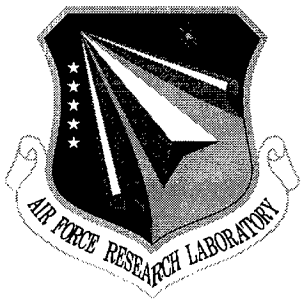
AFRL-IF-RS-TR-1998-212
Final Technical Report
December 1998



DIGITAL SIGNAL PROCESSING RAPID PROTOTYPING WITH FIELD PROGRAMMABLE GATE ARRAYS

University of Kansas

Kambhammettu Nalinimohan, Venkatesh Rao, and Glenn E. Prescott

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

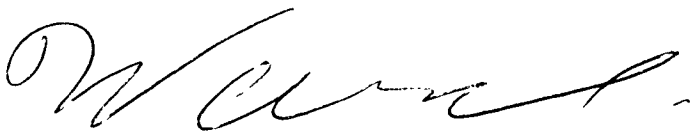
AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-1998-212 has been reviewed and is approved for publication.

APPROVED: 

STEPHEN C. TYLER
Project Engineer

FOR THE DIRECTOR: 

WARREN H. DEBANY, JR., Technical Advisor
Information Grid Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFGC, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1998	Final Feb 95 - Feb 98

4. TITLE AND SUBTITLE

DIGITAL SIGNAL PROCESSING RAPID PROTOTYPING WITH FIELD PROGRAMMABLE GATE ARRAYS

5. FUNDING NUMBERS

C - F30602-95-C-0214
PE - 62702F
PR - 4519
TA - 42
WU - 92

6. AUTHOR(S)

Kambhammettu Nalinimohan, Venkatesh Rao, and Glenn E. Prescott

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Kansas
Information and Telecommunication Technology Center
Lawrence KA 66045

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

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

AFRL/IFGC
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-1998-212

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Stephen C. Tyler/IFGC/(315) 330-3618

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

N/A

13. ABSTRACT *(Maximum 200 words)*

Until recently radio transmitters and receivers were almost exclusively implemented with analog electronic components. However, a new approach is now becoming popular -- one that employs digital electronics to implement most of the analog signal processing functions in the radio. The evolution in radio system design is driven by the ever increasing speed and decreasing cost of microprocessors and high performance analog-to-digital (ADC) and digital-to-analog (DAC) converters. It is no longer uncommon to sample a received signal at the intermediate frequency (IF) stage and process the signal with numerical algorithms using specialized digital signal processing (DSP) hardware. The DSP hardware performs a variety of operations on the signal including downconversion, demodulation, and filtering; all of which are inherently continuous-time (i.e., analog) processes.

Modern field programmable gate arrays can implement functions beyond the capabilities of today's DSP micro-processors. In fact, they have the potential to provide performance increases of an order of magnitude or better over traditional DSP microprocessors, but with the same flexibility. These devices can provide the programmability of software, the high speed of hardware and can be reconfigured in-circuit with no physical change to the hardware. In fact, FPGAs are really "soft" hardware, in that they are a good compromise between flexible all-software approaches which unfortunately limit throughput, and custom hardware implementations, which are more expensive and inflexible. FPGAs offer a powerful approach -- an architecture tailored to the specific application.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Digital Signal Processing, Field Programmable Gate Arrays, Communications, Spread Spectrum	108
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Until recently radio transmitters and receivers were almost exclusively implemented with analog electronic components. However, a new approach is now becoming popular - one that employs digital electronics to implement most of the analog signal processing functions in the radio. This evolution in radio system design is driven by the ever increasing speed and decreasing cost of microprocessors and high performance analog-to-digital (ADC) and digital-to-analog (DAC) converters. It is no longer uncommon to sample a received signal at the intermediate frequency (IF) stage and process the signal with numerical algorithms using a specialized digital signal processing (DSP) hardware. The DSP hardware performs a variety of operations on the signal including downconversion, demodulation, and filtering; all of which are inherently continuous-time (i.e., analog) processes.

1.1 DSP: Capabilities and Requirements

The mathematics of digital signal processing provides the framework for the design of software radio algorithms, while modern high speed digital electronic components

make real time implementation of these algorithms possible. However, the hardware currently available to implement DSP algorithms for all stages of the radio system is still limited in speed, accuracy and flexibility. Initially, digital signal processing was used only for baseband waveform processing. As digital electronic devices increased in speed, DSP was soon applied to signal processing functions performed at higher frequencies - e.g., the final IF stage in a radio receiver. Functions such as IF bandpass filtering, automatic gain control (AGC), and coherent modulation and demodulation are typically required at this stage. In the absence of a sufficiently high speed processing capability, innovative techniques such as sub-sampling are used to process bandpass signals of small to moderate bandwidth. This has allowed the boundary between analog and digital processing to be pushed as far up the signal path towards the antenna as permitted by physical electronic devices. For most types of moderate data rate communications - on the order of 100 kB/s or less - bandwidth is not a serious barrier to DSP techniques. However, military radio systems pose a notable challenge because of the wide bandwidth characteristics of spread spectrum modulation.

1.2 Military Radio Signal Processing

Military communication systems often require the use of spread spectrum techniques to provide an antijam (AJ) capability; or some measure of covertness through the use of low probability of intercept (LPI) waveforms. The result is that extremely wide bandwidth signals are present at the output stage of the transmitter and the input stages of the receiver. We know from the Nyquist theorem and fundamental bandpass sampling techniques that bandpass signals can be sampled at a rate no less than the bandwidth of the signal; so high frequencies alone do not put a limitation on DSP processor capability. However, wide bandwidth signals are a challenge for any type of digital signal processing hardware, and they are especially troublesome for conventional DSP microprocessors. While conventional DSP microprocessors are

optimized for real-time data processing, they are nevertheless implemented using the traditional serial-based architecture - an inherently serial architecture which uses a single multiplier and executes one instruction at a time. While providing the advantage of flexibility through programmability, this architecture limits the speed with which signal samples can be processed. Even modern DSP microprocessors operating at 40 million instructions per second (MIPS) have a useful bandwidth limit of less than 500 kHz. This is especially troublesome for military communication systems which employ AJ and LPI waveforms having typical bandwidths in excess of 10 MHz.

1.3 Advantages of Specialized Digital Hardware

When digital signal processing at wide bandwidths is required the radio designer turns to specialized hardware which can operate at much higher throughputs than is possible with a DSP microprocessor. These include application specific standard products (ASSP), application specific integrated circuits (ASIC), and field programmable gate arrays (FPGA).

Application Specific Standard Products (ASSP) such as FIR filters, correlators, and FFT processors, permit certain popular DSP algorithms or functions to be optimized in hardware at the cost of flexibility. Use of ASSPs can significantly increase the device count and often presents special interface problems which can lead to further complications. Furthermore, due to a narrow range of applicability, many ASSPs may not be available in state of the art process technology [1].

When performance is a factor and product volume is high, many designers turn to ASIC technology. ASIC technology offers the ability to design a custom architecture that is optimized for a particular application. For example a conventional DSP microprocessor has only a single multiply-accumulate (MAC) stage (see Section 3), so each filter tap must be executed sequentially. An ASIC implementation of a DSP algorithm, on the other hand, might have multiple parallel multiply-accumulate

(MAC) stages. When comparing the performance of the ASIC versus the DSP microprocessor it becomes apparent that the DSP microprocessor offers slow speed but maximum flexibility (due to programmability) while the ASIC provides high speed with minimal flexibility. Between these two extremes lies the field programmable gate array [2].

1.4 Field Programmable Gate Arrays

Modern field programmable gate arrays can implement functions beyond the capabilities of today's DSP microprocessors. In fact, they have the potential to provide performance increases of an order of magnitude or better over traditional DSP microprocessors, but with the same flexibility [3]. These devices can provide the programmability of software, the high speed of hardware and can be reconfigured in-circuit with no physical change to the hardware. In fact, FPGAs are really "soft" hardware, in that they are a good compromise between flexible all-software approaches which unfortunately limit throughput, and custom hardware implementations, which are more expensive and inflexible [4]. FPGAs offer a powerful approach - an architecture tailored to the specific application. Because the logic in an FPGA is flexible and amorphous, a DSP function can be mapped directly to the resources available on the device. Modern FPGAs have sufficient capacity to fit multiple MACs or algorithms into a single device along with the interface circuitry required by the application - a single chip solution.

Although FPGAs can out-perform DSP microprocessors under some circumstances, they are not universally the best choice for processing at every stage of the software radio. The limitations and advantages of FPGAs compared to those of the DSP microprocessor are examined further in the sections that follow. At the conclusion of this report, a suggestion is presented for the use of both the FPGA and the DSP microprocessor in a software radio testbed.

1.5 DSP Microprocessors

A modern programmable DSP microprocessor typically provides up to 200 MIPS or 50 MFLOPS. For example, the TMS320C40 has 50 MFLOPS at 25 MIPS with a 50 MHz clock. There are many high performance DSP processors on the market, but they are not suited to all DSP applications. Their general purpose architecture makes these DSP processors flexible but they may not be fast enough or cost effective for all systems. The DSP processor provides flexibility through software instruction decoding and execution while providing high performance arithmetic components such as a fast array multiplier and multiple memory banks to increase data throughput. The performance limit for commercially available DSP processors currently tops out at about 50 MIPS [6].

Before exploring how DSP functions can be implemented on a variety of programmable logic devices, a broader definition of digital signal processing is in order. The term "DSP" applies broadly to discrete-time mathematical processes executed in real-time. These include functions such as:

- Digital Filtering (FIR and IIR)
- Convolution
- Correlation
- Fast Fourier Transforms

Implementation of these functions involves only the basic digital operations of addition, multiplication and delay/shift as indicated in the equation below:

$$y(n) = \sum_{k=0}^{N-1} h(k)x(n-k)$$

where x(n) can be interpreted as the input data sequence, and h(k) is the impulse response sequence of length N, and y(n) is the output. Depending on the data format and suitable choice of tap coefficients, a number of different functions result:

- Digital Filtering and Convolution - h(k) are the filter coefficients
- Correlation - h(k) refers to another input sequence
- Fourier Transform - h(k) are constants in complex exponential form

Most of these functions require the incoming data to be multiplied or added with various internal feedback mechanisms to perform the desired mathematical function. This primitive function which is so common to DSP algorithms is called the multiply/accumulate (MAC) [3]. The MAC may actually consist of 6 to 12 operations; however, to increase performance, most general-purpose DSP processors perform a MAC in a single clock cycle or less. Most DSP processors have a fixed-point MAC while some have a more expensive floating point MAC. Each tap of a digital filter requires one MAC cycle - for example a 16-tap filter requires 16 MAC cycles. Because most DSPs only have a single MAC unit, each tap is processed sequentially, and all taps are processed during a single sample time interval, slowing overall system performance. Thus a 50 MHz (25 MIPS) DSP processor performs at less than 2 Msps [1].

The need to process instructions sequentially will always remain a fundamental performance limitation of microprocessors. Acceleration via dedicated hardware has long been a solution to this problem. Traditionally, this meant dedicated hardware in the form of an ASIC, or in some special cases, multiprocessing. Recently another viable alternative has been introduced - the Field Programmable Gate Array. The FPGA offers the advantage of fast hardware which can be reconfigured under software control. The use of FPGAs in DSP applications is the subject of the next section.

Chapter 2

Software Radio

The essential concept of software radio is that most of the analog signal processing operations of the radio transmitter and receiver are implemented with digital hardware using DSP techniques. The placement of the receiver analog to digital converter (ADC) and the transmitter digital to analog converter (DAC) as close to the antenna as possible are distinguishing characteristics of the software radio. In the software radio receiver, the approach often used is to digitize an entire band and to perform IF processing, baseband, bit stream and other functions completely in software [5]. This approach requires the use of high speed analog to digital converters and high speed DSP microprocessors. However, the signal processing requirements for military and commercial radio systems employing high data rate signals or spread spectrum modulation easily exceeds the processing speeds currently available in off-the-shelf DSP microprocessors. In this case, special purpose DSP hardware, application specific devices and field programmable gate arrays can play an important role.

The motivation for implementing radios in software is that a highly flexible and reconfigurable communication system can be implemented for relatively low cost. The ability to adapt the radio to its environment by changing filters, changing modulation schemes, switching channels, using different protocols and dynamically assigning

channels and capacity are features which are impractical to deliver with hardware alone. Since the behavior of the software radio can be changed so easily, defining a particular architecture does not limit the radio to one specific function. Instead, multiple radio systems can share a common front-end analog radio tuner while having independent digital processing for each individual radio channel. [5]

2.1 A Software Radio Architecture

A software radio is essentially a hybrid analog and digital processing system. As illustrated in Figure 2.1, fixed analog filtering and frequency conversion are still used in the RF stages. Conceivably, there will always be a need for an analog low-noise preamplifier to capture the signal from the antenna and establish the noise figure for the receiver. Also, a downconversion operation which places the signal at some convenient intermediate frequency and allows for additional conditioning of the signal before sampling will probably continue to be a part of the software radio system for the next decade.

