 billion operations per second (GOPS). The custom VLSI chip was able to perform at 5.02 GOPS [Gre96], so the FPGA solution’s performance exceeded the performance of the custom VLSI chip designed with 1996 process technology.

9.2 Power Consumption

The Virtex requires the following processing power according to the documentation on the Annapolis Micro Systems Virtex-based Starfire board (see Chapter 10) [Ann99]:

$$Power = \frac{N_{FF} \cdot F_{MHz}}{1 \times 10^{5}} + 4.3, \tag{9.1}$$

where N_{FF} is the number of flip flops clocked by the processing clock, and F_{MHz} is the frequency of the processing clock in MHz. For the linear DA design, about 6,700 flip-flops and shift registers are clocked by the processing clock, so the power consumption for the design is 12.14 watts with a 117 MHz clock.

With 7.475 billion operations performed per second at 12.14 watts, the throughput/power factor for the linear DA design is 0.62 GOPS/Watt. The custom VLSI chip's throughput/power for its 512-tap real mode was 1.57 GOPS/Watt, or 2.5 times better than the FPGA design.

Chapter 10

Physical Implementation

10.1 Annapolis Microsystems Starfire Board

The Annapolis Microsystems Starfire Reconfigurable Computing Engine was used to demonstrate the distributed arithmetic design on real hardware. The Starfire board contains a single Virtex XCV1000 and two local SRAM memories, and is connected via PCI bus to a PC [Ann99].

10.2 Implementation

A host program was written for the PC which would load the Virtex XCV1000 with its active configuration, load one local memory with a block of input data used during simulation, load one of the distributed arithmetic filter's coefficient banks with pre-computed partial products, set the XCV1000's clock rate, and start the filter running. The filter would then operate, pulling input data from the input data local memory and outputting it to the other local memory. Upon completion, the host program would retrieve the output data from its local memory and compare it to a pre-computed correct set of filtered data. Figure 10.1 shows a block diagram of the Starfire board with the Virtex programmable logic element and the filter loaded into that element.

Due to time constraints, only an eight-tap DA filter (without CELL_MAKER) was able to be implemented on the Starfire board. This filter successfully ran on the Starfire up to 97 MHz, above the timing analyzer's maximum reported performance of 85 MHz. In addition, in this design, the performance bottle-

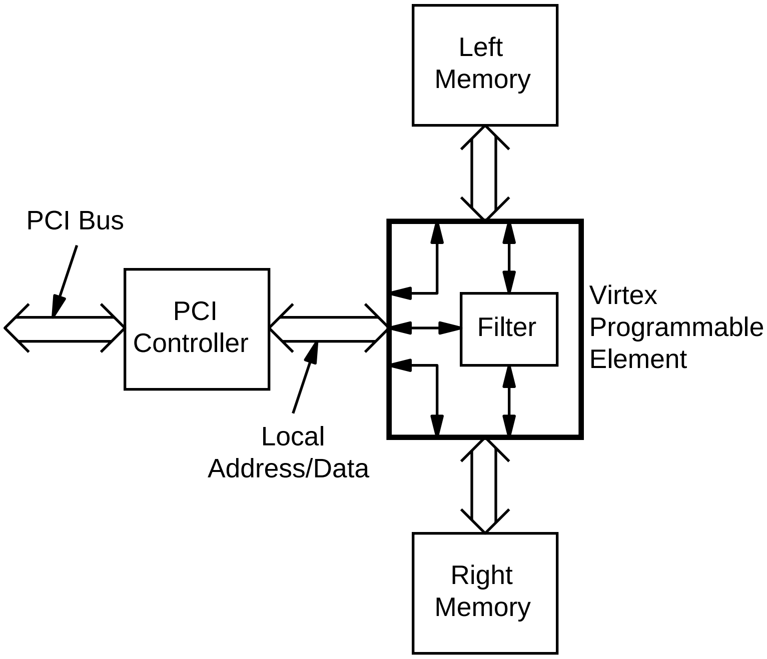


Figure 10.1: Starfire Board With Filter

neck was within the memory control logic, not within the filter or its control logic, so in a real system with streaming input and output data, the filter should be able to perform faster.

Chapter 11

Conclusions

A Virtex FPGA implementation using a layout-constrained linear-systolic distributed arithmetic design was built, simulated, and fielded in a reconfigurable-computing hardware board. This implementation was able to meet all of the specifications of a custom VLSI chip designed to perform the front-end digital signal processing for an adaptive radar system. This implementation demonstrated that reconfigurable computing now has the performance to meet the demanding requirements of high-bandwidth signal processing applications. Using reconfigurable computing for such an environment instead of custom VLSI would gain many benefits, including lower costs, faster and easier development and production times, and the ability to create entirely new types of systems and applications that can dynamically change their hardware in-system.

The layout of a FPGA design drastically affects its performance. The VHDL synthesis and place and route tools cannot be trusted to place a well-designed systolic structure optimally into a FPGA to maximize its performance. Placement constraints must be introduced into the placement process to produce the most optimal design, which can be done with an automated tool such as CELL_MAKER.

One drawback to a FPGA solution is that it consumes more power than a custom VLSI solution. Therefore, in applications where power or extreme performance is a necessity, reconfigurable computing may not apply. However, it is a computational medium that is rapidly opening to new classes of processing applications, and should continue to do so as the technology continues to improve.

References

[Ann99] Annapolis Micro Systems, Inc. *STARFIRE Reference Manual*, 1999. Revision 1.0.