Using a sufficiently fast DSP microprocessor, a single device could be used to process the signal through all stages of the communication system. However, the signal processing requirements for each stage are quite different. In the IF stages, relatively simple high speed digital processing is needed, and special purpose DSP hardware can be used to satisfy this requirement. At this stage, signal processing is usually limited to filtering, correlation or FFT processing. At the baseband stage the spread spectrum modulation has been removed and the bandwidth of the signal is much narrower, meaning that fewer samples need to be processed per unit time. However, the complexity of the algorithms required at this stage increases dramatically, and the extra time between samples is required in order to implement digital phase locked loops and other computationally intensive algorithms. Use of simple, high speed DSP processing at the wide bandwidth stages and slower, more flexible processing at the lower bandwidth stages will efficiently satisfy both the complexity and high throughput requirements of modern radio systems [4].

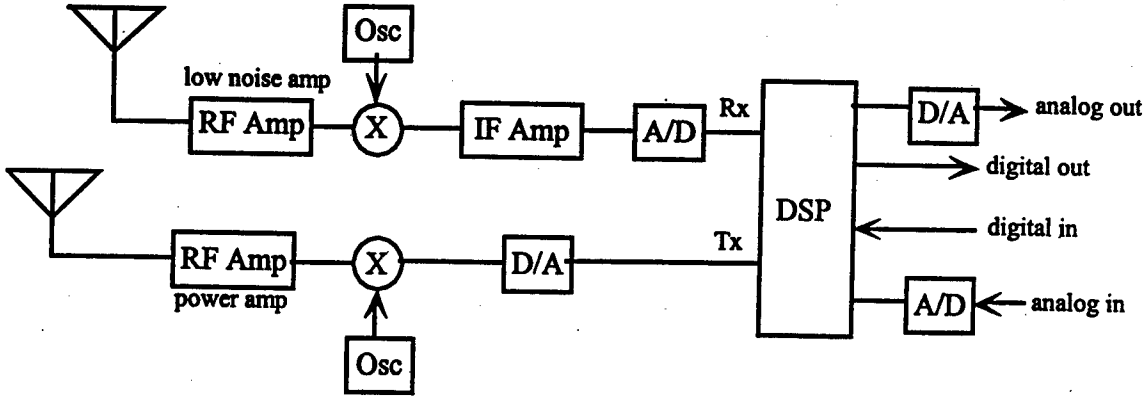


Figure 2.1 - Software Radio System

2.2 Software Radio Processing Requirements

The single most critical requirement in software radio is real-time processing. If the system is to operate in real time, then the data must be moved in and out of the DSP microprocessor on a regular (i.e., sample by sample) basis, where hundreds of instructions may need to be executed for every sample that enters the processor. Obviously, low sample rates are desired for this reason. However, the sample rate requirement is dictated primarily by the information bandwidth of the signal. The information bandwidth in radio systems ranges from under 4 kHz for HF voice band channels to over 1 MHz for cellular systems. Spread spectrum (or CDMA) systems are a notable challenge, especially for military systems where interference excision techniques, or chip wave shaping (for LPI enhancement), are applied directly to the spread waveform. This requires that complex signal processing be applied at the chip level, which can be one or two orders of magnitude wider bandwidth than the information signal.

A well designed system will use a variety of sampling rates to achieve an efficient flow of data through the processor. At the A/D or the D/A stage, over sampling is quite often used. Over sampling of the signal is useful to shift aliases out of band and simplify filtering, so faster sample rates and narrower bandwidths are used. On the other hand,

novel under sampling sampling techniques are possible with stable, linear analog-to-digital converters. Under sampling techniques can be used to implement bandpass sampling - digitizing the signal in the second or third Nyquist zone, so that the desired signals will be aliased in-band by the sampling. Both of these techniques can be combined as needed within the various stages of the software radio to enhance the signal to noise ratio yet maintain the sample rate at the lowest practical level.

When the time between samples is on the order of tens of microseconds to hundreds of nanoseconds such single-sample operations require hundreds of MIPS (million instructions per second) and/or MFLOPS (million floating operations per second) to Giga-FLOPS. A good FIR/IIR channel selection filter could require about 100 operations per sample at 30 Msps, or 3000 MIPS. Using a naive brute force approach, we would require 15 to 60 DSPs cooperating for this section alone, repeated for every channel. As a result, even with faster devices, software on DSPs still cannot be used for the down conversion itself, but must still essentially operate at baseband (albeit a much wider baseband, up to a few MHz). Even the most fundamental demodulation or tuning algorithm requires 10 operations per sample, which would limit a DSP microprocessor to filtering signals with a bandwidth of a few hundred kHz. In a conventional voice-band cellular system, baseband processing requirements can range from 10 to 100 MIPS/MFLOPS per channel; while any digital signal processing at the IF frequency can drive the processing requirements to 500 MIPS/ MFLOPS and upwards of 10 GFLOPS [5].

We contend with these formidable processing challenges by abandoning the use of general purpose processors in favor of a mixed approach in which high speed digital hardware is used in the earliest stages, doing much of the filtering and processing in fast digital logic. When the signal reaches the post-IF stages, the processing load has been reduced considerably so that it can now be effectively handled by general purpose DSP processors. As long as this specialized hardware is versatile and is controllable to some extent from software, a hybrid architecture will meet our requirements. Most IF processing and chip rate processing can be off-loaded to these special purpose devices until the day that general purpose processors with sufficient processing power are available and cost effective.

2.3 DSP Hardware Alternatives

The most significant limiting factor in development of software radio systems has been the lack of sufficiently fast hardware - most notably, fast DSP microprocessors. As high performance, high speed ADCs have become available commercially, hybrid techniques using specialized digital hardware have become more common, while use of DSP microprocessors has lagged behind [5]. DSPs are getting ever faster, but it will be a while before we can use a single 'ultimate' chip to do everything. Instead, the idea of using multiprocessing to share the effort seems attractive.

Multiprocessing as an alternative to the processing limitations of conventional DSPs can have only limited success. First of all, traditional DSP architectures were not well suited to multiprocessing. In fact, there are only one or two commercially available DSP processors which have the architecture to efficiently support multiprocessing - most notably the Texas Instruments TMS320C40. Also, software to support parallel and multiprocessing is scarce and expensive. Secondly, it is a characteristic of a DSP (as contrasted with a conventional microprocessor) that it must operate on a continuous flow of data. There are few functions in the software radio that could benefit from the power of parallel processors.

Software radios ideally place most IF, and all baseband, bit stream and source processing in a single processor. However, when we examine the speed requirements of the IF stage, especially when spread spectrum is used, we conclude that we need a special purpose device - and this is where FPGAs come into favor. Some of the lower data rate anti-jam tactical communications standards, such as HaveQuick and SINCGARS, are best implemented using high dynamic range software-oriented digital signal processing. In these radios, FPGAs could effectively provide the core of real-time sample rate and baud rate pipelined processing. They could also be used in their more conventional role of providing gate level support for the other processors and ASICs that make up the system [4].

Recent technical history has suggested that only software and not hardware possesses the programmability that is needed for versatile multi-role radio designs. The

flexibility of software is high; but the throughput necessary for any respectable data rate is low, making it suitable only for voice processing data rates. However, the availability of high speed FPGAs provides a greatly enhanced DSP capability which can be reprogrammed to handle wideband digital signal processing tasks. This permits a flexible architecture consisting of dedicated wideband ASICs, FPGAs, and programmable narrowband DSP processors. In the near future, reconfigurable modem architectures will provide in excess of 400,000 gates of programmable hardware with throughputs measured in the 100 millions of operations per second and at power consumption levels under 2 watts [4].

2.4 A DSP Testbed for Military Tactical Radio

Current research at the Rome site of the AF Research Laboratory is focused on the development, testing and evaluation of algorithms for future Air Force radio systems. It is understood that these are radio systems which will be implemented using state-of-the-art digital signal processing hardware. Therefore, the question of how to make the best use of currently available DSP technology is one that must be answered. After examining the strengths and weaknesses of DSP microprocessors, application specific signal processing devices and FPGAs, we have concluded that there is an appropriate place for each of these technologies in the modern software radio. One notable characteristic of military radio systems is that they employ wideband spread spectrum modulation in order to reduce the effects of jamming or to provide some level of signal covertness. In either case, at the receiver, we would like to defer any signal processing of the spread spectrum signal until after the despreading function has been completed. However there are military requirements which necessitate processing at the spread bandwidth. For example, interference excision and detection of intercepted covert communications signals are often essential capabilities of these radios. In order to test the effectiveness of excision and detection algorithms, a testbed is needed which can process the wide bandwidth spread spectrum signal in real time. General purpose DSP

microprocessors would be the most cost effective and flexible solution, however DSP microprocessors are not capable of handling the high data throughput rates required at spread spectrum bandwidths. Special purpose digital hardware is a likely candidate, but these devices are most appropriately used for those functions which will never need to change - the downconversion process, for example. The FPGA however, is a powerful new technology which can be used to maximum advantage in this application. The engineer can configure the internal structure of the FPGA via software - in effect, generating a unique hardware design under software control which suits the signal processing task at hand.

2.4.1 Proposed Testbed Configuration

A proposed system architecture for the software radio testbed is shown in Figure 4. FPGAs are used to advantage in the chip stream section of the transmitter and receiver where a few simple DSP operations need to be performed at a high rate of speed. In FPGA X1, interference excision processes of any description can be implemented immediately following the A/D converter, including FIR filters, Fast Fourier Transforms, and even adaptive filters. Following this stage, the de-spreading process can be performed using FPGA X2. On the transmitter side, spreading is accomplished with X5. Flexibility can be obtained by using FPGAs here in order to quickly modify the spreading sequence, giving an added dimension to the role of the spreading sequence in maintaining interference-free or covert communications. A need still exists for the general purpose DSP microprocessor to perform the complex operations of demodulation, system timing, carrier extraction and adaptive equalization. Also, multiple DSP processors can be configured to perform complex tasks in parallel using an appropriate processor, such as the TMS320C40. Once the bit level decision has been made, digital signal processing using 16-bit (or longer) word lengths is no longer needed - the signal of interest is now in the form of a binary information bit stream. This data bit stream can now be processed much more efficiently using digital logic techniques. In this stage X3 and X4 represent FPGAs which can be used to implement a variety of bitstream processes such as interleaving, forward error correction, compaction/compression, and encryption.

The testbed is implemented with a balance of all three forms of current DSP hardware technology - FPGAs, ASSPs, and DSP microprocessors - each used in the most appropriate stage of the software radio. This testbed can be easily used to provide the best in currently available signal processing technology to implement and test radio algorithms

2.4.2 A Rapid Prototyping Facility

The testbed described in the previous section can be implemented using equipment and software presently available at the University of Kansas and the AFRL Rome site. Figure 5 shows the overall configuration of the testbed using a combination of software and hardware tools. The software tools are used to program the DSP microprocessor and to configure the FPGAs. At the top level is the Signal Processing Worksystem (SPW) which allows the algorithms for both the FPGA and DSP microprocessor to be developed, tested and interconnected at the highest level of abstraction. On the FPGA side, the SPW design is input to the fixed point Hardware Design System (HDS), which is actually part of SPW. HDS converts the top level design to a form which is suitable for implementation in digital hardware. At this point the Synopsys tools are used to convert the HDS output into VHSIC (Very High Speed Integrated Circuit) Hardware Design Language - VHDL. The VHDL description of the design is then input to the Xilinx tools which formats it for placement in the FPGA, whose internal circuitry is configured to implement the design.

The FPGAs are physically located on the Aptix Field Programmable Circuit Board, which allows FPGAs and other digital hardware to be placed and interconnected via software. The Aptix board is connected to one of the communications ports of the TMS320C40 processors located in the Ironics VME box. The C40s are the DSP microprocessors which perform the more complex tasks in the radio as discussed previously. The C40s are programmed from the top level using the SPW in the same manner that it was used when beginning design of the FPGA algorithm. After the algorithm design has been completed, the SPW output is processed by the Code Generation System (CGS) which is an ancillary function of SPW. The code generation

system produces C40-specific assembly language which is then input to the Texas Instrument floating point software tools, where it is compiled and loaded into the C40 microprocessors. The SPW Multiprox tool is also part of the software set which allows the efficient partitioning of DSP tasks onto multiple processors.

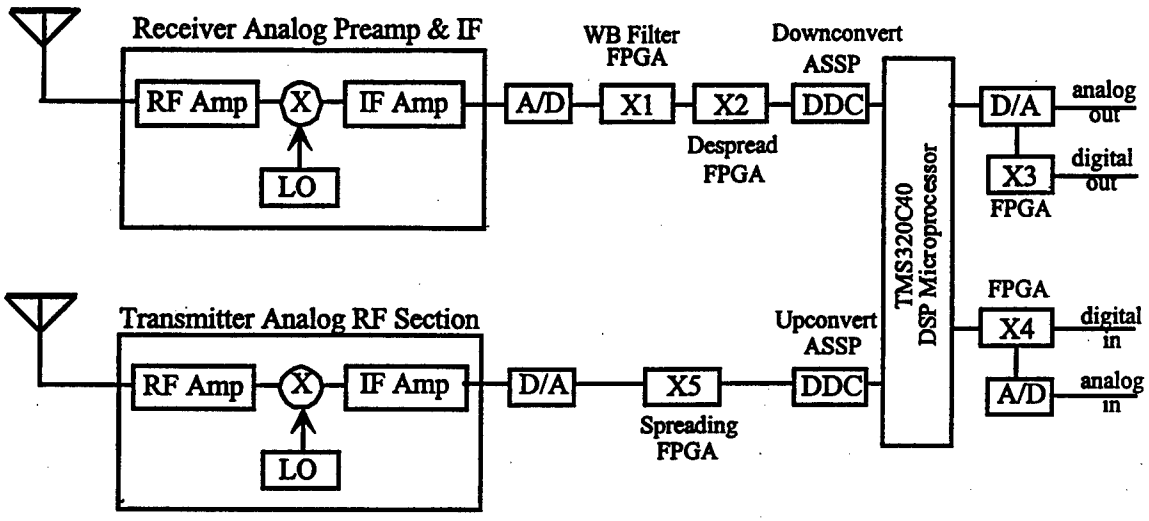


Figure 2-2: Tactical Radio Testbed

There are several hardware additions to the system which support the overall testbed. For example, a multiple channel analog interface board is available as part of the C40 system and is installed in the Ironics VME chassis. This analog interface board is used to input or output the baseband information signal - speech, data, or compressed video - directly into the C40 DSP microprocessors. They can also bypass the C40s and exchange signals directly with the Aptix board in order to have direct access to an FPGA for bit stream processing. Also Connected to the Aptix board are high speed (40 MHz) A/D and D/A converters which are interfaced to the analog RF & IF system. Communication signals at RF or IF can be routed to and from the Aptix board through the A/D and D/A converters, to complete the implementation of the testbed described in the previous section.

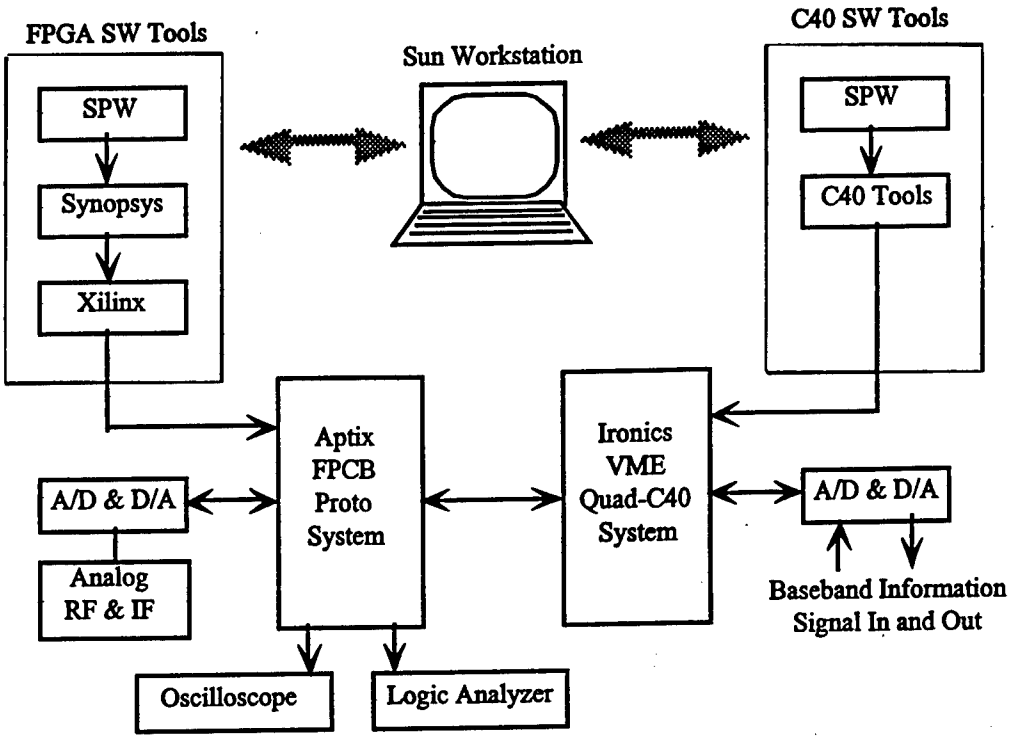


Figure 2-3: Rapid Prototyping Configuration

Chapter 3

Field Programmable Gate Arrays

Programmable hardware has been available for many years - conventional memory devices are the most obvious example. Various PLDs (programmable logic devices) have long been used in implementing state machines and "glue" logic, among other things. However, the available devices have tended to have restricted architectures and to be rather small [7]. The last decade has seen a significant change with the introduction of a variety of field programmable gate arrays, as well as an evolution of some PLDs into much larger devices with extended architectures. Essentially, the FPGA is a general purpose programmable logic device consisting of a regular array of cells with distributed routing that can be configured with a specific design by the user, without the need to fabricate an application specific device (i.e., an ASIC) [8].

3.1 Programmable Logic Technology

There are a variety of FPGA architectures available depending upon the manufacturer. However, there is one broad distinction that can be made regarding FPGA structure: the architectures are either *course-grained* or *fine-grained* [7]. The earlier devices were

simple arrays of logic gates which were programmable in the field in much the same way as a conventional ROM. These devices are considered fine-grained in the sense that there can be a large number of very simple logic operations which can be interconnected. On the other hand, modern FPGAs have a relatively smaller number of more complex logic cells available.

Other than granularity, FPGAs are differentiated by their chip level architecture and their interchip wiring organization. As an example, the Xilinx 3000 family FPGAs consist of an array of cells called CLBs (configurable logic blocks). Each CLB contains two latches and a function generator as illustrated in Figure 3.1. The internal connections within the cell and the lookup table in the function generators are determined by configuration bits held in an integrated SRAM. This allows an individual cell to implement quite complex combinational and sequential functions. The routing resources allow the cells to be connected as required, at least in principle. In practice, the problem of routing a congested design is the major obstacle in obtaining highest performance.

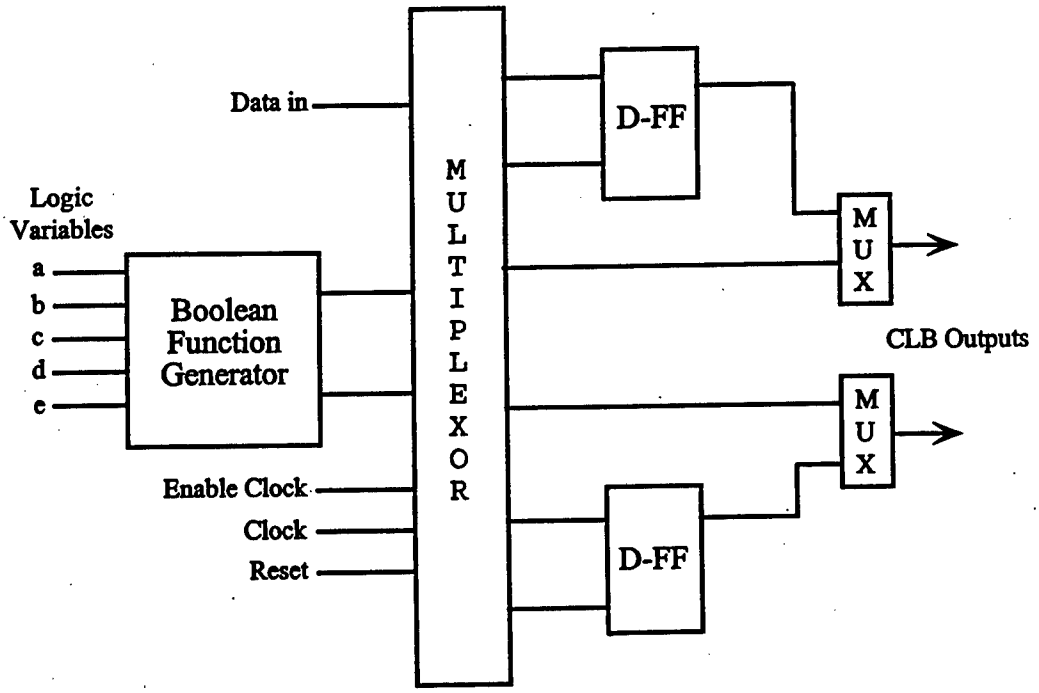


Figure 3.1 - Configurable Logic Block of the X3000 FPGA

FPGAs are just beginning to have a significant impact, although their cost is still relatively high (i.e., hundreds of dollars for the largest devices). Two application areas which traditionally have dominated their use are general purpose gate-level logic support (i.e., glue logic) and emulation of new IC designs. However, FPGA manufacturers believe that their products will change the way in which digital design is approached in a revolution similar to that engendered by the microprocessor [7]. The fact that FPGAs are now being investigated for use in high speed DSP applications is an indication of the broad impact they may have in digital applications of all kinds.

3.2 Practical Consideration in the Use of FPGAs

Because the FPGA is programmable in manner similar to a microprocessor, it is already becoming widely used. However, the configuring of hardware to fit a specific computation is significantly different from the programming of a microprocessor. In particular, the microprocessor has a fixed instruction set, and all solutions are algorithmic in nature. In contrast, an FPGAs internal structure must be customized to implement a particular algorithm. Since digital hardware designs are not software driven, the overhead associated with command interpretation, scheduling and execution is eliminated and there is a substantial gain in speed. Furthermore, a hardware design can take advantage of parallel implementations to eliminate bottlenecks [2]. It is interesting to note that we may even combine the two approaches and compile a specialized microprocessor into the FPGA with a restricted instruction set chosen to suite any particular application.

It often occurs that a computation is better suited for either dedicated hardware or microprocessor software. This is the situation we are examining in the software radio - when to use FPGAs and when to use DSP microprocessors. Simply stated, an FPGA is appropriate when the design calls for the performance of an ASIC and the flexibility of a microprocessor. An FPGA should not be used if the algorithms to be implemented are complex, or vary significantly in structure or complexity. Determining when to offload

DSP algorithms to FPGAs requires an analysis of speed versus problem size. At one end of the scale, problem size gets very large and direct hardware solutions become too difficult and expensive to build [2].

The advantage of FPGAs is that they represent a compact integrated programmable hardware solution which can be user configured for any conceivable logic design. Current designs contain in excess of 40,000 logic gates, all under the control of the designer. On the other hand, FPGAs have some notable disadvantages. First, there internal routing contributes substantial delay between logic elements resulting in a significant limitation in performance, although parallelism and pipelining can still be used. The second disadvantage is that it is not possible to execute a variety of arithmetic operations within the logic resources available. Added to this is that the programming of FPGAs is difficult, especially when implementing DSP functions [9].

3.3 Binary Arithmetic Techniques for FPGAs

The primary limitation of the FPGA when used in DSP applications is arithmetic - most notably multiplication. When FPGAs are used for DSP applications, the multiplier circuits must be implemented with the available chip resources. However, a hardware multiplier is a reasonably complex circuit, as evidenced by the fact that conventional DSP microprocessors contain only a single hardware multiplier, and it occupies most of the real estate on the chip. A state-of-the-art FPGA can support no more than a handful of multipliers, meaning that brute force multiplication is often avoided in some of the most common operations - e.g., filtering or correlation.