[ASR98] James R. Anderson, Siddharth Sheth, and Kaushik Roy. “A coarse-grained FPGA architecture for high-performance FIR filtering.” In *Proceedings of The 1998 ACM/SIGDA Sixth International Symposium on Field-Programmable Gate Arrays (FPGA ’98)*, pages 234–243, Monterey, CA, 22-24 February 1998.

[BAK96] Duncan A. Buell, Jeffrey M. Arnold, and Walter J. Kleinfelder. *Splash 2: FPGAs in a Custom Computing Machine*. IEEE Computer Society Press, 1996.

[CKJ$^+$98] Jin-Cyun Chung, Yong-Bae Kim, Hang-Geun Jeong, Keshab K. Parhi, and Zhongfeng Wang. “Efficient parallel FIR filter implementations using frequency spectrum characteristics.” In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS ’98)*, pages 354–358, Monterey, CA, June 1998.

[Con96] Doug Conner. “Reconfigurable logic: hardware speed with software flexibility.” *EDN*, 28 March 1996.

[Gos95] Gregory R. Goslin. “Using Xilinx FPGAs to design custom digital signal processing devices.” In *DSPX 1995 Technical Proceedings*, page 595, 12 January 1995.

[Gos96a] Gregory R. Goslin. “A guide to using field programmable gate arrays (FPGAs) for application-specific digital signal processing performance." In John Schewel, Peter M. Athanas, V. Michael Bove, Jr., and John Watson, editors, *Proceedings High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*, pages 321–331, Boston, MA, 20-21 November 1996. SPIE-The International Society for Optical Engineering. Proc. SPIE 2914.

[Gos96b] Gregory R. Goslin. "Implement DSP functions in FPGAs to reduce cost and boost performance." *EDN*, 1996.

[Gre96] Joe Greco. Personal Communication, 19 June 1996.

[GW75] Herbert L. Groginsky and George A. Works. "A pipeline fast Fourier transform." In Liu [Liu75], pages 369–373.

[Hau98] Scott Hauck. "The roles of FPGAs in reprogrammable systems." *Proceedings of the IEEE*, 86(4):615–638, April 1998.

[Kna95] Steven K. Knapp. "Using programmable logic to accelerate DSP functions." Technical report, Xilinx Publications, 1995.

[Liu75] Bede Liu, editor. *Digital Filters and the Fast Fourier Transform*. Dowden, Hutchinson, and Ross, Inc., 1975.

[Liu95] Joseph Liu. "Detailed model shows FPGAs' true costs." *EDN*, 11 May 1995.

[MM99] Tyler J. Moeller and David R. Martinez. "Field programmable gate array based radar front-end digital signal processing." In *IEEE Symposium on Field-Programmable Custom Computing Machines*, Napa Valley, CA, 21-23 April 1999. IEEE Computer Society.

[MMT00] David R. Martinez, Tyler J. Moeller, and Ken Teitelbaum.

"Application of reconfigurable computing to a high performance front-end radar signal processor." *Journal of VLSI Signal Processing Systems*. Kluwer Academic Publishers, 2000. Submitted for publication.

[New95] Bernie New. "A distributed arithmetic approach to designing scalable DSP chips." *EDN*, 17 August 1995.

[OS89] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, 1989.

[OW75] Alan V. Oppenheim and Clifford J. Weinstein. "Effects of finite register length in digital filtering and the fast Fourier transform." In Liu [Liu75], pages 182–201.

[PH95] R. Petersen and Brad L. Hutchings. "An assessment of the suitability of FPGA-based systems for use in digital signal processing." In *The 5th International Workshop on Field-Programmable Logic and Applications (FPL '95)*, pages 293–302, Oxford, England, August 1995.

[PP97] David A. Parker and Keshab K. Parhi. "Low-area/power parallel FIR digital filter implementation." *Journal of VLSI Signal Processing*, 17:75–92, 1997.

[Son] William S. Song. "VLSI bit-level systolic array for radar front-end signal processing." Technical report, MIT Lincoln Laboratory.

[Sti98] George W. Stimson. *Introduction to Airborne Radar*. Scitech Publishing, Inc., 2nd edition, 1998.

[VH98] John Villasenor and Brad Hutchings. “The flexibility of configurable computing.” *IEEE Signal Processing Magazine*, pages 67–84, September 1998.

[War94] Jim Ward. “Space-time adaptive processing for airborne radar.” Technical Report TR-1015, MIT Lincoln Laboratory, 13 December 1994.

[Whi89] Stanley A. White. “Applications of distributed arithmetic to digital signal processing: A tutorial review.” *IEEE ASSP Magazine*, July 1989.

[WWC88] Chin-Liang Wang, Chee-Ho Wei, and Sin-Horng Chen. “Efficient bit-level systolic array implementation of FIR and IIR digital filters.” *IEEE Journal on Selected Areas in Communications*, 6(3):484–493, April 1988.

[Xila] Xilinx Publications. “FPGAs and DSP.” Technical report.

[Xilb] Xilinx Publications. “The role of distributed arithmetic in FPGA-based signal processing.” Technical report.

[Xil98a] Xilinx Publications. “Virtex 2.5 V field programmable gate arrays.” Advance product specification, 9 November 1998. Version 1.1.

[Xil98b] Xilinx Publications. “Xilinx breaks one million-gate barrier with delivery of new Virtex series,” 26 October 1998. Press Release.