When implementing multipliers in hardware, two basic alternatives are available: The fully parallel array multiplier and the fully bit-serial multiplier. The advantage of the fully parallel array multiplier is that all of the product bits are produced at once which generally results in a faster multiplication rate. The multiplication rate for this adder is simply the delay through the combinational logic. However, parallel multipliers also require a large amount of area to implement. Bit serial multipliers on the other hand

generally require only 1/Nth the area of an equivalent parallel multiplier but take 2N bit times to compute the entire product (N is number of bits of multiplier precision) [6]. This concept is illustrated in Figure 3.2. A new trend is to incorporate a limited number of hardware multipliers within the FPGA. For example, AT&T incorporates a 4 X 1 multiplier in each programmable function unit of its ORCA FPGA family.

Innovative techniques which avoid conventional multiplication in computing FIR filters and other DSP algorithms have been investigated by a number of researchers. For example use of distributed arithmetic techniques have been reported [12], which makes extensive use of look-up tables, an approach which allows a considerable savings in chip resources.

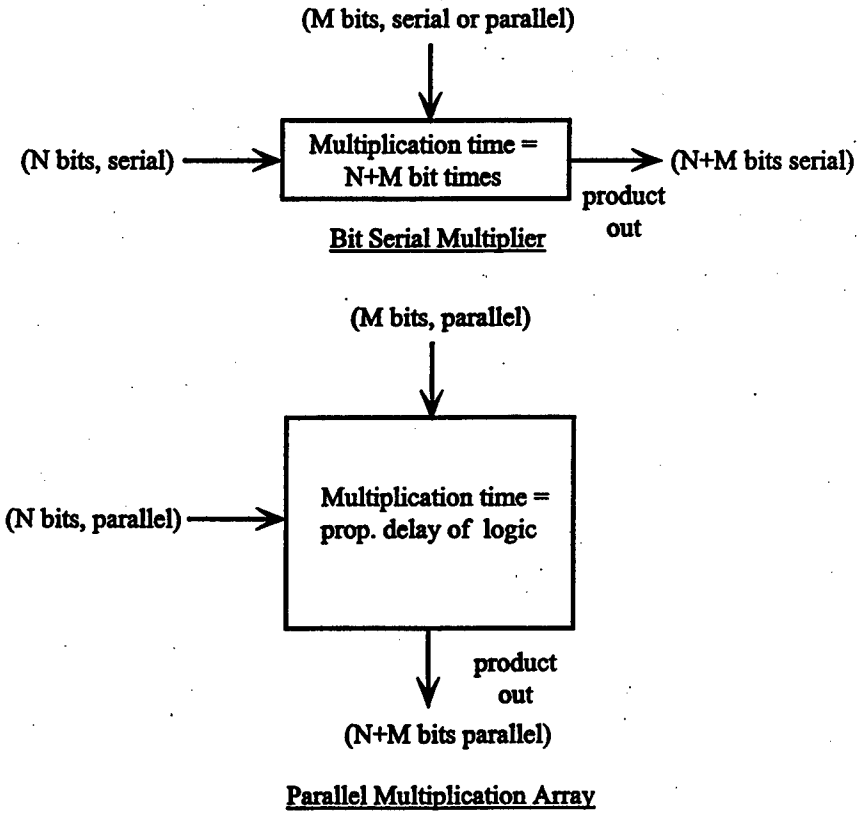


Figure 3.2 - Block Diagram of Basic Multiplier Alternatives

Distributed Arithmetic are computational algorithms that perform multiplications with look up tables. This algorithm is generally used to perform important DSP filtering and frequency transforming functions. Since most of the recent architectures of the programmable logic have supported the look-up table methodology distributed arithmetic has become very popular.

Distributed Arithmetic differs from conventional arithmetic only in the order in which it performs operations. Take for example a four-product MAC function that uses a conventional sequential shift and add technique to multiply four pairs of numbers and sum results. The four-multiplication are performed simultaneously and the results are then summed when the products are complete. This method of implementation requires n-clock cycles for data sample of n-bits. Hence , the processing clock rate is equal to data rate divided by the number of data bits. During each data clock-cycle, the four-multipliers simultaneously create four-product terms, that eventually are summed into the output. The distributed arithmetic differs from this process by adding the partial-products before, rather than after, the bit-weighted accumulation.

By using Distributed Arithmetic, the operations are reordered. The reordering reduces the number of shift-and-add circuits to one, but does not change the number of simple adders. Distributed arithmetic is of two types the serial and parallel distributed arithmetic.

Distributed arithmetic is useful in filtering applications, where the coefficients are constant. Adders and AND gates are made use of to implement multiplication with coefficients. But in distributed arithmetic the AND functions and adders are replaced with look up tables (LUT). If a single bit is made use of to access the LUTs then it is called serial distributed arithmetic, where the incoming sample of the signal stored in a shift register and a bit at a time is shifted out. The other type of distributed arithmetic is Parallel distributed arithmetic (PDA), where the number of bits used to access the LUTs are more than one. The overall performance of PDA is better than SDA as in the former case the number of bits processed during each clock cycle is increased.

This section provides a brief overview of the popular multiplication techniques, and presents the advantages and disadvantages of each technique in terms of speed, complexity, and area requirements for FPGA implementation.

3.3.1 Conventional Arithmetic

We will first examine how binary multiplication is carried out. Assume that we want to multiply two 4-bit numbers as shown below:

```
Multiplicand   1001
Multiplier     0110
------------------------
                0000
               1001
------------------------
              10010       Partial Product
             1001
------------------------
             110110       Partial Product
            0000
------------------------
Product    00110110
```

As shown above, each bit of the multiplier is multiplied with the multiplicand producing partial products which are appropriately weighted and added together to give the final product. As a typical example, consider a four-product Multiply-Accumulate (MAC) unit used to multiply four pairs of numbers and sum the results (a typical application of this – with slight modifications – would be in FIR filters). Figure 3.3 below illustrates a hardware implementation of this MAC unit.

Each MAC block multiplies the coefficient Y by one bit of the data A from the shift register using an AND operation, thus forming partial products. The output of the AND gate feeds an adder and a register. The second input of the adder is fed with the output of the register which represents the previous partial product divided by two (shifted to the right) thus appropriately weighting the partial products. In summary, for each clock cycle, each data bit (Ai) is multiplied with the coefficient (Y), and the register's output (previous partial product) is right shifted and added with the current partial product. For the whole four-product unit, the four multiplications (data A, B, C, D

times the coefficients Ya, Yb, Yc, Yd) are carried out simultaneously, and the results are added together when the products are complete.

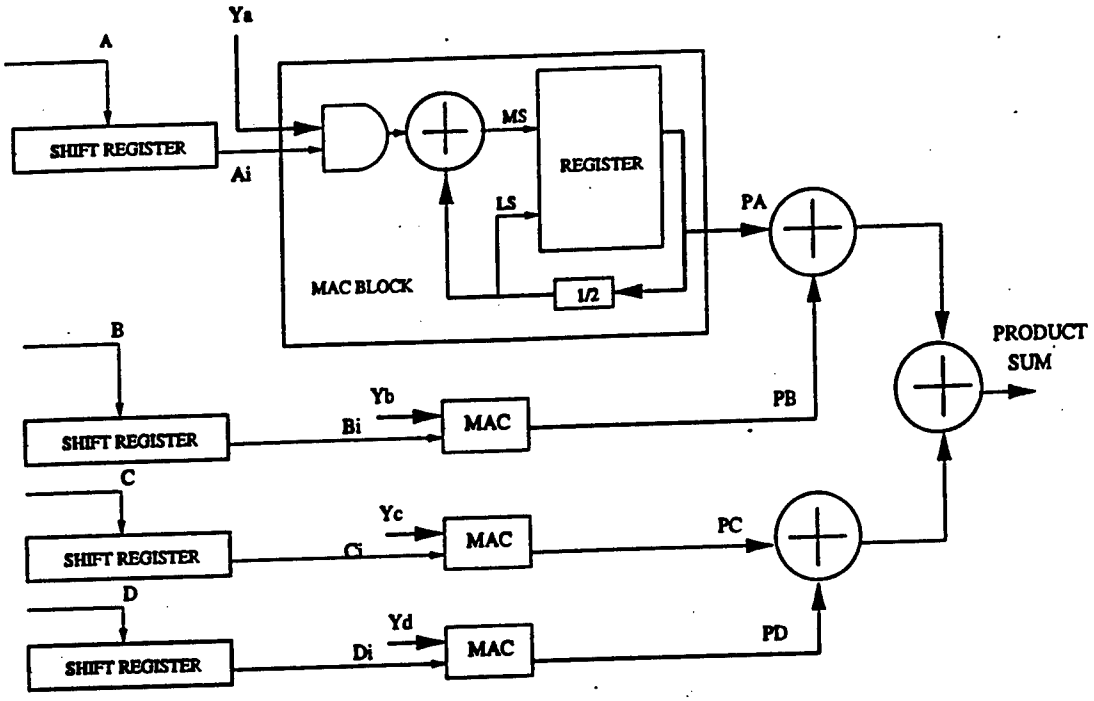


Figure 3.3 - Four-Product MAC Unit with Conventional Arithmetic

This procedure requires *n* clock cycles for data samples of *n* bits. During each cycle, the four multipliers simultaneously create four partial products PA, PB, PC, PD that are summed together to form the output. In terms of speed, this arithmetic technique is not efficient because to process the entire data sample, the processing rate is equal to the clock rate divided by the number of bits in the data sample. Also, in terms of surface real estate needed on an FPGA, the MAC block in each data path increases the area occupied on the chip.

3.3.2 Serial Distributed Arithmetic (SDA)

The Serial Distributed Arithmetic technique is more efficient than the conventional approach. SDA differs from the conventional arithmetic only in the order it performs

operations. That is, in the SDA approach, the partial products generated by multiplying each data bit with the coefficient, are added before, rather than after, the bit-weighted accumulation in the MAC block.

By reordering the operation as described above, the number of shift-and-add (MAC block) circuits is reduced to one, resulting in tremendous saving in FPGA area. A Serial Distributed Arithmetic circuit is shown in Figure 3.4.

Further improvement in that circuit can be achieved if we consider that in many DSP applications the coefficients Y are constants (e.g., a standard FIR filter). In such a case, the AND operations and the adders depend only on the input data bits from the shift registers. As stated before, these AND gates and adders generate the partial products and add them together before the weighting accumulation. We can therefore pre-compute all possible partial products (sums of the coefficients), and place them in a Look-Up-Table (LUT) which replaces all the AND gates and adders. Therefore, the input data bits from the shift registers, will just serve as indexes (address inputs) of the LUT. The LUT circuit for a Serial Distributed Arithmetic approach is shown in Figure 3.5.

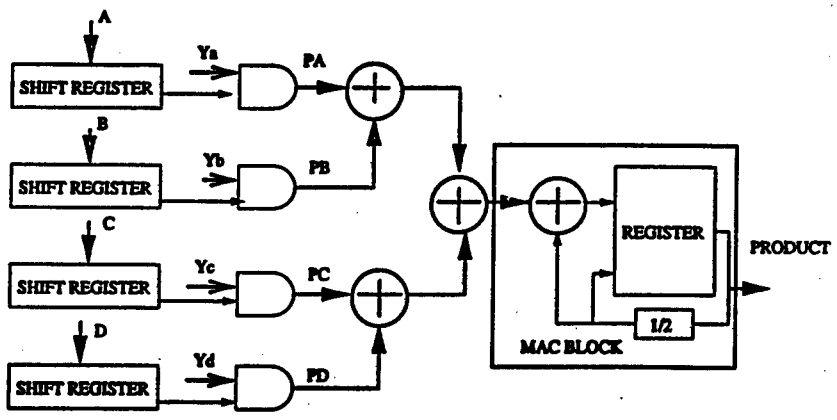


Figure 3.4 - Four-Product MAC using SDA.

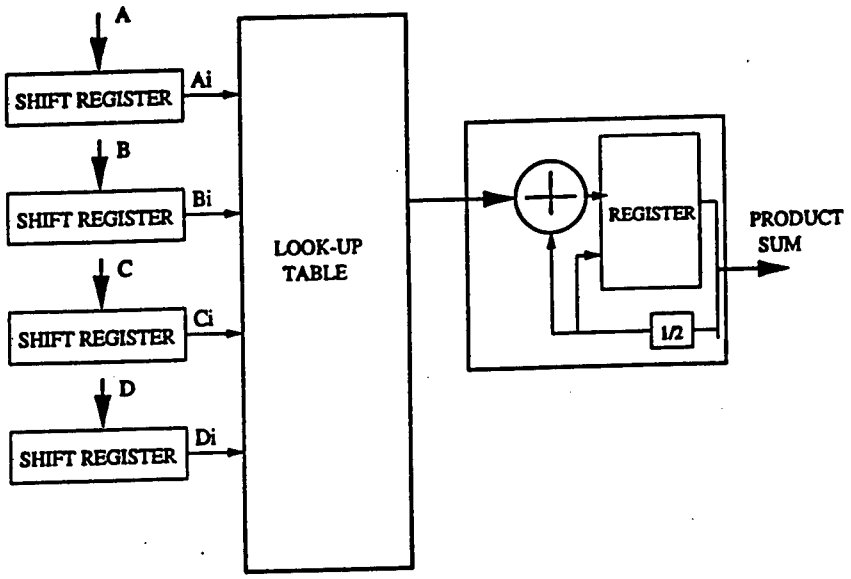


Figure 3.5 - A LUT Replaces the AND Gates & Adders in SDA

The Look-Up Table contains 16 memory locations (4 address inputs) which hold all possible sums of the coefficients. In our example where we have four coefficients, the LUT width should be equal to the coefficient width plus two extra bits to allow for word growth. The data bits which address the LUT, determine which sum of coefficients will be placed in each LUT memory location. For example, when all four data bits are 1, the sixteenth memory location (1111) will contain the sum of all four coefficients. Any data bit that is zero eliminates the corresponding coefficient from the sum. Table 3.1 below illustrates the contents of the 16-Word LUT.

By implementing the LUT in place of the AND gates and adders, the speed improvement is significant because we avoid adding the partial products in real time. Rather, all the partial product sums are pre-computed and are readily available at the output of the LUT. In summary, the optimized Serial Distributed Arithmetic technique using LUT is much more efficient compared with Conventional Arithmetic, both in terms of speed (LUT) and area occupation (only one shift-and-add circuit is needed since partial products are added before, rather than after, the bit-weighted accumulation).

TABLE 3.1 - CONTENTS OF 16-WORD LUT				
Ai	**Bi**	**Ci**	**Di**	**LUT Contents**
0	0	0	0	0
1	0	0	0	Ya
0	1	0	0	Yb
1	1	0	0	Ya + Yb
0	0	1	0	Yc
1	0	1	0	Ya = Yc
0	1	1	0	Yb = Yc
1	1	1	0	Ya + Yb + Yc
0	0	0	1	Yd
1	0	0	1	Ya + Yd
0	1	0	1	Yb + Yd
1	1	0	1	Ya + Yb + Yd
0	0	1	1	Yc + Yd
1	0	1	1	Ya + Yc + Yd
0	1	1	1	Yb + Yc + Yd
1	1	1	1	Yz + Yb + Yc + Yd

3.3.3 Parallel Distributed Arithmetic (PDA)

Increasing the number of data bits processed for each clock cycle from one to two results in twice the throughput. However, there is a trade off here in terms of FPGA area used, which is increased dramatically since more CLBs are employed. Figure 3.6 below illustrates a block diagram of a 2-bit PDA circuit.

In the 2-bit PDA approach, the data word is split into even and odd bits and is fed into two parallel shift registers which are half the bit width of the shift registers used in the SDA approach. Therefore, for each clock cycle we process two bits of the data sample instead of one bit as in the SDA technique. However, now two LUTs are needed, one to store partial products of the even bits, and the other for the odd bits. Also, an adder is

required to sum these two partial products, and also the shift-and-add circuit (scaling accumulator) at the end, shifts the data by two-bits (divide by 4) for scaling of the final product. With this approach, more than two parallel bits can be processed at a time. In fact, this concept can be extended to a fully parallel PDA circuit in which all n bits of a data sample are processed on each clock cycle, resulting in increased throughput. However, this would result in a very significant increase in hardware area, so the number of parallel bits sampled at one time should be increased only to meet the required speed performance.

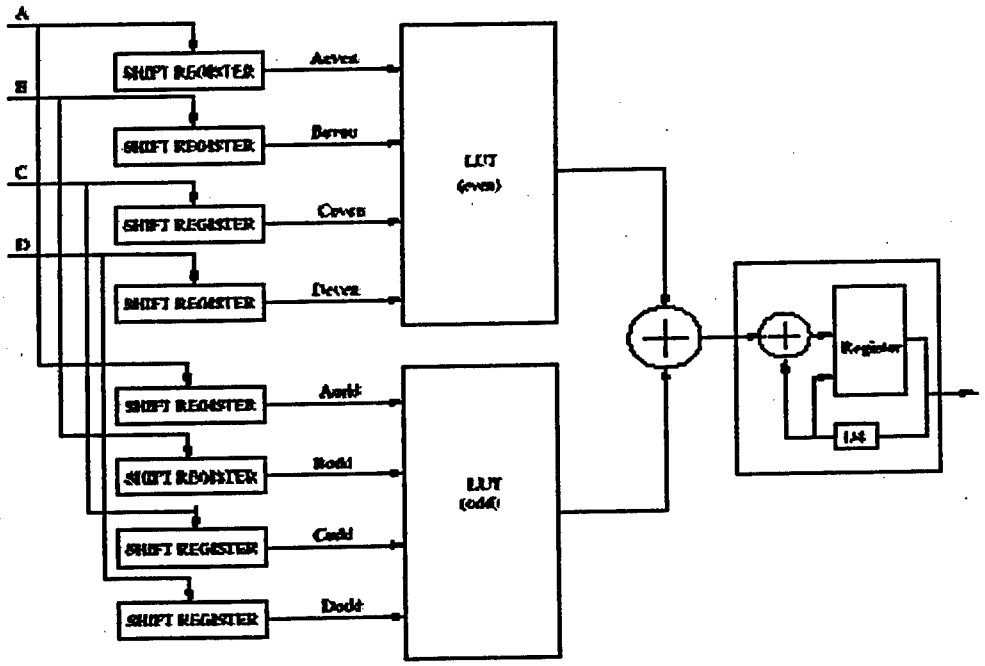


Figure 3.6 - 2-bit Parallel Distributed Arithmetic.

3.3.4 Constant Coefficient Multiplier using LUT

In the preceding sections, we focused on distributed arithmetic methods for binary multiplication. In this section, a different approach is presented for multiplying data with a fixed coefficient. This method is fully parallel, and uses Look-Up-Tables to store all

possible partial products of data times coefficient. To understand this hybrid technique of multiplication, consider how hexadecimal multiplication is performed:

```
  55
x 2B
----
  3A7  = Bx55
+0AA0  = 2x55
----
0E47
```

Now, having the 55h multiplication table on hand, makes the multiplication of 55h with any number very easy. Table 3.2 below shows a 55h multiplication table.

TABLE 3.2: MULTIPLICATION TABLE	
0x55 = 000	8x55 = 2AB
1x55 = 055	9x55 = 2FD
2x55 = 0AA	Ax55 = 352
3x55 = 0FF	Bx55 = 3A7
4x55 = 154	Cx55 = 3FC
5x55 = 1A9	Dx55 = 451
6x55 = 1FE	Ex55 = 4A6
7x55 = 253	Fx55 = 4FB

The fixed coefficient multiplication table can be stored in a ROM Look-Up-Table. In our example we assumed that both data and coefficients are 8-bits wide. The data should be split in two 4-bit segments, and each segment is used to address one of the two ROM LUTs as shown in Figure 3.7 below.

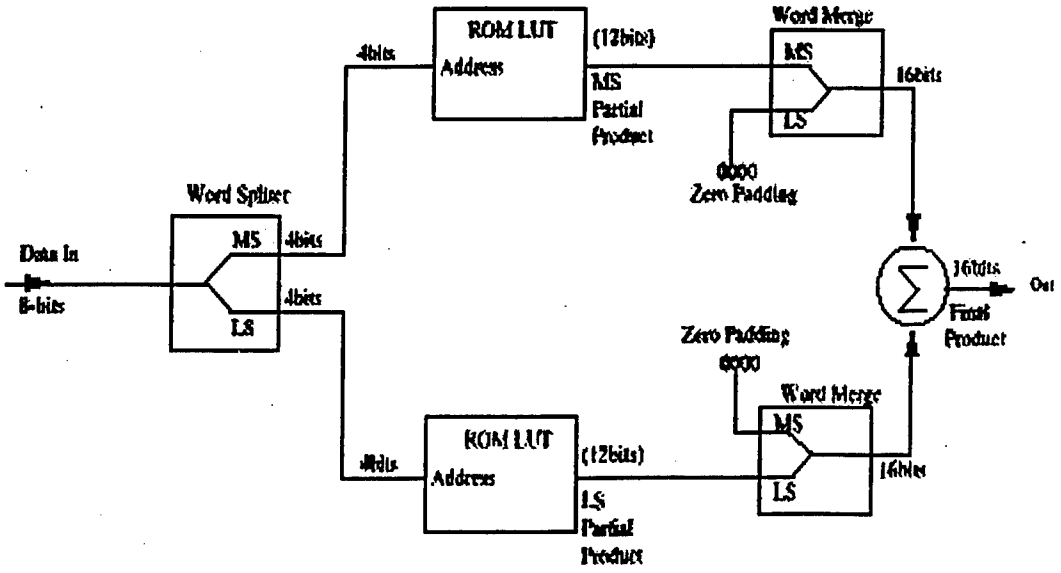


Figure 3.7 - Constant Coefficient Multiplier using LUT.

Each ROM LUT is a fixed coefficient multiplication table and contains 16 entries which represent all possible partial products from *0 x "coefficient"* up to *F x "coefficient"*. The upper (Most Significant) LUT contains all the partial products of the 8-bit coefficient times the most significant 4-bits of the incoming data. Similarly, the lower (least significant) LUT contains all the possible partial products of the 8-bit coefficient times the least significant 4-bits of the data. Therefore, at the output of each LUT we have a 12 bit partial product (4 bits data + 8 bits coefficient). Each 12 bit partial product is appropriately zero padded (for scaling) and then summed to produce the final product at the output of the multiplier. This approach is very efficient in both speed and area required on the FPGA. The data samples are processed in parallel resulting in very high processing rates. We could have used only one ROM LUT and apply all the eight bits of the data on it, but this would consume more space on the FPGA, because it would need a ROM with $2^8 = 256$ memory locations. By splitting the data in two, each ROM LUT has now $2^4 = 16$ memory locations resulting in area saving.

3.4 Using FPGAs for DSP Applications

The FPGA has recently generated interest for use in DSP systems because of its potential to implement an infinite variety of custom hardware solutions while still maintaining the flexibility of a conventional programmable device [6]. Although DSP microprocessors have complete algorithm flexibility, their performance is limited because algorithms are implemented by sequential MAC operations, as previously described. Additionally, DSP microprocessors have an overhead for reading in the operands and writing the result through a single data port. Therefore, a DSP microprocessor may require at least four cycles (i.e., read, multiply, add and write) to perform the simplest of algorithms, resulting in 10 MIPS performance from a 40 MIPS processor [1].

Because DSP algorithms are optimally mapped to the device architecture, FPGA performance can significantly exceed DSP processor performance. For example, a DSP microprocessor can implement an 8-tap FIR filter at 5 Msps. An FPGA can implement the same FIR filter at 100 Msps [1]. FPGAs will never completely replace general purpose DSP processors, however. Current generation programmable logic addresses only the fixed point DSP portion of the market. General purpose DSPs still dominate in floating point performance. Also, general purpose DSP processors utilize familiar software methods, while using programmable logic requires a completely different approach on the part of the DSP designer. Implementing DSP functions in FPGAs provide the following advantages over conventional DSP hardware:

a. *Parallelism* - Using FPGAs can lead to significantly higher performance than a typical DSP processor for some applications.

b. *Efficiency* - An FPGA can be optimized for specific algorithms, thus achieving the performance of hardware with the flexibility of software.

c. *In-circuit Reconfigurability* - Permits the algorithm or function to be changed while operating in-circuit. An additional benefit of FPGAs over ASICs is that they can be

reprogrammed on the fly in the system. Consequently, a single FPGA can implement different DSP functions at various times in a system to boost overall performance.

d.. *Adaptability* - A device that can implement large internal RAM blocks can be used to implement real-time adaptive functions at a throughput that cannot be matched by conventional DSP solutions.

3.4.1 Pipelined Architectures

Since the FPGA CLBs contain flip-flops, they are used for storing and delaying the signal. The purpose of pipelining is to increase the speed at which the system operates by decreasing the delay of the critical path of the system. The Figure 3.8 below illustrates the concept of pipelining.

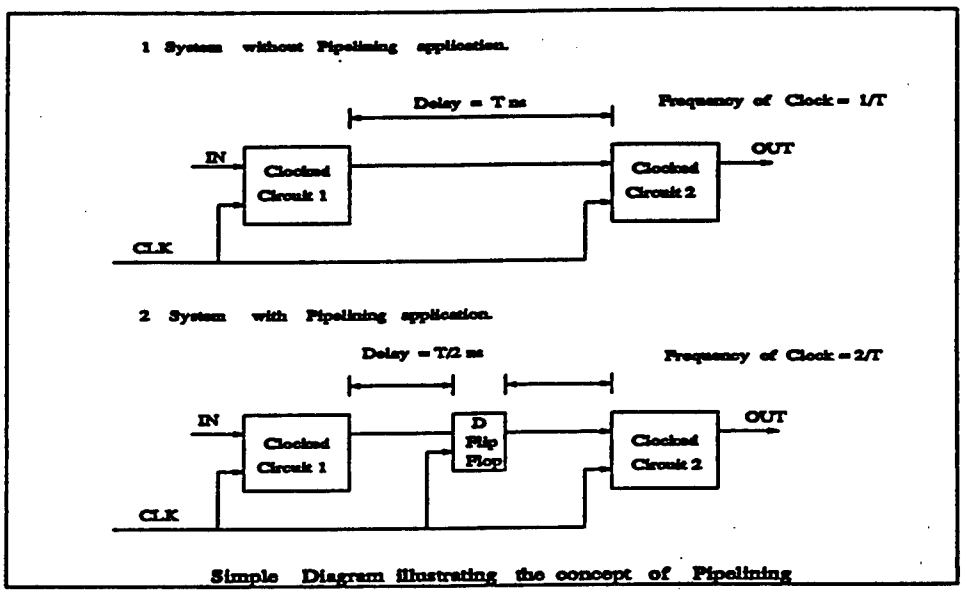


Figure 3.8 - The Pipelining Concept

In this figure we have two clocked circuits which are driven by the same clock. Hence the speed of operation depends on the delay that separates the two clocked circuits. If the delay is more than the clock period, then the frequency of operation is limited by the delay between them. To increase the frequency, register elements can be

include in the delay path thereby decreasing the delay in the path and increasing the speed of operation. This is shown in part two of Figure 3.8. Pipelining is a trade off between speed versus resource utilization. Using too many pipelined stages in the system results in an enormous consumption of hardware resources; and in designs where area and power are of main concern, pipelining may not be advisable.

3.4.2 Design of Data Paths

The data path is that part of the system that performs the data processing operations, so the design of the data path in any system design is very important. The speed of the system increases if there are a number of data paths performing the same function in parallel, however this kind of system design requires lots of hardware resources so it is important to decide whether or not the data path will be shared.

A critical factor in designing the data path is the precision requirements of the design. If very high precision is required, such as needed in DSP applications, then the data path should implement a floating point processing of the signals. This would lead to excessive use of resources since two registers are required to store a floating point number in the data path. Therefore, all of the present designs make use of fixed point data paths.

Though fixed point data path design leads to using fewer hardware resources, the accuracy of the results obtained is less, when compared to the floating point designs. By making use of fixed point algorithms, the number of resistor elements can be reduced, which in turn can be used to implement parallel data paths and pipelining to improve the speed at which the design operates.

3.4.3 Routing Delays

Routing Delays play a major role in the hardware design since they limit the operating speed of the system. Very complex designs often suffer from routing delays because of the large number of logic circuits implemented in the device, resulting in less space

available for achieving optimal routing. This is a concern for DSP systems as many arithmetic operations need to be performed. Arithmetic elements are required which typically occupy a large percentage of the available resources, which affects the routing of the design.

The recent trend in FPGAs is to use common function (or macro) blocks to aid in developing systems with lower routing delays. For example the Xilinx FPGAs supports XBOLX modules such as adders, subtractors, incrementors, decrementors etc. , which are used extensively to implement arithmetic functions on Xilinx FPGAs with very low routing delays. Thus by decreasing the routing delays we can increase the frequency of operation of the system.

3.5 FPGA Applications in Software Radio

The DSP Functions that FPGAs do best are those requiring high sample rates and short word length. They are especially suited for FIR filter designs employing lots of filter taps and fast correlators. The lookup table architecture of FPGAs provides a fast and efficient way to build correlators [3]. More taps can be added to the parallel filter with only a small performance tradeoff with additional parallel silicon resources. In contrast, DSP processors exhibit a linear decrease in performance as the number of taps increases (Table 3.3). An 8-tap, 8-bit FIR filter implemented on an Altera device needs only 80% more silicon than one 8 x 9 bit fixed multiplier (Table 3.4)[1].

TABLE 3.3 - FULLY PARALLEL 8-BIT FIR FILTER (FLEX 8000A FPGA)

# of Taps	*Performance (MSPS)*	*Equivalent MIPS (DSP Processor)*
8	104	832
16	101	1,616
24	103	2,472
32	105	3,360

TABLE 3.4 - SILICON RESOURCE COMPARISON		
Function	*Inputs & Outputs*	*Flex 8000A Logic Cells*
FIR Filter	8-bit data, coeff 17-bit output	296
Fixed Point Multiplier	8 x 9 bit data 17 bit output	164

Table 3.5 shows the performance of multipliers implemented on the Xilinx 4000 family. Note that parallel multipliers require a larger proportion of the device, while bit serial implementations are slower. The first number in the Multiplier Speed column for the bit-serial multipliers is the clock speed, while the second number is the multiplier speed.

TABLE 3.5 - XILINX 4000 SERIES FPGA MULTIPLIERS			
Type of Multiplier	# CLBs	% of FPGA	Mult. Speed
8 bit unsigned (parallel)	64	16%	8.54 MHz
16 bit unsigned (parallel)	242	60%	3.8 MHz
8 bits unsigned (bit-serial)	17	4%	73.1/4.6 MHz
16 bit unsigned (bit-serial)	33	8%	62/1.9 MHz

FPGAs can efficiently implement IIR filters. For example, a lookup table based vector multiplier can be used to create a complete second order section of an all pole analog filter. The vector multiplier requires the same resources and operates at the same speed as a fixed point multiplier. A Butterworth filter can run at a rate of 25 Msps and require only 139 logic cells [1].

Altera has developed high speed FIR filter megafunctions that are optimized for their own FPGA structure. These filters can be implemented in parallel or serial form allowing a tradeoff between silicon resources and performance. Parallel filters can perform at rates up to 100 Msps enabling digital processing of RF-IF data. Serial filters

require less logic and still perform at 5 to 6 Msps. In a Spread Spectrum RF modem application, an Altera FPGA can implement the receiver's correlation filter function at a chip rate over 60 MHz. A DSP processor can perform the remaining tasks, such as quadrature phase shift key (QPSK) demodulation. The resulting DSP application can deliver six times the data rate as the DSP processor alone.

Chapter 4

Rapid Prototyping Concepts

Designing with FPGAs requires computer assistance at almost every stage of the design including detailed specification, simulation, placement and routing. The use of schematic capture based CAD tools is a common approach to the design of custom logic devices using FPGAs. This process is often combined with logic level simulation to verify a specific design. One method of increasing the range of architectural solutions that a designer may explore in a reasonable time is to specify the DSP system with a hardware description language (HDL) [10]. This steps the design process up one level and allows a generic functional description of the target system which can be further simulated or implemented directly onto an FPGA after the HDL code is converted using the FPGA manufacturers software.

In DSP applications, arithmetic circuitry for operations such as addition, subtraction and multiplication are commonly required. These arithmetic circuits can be designed and implemented by employing user-generated or manufacturer-provided sub-circuits, which can be reused. However, as these designs can only be simulated at the logic gate level, it is difficult to verify the functional performance of the algorithms being implemented. It is particularly difficult to determine the potential undesirable side effects of finite precision arithmetic, as this may require that large data sets be simulated and

translated from numerical values to logic levels and vice versa [10]. However, new software tools are being developed which raise the design process to yet another level, allowing the designer to begin at the system level.

Simulation tools such as Cadence's Signal Processing Worksystem (SPW) now have features which allow the engineer to design hardware logic systems and DSP fixed point systems using the traditional block diagram functional description of the circuit. This design is then immediately converted into a hardware description language. Other SPW tools allow the design to be simulated via the HDL description of the system and then linked into a manufacturers software tools which support specific devices. Most manufacturers, in the interest of making their product more attractive to their customers, have developed a set of stock logic elements which can be reused within their device to assist the engineer in quickly achieving any design. Once suitable design tools and automatic methods are perfected, designers and programmers will be able to create custom hardware circuitry and pipelines to suit the problem at hand - the term 'soft hardware' suggests that hardware will become as readily created and malleable as software. In a practical sense this will mean that the turn-around time for custom hardware will be just as short as software development is today.

4.1 Design Flow

Figure 4.1 illustrates the flow of rapid prototyping. The flow of the design is from the functional description of the system to the hardware implementation. The functional description of the system is implemented in either SPW (Signal Processing WorkSystem) or in VHDL. The functional description is usually is at the system level, where algorithm functions are of primary interest. A reconfigurable flat-form is used to aid the flow of rapid prototyping, and commercial CAD tools are used to integrate the design.

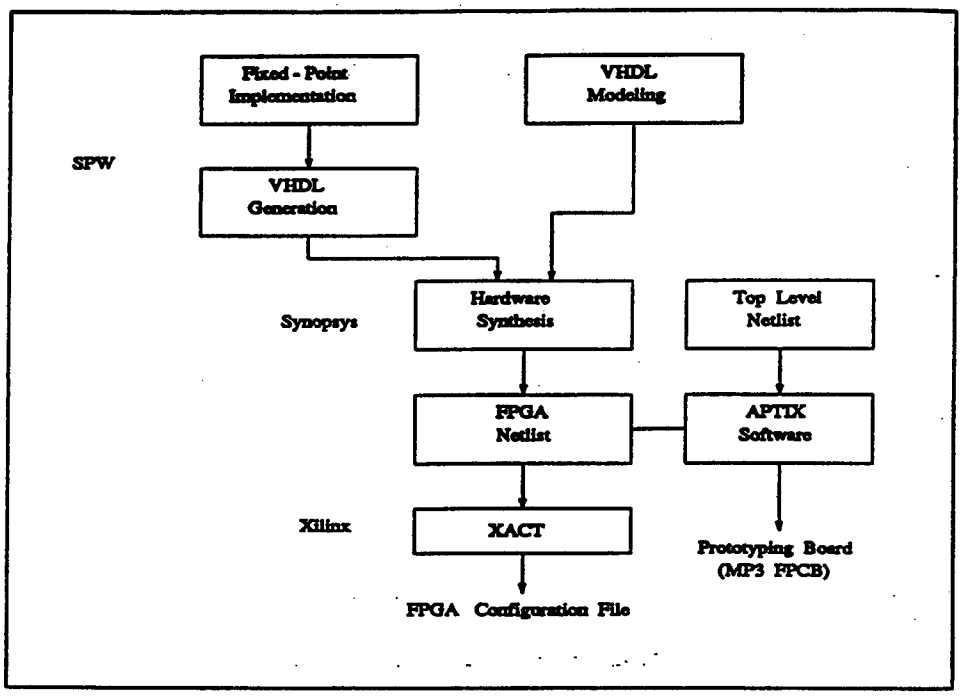


Figure 4.1 - Block Diagram of Rapid Prototyping Flow

4.1.1 Functional Description using SPW and VHDL

The system to be implemented is functionally described using the Hardware Design System (HDS) of SPW. The algorithm is designed using the blocks available in HDS. The functional description of the system is completed using the fixed point blocks to control the use of hardware resources which are quite limited in reconfigurable hardware. The algorithm developed is simulated in the SPW environment before the implementation is completed to make sure that the functional description of the system is correct. To implement the design on hardware, the HDS component of SPW provides a link which generates VHDL code for the system designed in HDS. The generated VHDL code is used for synthesis in order to implement the system on the targeted device.

The other way to functionally describe the system is with VHDL. The advantage of using hand-coded VHDL, rather than using VHDL code generated by the schematic tool, is that for very large and complex designs, schematic capture of the system

becomes difficult and impractical. Also, the hand-coded VHDL is very flexible in that, if certain enable signals are required for flip-flops, counters etc., then the blocks provided by HDS can be modified and customized. It is also true that code generated from a schematic tool tends to be less efficient and require more hardware resources that would be required if the system were hand-coded with VHDL.

4.1.2 Simulation and Synthesis

The next step in the design flow is to simulate the design using standard tools that verify the functionally of the design. If the simulation results are satisfactory then the design is synthesized using standard tools that target the design to specific FPGAs. The systems designed using SPW can be simulated in that environment. But hand-coded VHDL, need to be compiled and simulated using VHDL simulation tools such as Mentor, Synopsys, etc. Figure 4.2 shows the flow followed during the simulation of the system.

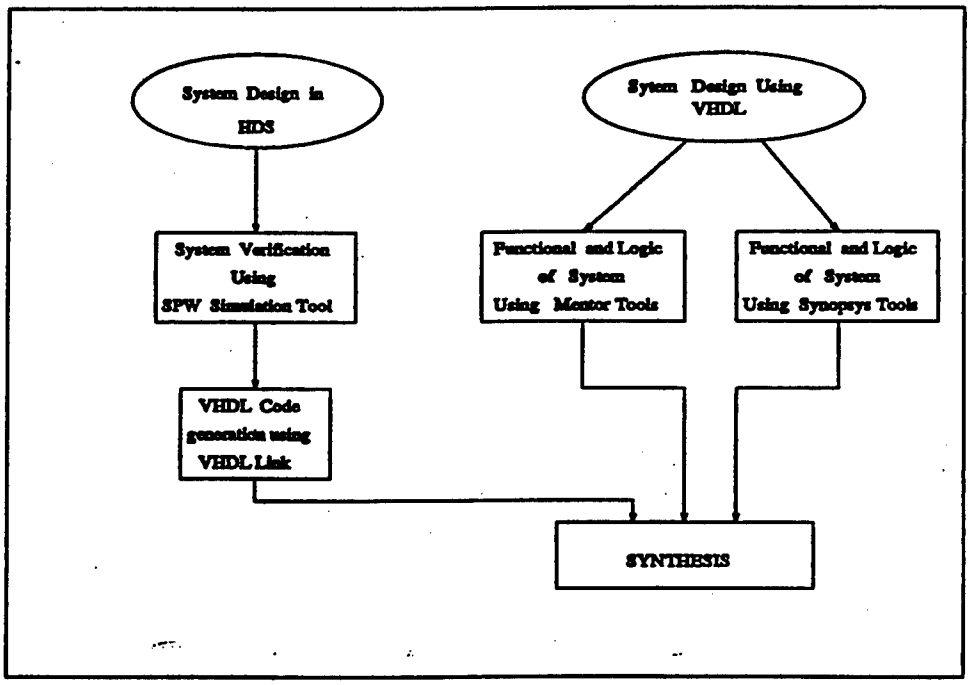


Figure 4.2 - Block Diagram of Simulation Flow

Synthesis is the procedure that makes possible the implementation of the system on the targeted hardware. It is also one of the key factors which aids in rapid prototyping. Synthesis tools translate the high level design into gate and register levels which the routing software can process. The synthesis tools generate netlist files that are used by the routing software to generate files that are used to define the hardware physically. For example the Synopsys FPGA compiler generates a top level netlist which is used by the Xilinx software to partitions places and route the design. Figure 4.3 shows the flow of the synthesis procedure.

High level system design is gaining popularity since it allows the designers to describe systems at a high level using schematic capture. The high-level design approach reduces library and technology dependence, enabling re-targeting to other libraries, such as an FPGA library, with greater ease.

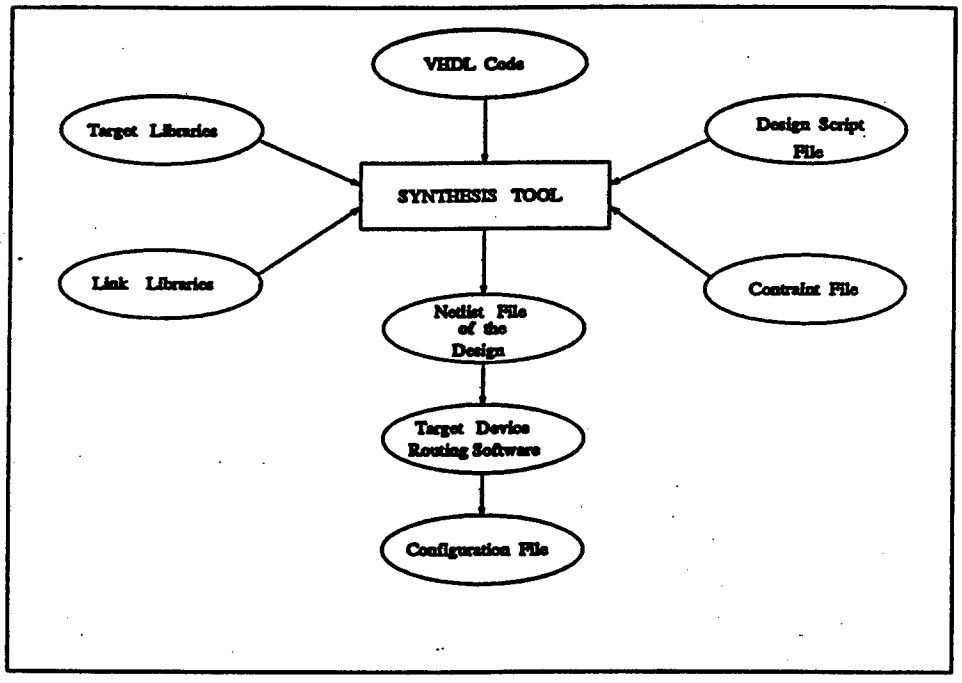


Figure 4.3 - Block Diagram of Synthesis Procedure

4.1.3 Implementation

Implementation occurs at two levels: first is at chip level where the entire system is partitioned into smaller submodules, and these modules are implemented on the target FPGA. The second level of implementation is at the system level, where all submodules are integrated and the entire system is tested.

Chip level implementation on the FPGA is an important part of the rapid prototyping process. Synthesis tools provide the designers with netlist files of the submodules to be implemented. During the synthesis, constraints are provided to meet design specifications. The reconfigurable nature of FPGAs also aids the rapid prototyping approach. Every FPGA has its own placement and routing software to map the design. The software partitions the design and places the logic into the configurable logic blocks and finally does the routing of the entire design. The software generates a bit file to configure the FPGA device.

System level implementation is accomplished using the rapid prototyping board, known as the field programmable circuit board (FPCB). The FPIC (field programmable interconnect component), are programmable interconnect components which form the core of the programmable circuit board. For example, the Aptix MP3 reconfigurable board has three programmable interconnect components used for routing purposes. The MP3 board also supports diagnostic programmable interconnect components which aid in viewing signals on the diagnostic instruments. The FPIC is configured through a Host Interface Module (HIM) , which transfers data from a workstation to program the FPIC. A Stand-alone Program Module (SPM) can be utilized to perform the same function without a workstation.

The FPCB provides fully automated downloading of configuration data to both FPGAs and FPIC devices. As the board supports the combination of FPGAs with standard components(memory, DSP and microcontrollers) makes the MP3 uniquely suited for DSP system prototyping.

4.1.4 Verification

The rapid prototyping environment helps in debugging and verifying very complex systems. As stated earlier, the FPIC devices used on the FPCB are of two types: one is used for routing purposes and is designated as an FPIC(R) device . The other type of device is the diagnostic device, which is used for probing, debugging and verifying signals in the design and is designated as FPIC(D). These FPIC(D) devices can be connected to the logic analyzer with help of diagnostic pads. The software for the board is called AXESS, and it programs both the diagnostic FPIC devices and logic analyzer.

This setup provides a very powerful debugging capability, since each signal that appears in the system level netlist can be routed through one or more FPIC(D) devices and viewed on the logic analyzer. The signals to be viewed are selected with the help of diagnostic device interface provided by the software. The software automatically programs the diagnostic FPIC device to display the selected signals on the logic analyzer. At the same time, the diagnostic interface facility configures the logic analyzer. The diagnostic interface provided by the reconfigurable board software does the channel assignment and labeling of the waveform displays.

4.2 Prototyping

In traditional prototyping approaches, the design is mapped to a technology that allows speeds such that all interfaces to targeted applications can operate in real time. But rapid prototyping provides flexibility for system emulation to explore architectural and implementation alternatives available for achieving the desired system function.

Prototyping was commonly done using custom printed circuit boards and wire wrap technologies until the design complexity became too large to make these approaches feasible. The new technologies such as FPIC, FPCB, and FPGA have created a new path that enables mapping of complex logic into programmable hardware which can meet the real-time operating frequencies of DSP applications. The main aim

of rapid prototyping is to design, implement and verify systems quickly, hence aiding in bringing products faster to the market when compared to traditional prototyping methods.

Chapter 5

Implementation Case Studies

This chapter describes the DSP algorithms designed and implemented on the Aptix MP3 reconfigurable circuit board. The algorithms designed here are digital filters, fast Fourier transforms and communications modules such as the PN sequence generator. All designs developed are targeted to Xilinx FPGAs and use a reconfigurable hardware platform (the Aptix MP3) to illustrate the concept and the speed with which systems can be designed and implemented using rapid prototyping.

The digital filters described in this chapter are designed using both a locally developed FIR CAD tool and the Signal Processing Worksystem (SPW) software. The FIR CAD tool generates XNF files for the filter, but filters designed using SPW employ commercial tools to generate the filter XNF file. Other systems examined in this chapter include an eight point FFT, and a scalable FFT (16, 64 points). The scalable FFT algorithm designs use a standard memory chip. The following sections give the details of the designs developed and their implementation.

5.1 FIR Filter Design using CAD Tools

Finite Impulse Response (FIR) filters play an important role in the design of practical discrete-time systems. At the heart of a FIR filter lies the multiplication function, which introduces the coefficients of the filter in the design. Each filter tap has its own multiplier, which gives the product of the input data with the coefficient. When implementing a FIR filter on an FPGA, the multiplication function imposes a bottleneck on the speed, performance and area requirements of the design, therefore the designer should focus on enhancing the performance of these multipliers and hence of the whole design. Multiplication techniques include Shift-and-Add, Adder Tree, Logical Tree, multiplication by a power-of-two, and constant coefficient multiplier using Look-Up-Tables (LUT). The last technique is the key to high performance in FIR filters with fixed coefficients.

FIR filters are very useful in DSP applications because they are inherently stable and can be designed to exhibit linear phase characteristics.. The general Direct Form structure of an FIR filter is shown in Figure 5.1.

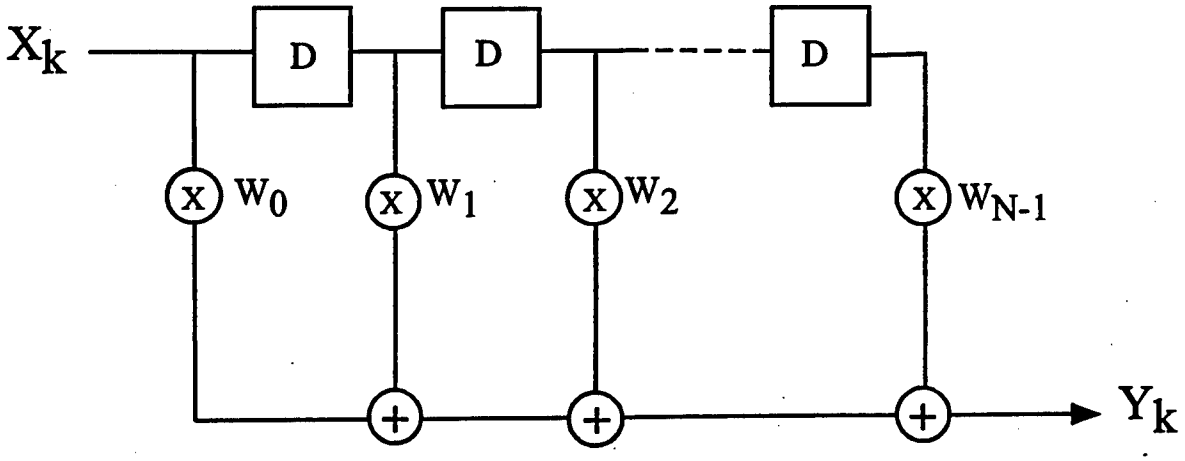


Figure 5.1 - Direct Form structure of an FIR filter

The algorithmic form of the Direct Form FIR filter is given by:

$$y(n) = h_0x(n) + h_1x(n-1) + h_2x(n-2) + \text{----} + h_{n-1}x(n-m)$$

For linear phase response of the filter, the impulse response must satisfy the symmetry condition:

$$h[M-n] = h[n] \quad \text{for } n=0,1,2.....M$$

The general Direct Form structure exhibits excessive redundant hardware and poor timing characteristics when implemented in hardware. An alternative inverted structure implementation is shown in Figure 5.2.

In the inverse structure, the data samples are applied to all the tap multipliers at the same time. Processing of the data samples is done in parallel and hence the overall timing performance is stabilized. Also, by exploiting the symmetric nature of the coefficients, we can reduce the number of coefficient multipliers needed by one half. Moreover, if we use Look-Up-Tables instead of regular multipliers, the time delay incorporated in the multiplication function is dramatically reduced. Each Look-Up-Table in each Tap contains all the possible products obtained when we multiply the specific tap coefficient with the incoming data. Therefore, the data bits are applied on the address input of the LUT (which is basically a ROM) and the corresponding "data * coefficient" product is obtained automatically on the output of the LUT

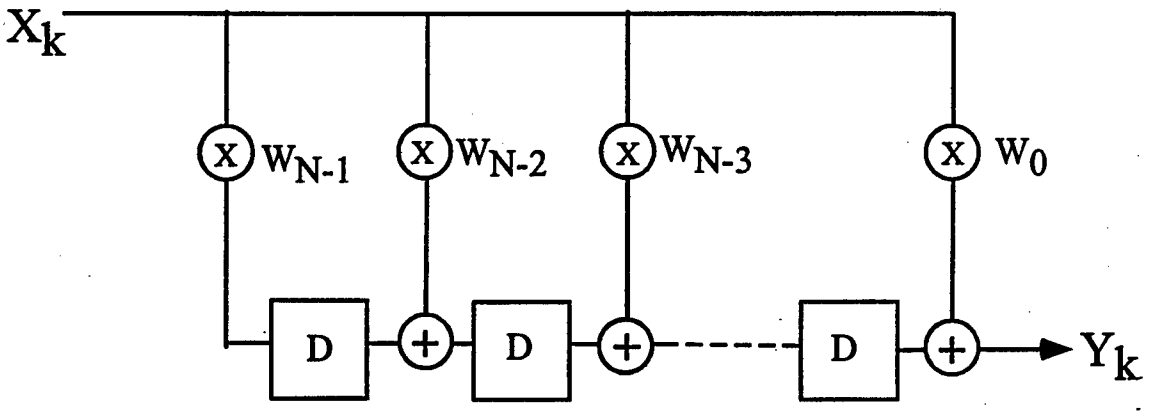


Figure 5.2 - The Block Diagram of Inverse FIR Filter Structure

An efficient FIR filter architecture suitable for FPGAs, is implemented using a locally designed CAD tool. The filter architecture uses coefficients that can be expressed

in the form of a sum or difference of two terms, both of which are powers of two. Multiplication in binary arithmetic by a power of two is simply a shift operation. Implementation of algorithms with multiplication may be simplified by using a limited number of power-of-two terms, thus decreasing the number of shift and add operations required. The FIR filter structure assumed by the FIR CAD tool is the inverted form.

To obtain good performance a small number of power-of-two terms are used in approximating each coefficient value of the filter and hence an optimization technique is carefully selected to derive the coefficient values.

5.1.1 Low Pass Filter

To design a low pass filter, the coefficients of the filter are obtained using the Filter Design System (FDS) feature of the Signal Processing Worksystem (SPW). The frequency specification is used to obtain the coefficients for implementing the filter.

The low pass filter design in this example consists of 11 taps, each represented by 10 bits. The frequency specification of the filter requires a passband cutoff frequency of 0.1Fs (where Fs is the sampling frequency) at 3dB attenuation and a stopband cutoff frequency of 0.15Fs at 18dB attenuation. The values of the taps are obtained using FDS. These coefficient values are used to develop a simulation model in the SPW Hardware Design System (HDS). The simulation results are helpful in validating both theoretical and implementation results. The simulation model consists of taps, adders and delay elements. The taps are implemented using the shift and add method in order to emulate the function of the FIR CAD tool in which multiplication of the samples are implemented by the same method.

Each tap is implemented using the shift block, adders or subtracters provided in HDS. The requirement is that the tap values be expressed as the sum of two, power-of-two terms. The required shift is provided as a parameter to the shift block. The low pass filter implementation is accomplished using the FIR CAD filter tool. The tool takes the coefficients provided from FDS and the number of bits required to represent them. The tool uses the number of coefficients, and the number of bits required in order to represent the coefficients and their values as input to a XNF (Xilinx Netlist File) file and a HDL

(Hardware Description Language) file. The XNF is then used to provide the bit file required to configure the FPGA according to the design requirements. Before generating the bit file, the HDL file is used to perform a check of the logic circuitry. Since the input required for testing the HDL file is the sampled input signal, the filter model is tested using SPW in the simulation mode.

As previously mentioned, the CAD tool takes coefficients which are expressed as a sum of powers of two. The reason for expressing the coefficients in powers of two is that the hardware implementation for multiplication is implemented by a shift and add operation on the input samples. Therefore, we are able to accommodate more useful logic on the FPGA and can therefore implement filters with a larger number of taps.

After the logic circuitry is tested using the HDL file and the bit file for the Low Pass filter is obtained, the design is implemented on the Aptix MP3 board. To implement the design on the board we need a top level netlist file and a clock which drives the FPGAs on the board. Also, an essential part of the implementation is the input/output FPGA which is used to route data to and from the board in order to provide signals to the filter. The other component on the board is a Xilinx FPGA, on to which the low pass filter design is downloaded.

The top level netlist file needed to implement the design is a file showing interconnections of the components. This is an input file to the Aptix MP3 software (AXESS). The AXESS software uses the top level netlist file to configure FPIC devices on the MP3 board according to the required interconnections between the components. After configuring the FPIC devices, the bit files for configuring the I/O FPGA and FPGA on which the low pass filter is implemented are downloaded. The I/O FPGA is used to buffer the input signals and route them to the target FPGA containing the low pass filter. The logic implemented on the I/O FPGA is simple buffering. The XLS (Xilinx Logic Synthesizer) is used to implement the logic.

The design of the low pass filter is tested using sampling frequencies of 1 MHz and 5 MHz. The results obtained from the implementation are compared with the simulation and theoretical results. A sine wave of varying frequency is used to test the low pass filter. According to theory, any frequency within 0.1Fs is reconstructed by the filter, but frequencies at and beyond stopband are by reduced by at least 18dB. The Figure 5.3

illustrates the theoretical result of low pass filtering. Figure 5.4 compares the magnitude response of the implementation with the theoretical response obtained from MATLAB.

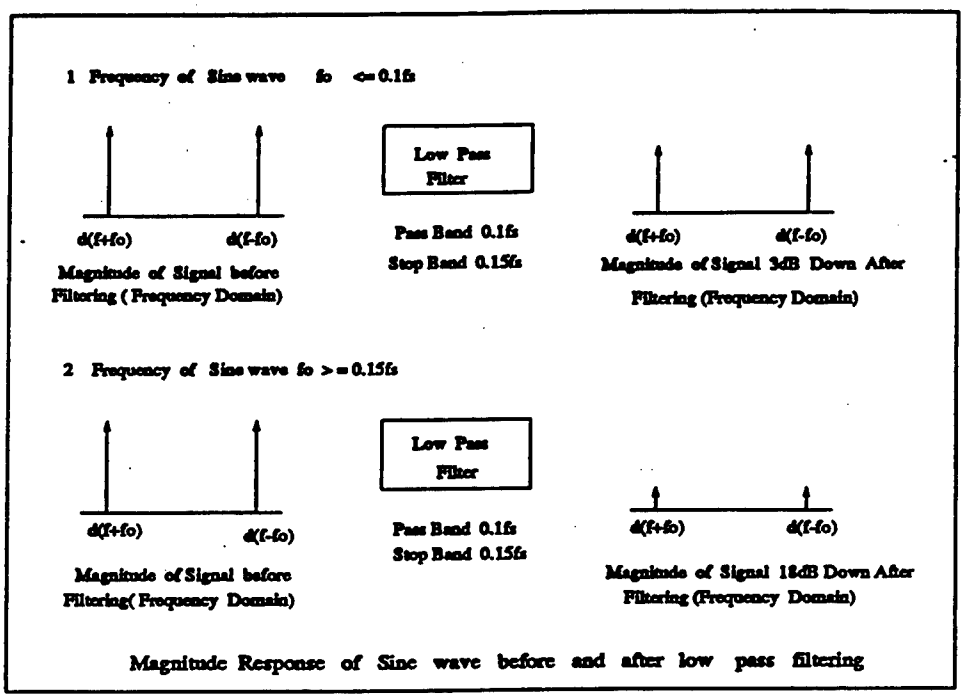


Figure 5.3: Theoretical Results: Low Pass Filtering

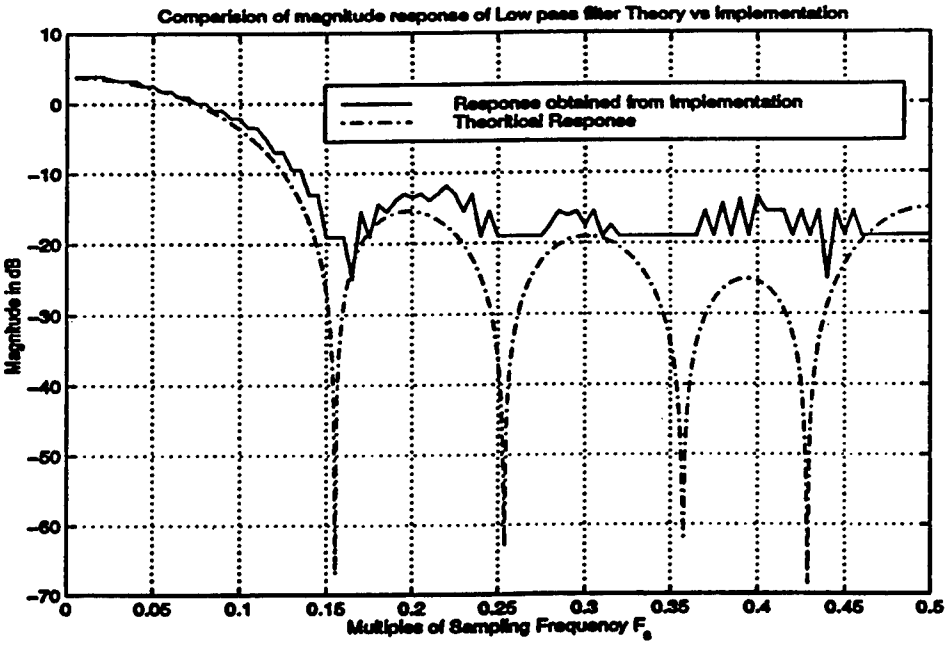


Figure 5.4: Comparison of Magnitude Response of Low Pass Filter

5.1.2 High Pass Filter

Design of the high pass filter is accomplished in the Hardware Design System (HDS) of SPW. The filter has the following frequency specifications: the stop band cutoff frequency is 0.01Fs with an attenuation of 13dB; while the pass band cutoff frequency is 0.05Fs with an attenuation of 3dB. With the help of the Filter Design System (FDS), the values of the coefficients are obtained. The number of taps needed to obtain the required response is 11.

The coefficients that are obtained are used to design an inverted form FIR filter. The architecture of the filter model is the same as in the FIR CAD tool, but the coefficients that are obtained are not rounded to the nearest integer. Rather, the coefficients used to implement the filter are in fractional form. To implement a multiplication of input samples with coefficients, the shift and add method is used. Since the coefficients are represented as fractions, we require the use of a shift block that performs a right shift of the samples. The filter is designed by making use of the shift, adder, subtracter and flip-flop blocks. A simulation model for the filter is developed and the model is tested using a sine wave of varying frequency as the input. The hardware description for the design is obtained from the SPW-VHDL link. The link generates VHDL code for the filter model developed in HDS. The VHDL description of the design is used by the synthesis tool to generate a netlist of the filter which is later used by the Xilnx FPGA software to provide the configuration file. The implementation of the filter is accomplished using the Aptix MP3 system emulator. The filter is implemented on a Xilinx 4013PQ208-4 FPGA. The design consumes 322 logic blocks (CLBs) and can run at 10 MHz clock frequency. Table 5.1 gives the number of coefficients required to design the high pass filter, along with the values of these coefficients. The coefficients obtained are expressed in the form of sum or difference of two-power two terms.

The number of CLBs required to implement each weight is also provided. The fixed point shift blocks are used to express the coefficients. Figure 5.5 shows a comparison between the theoretical magnitude response and the response obtained from implementation of the filter on a Xilinx 4000 series FPGA.

TABLE 5.1 - HIGH PASS FILTER DESIGN PARAMETERS			
Weight number	**Value of the weights**	**Representation of the weights**	**Number of CLBs required**
W0, W10	-0.2185	$-2^{-2} + 2^{-5}$	25
W1, W9	-0.046875	-2^{-4}	45
W2, W8	-0.0625	-2^{-4}	25
W3, W7	-0.0625	-2^{-4}	25
W4, W6	-0.0625	-2^{-4}	25
W5	1	2	35

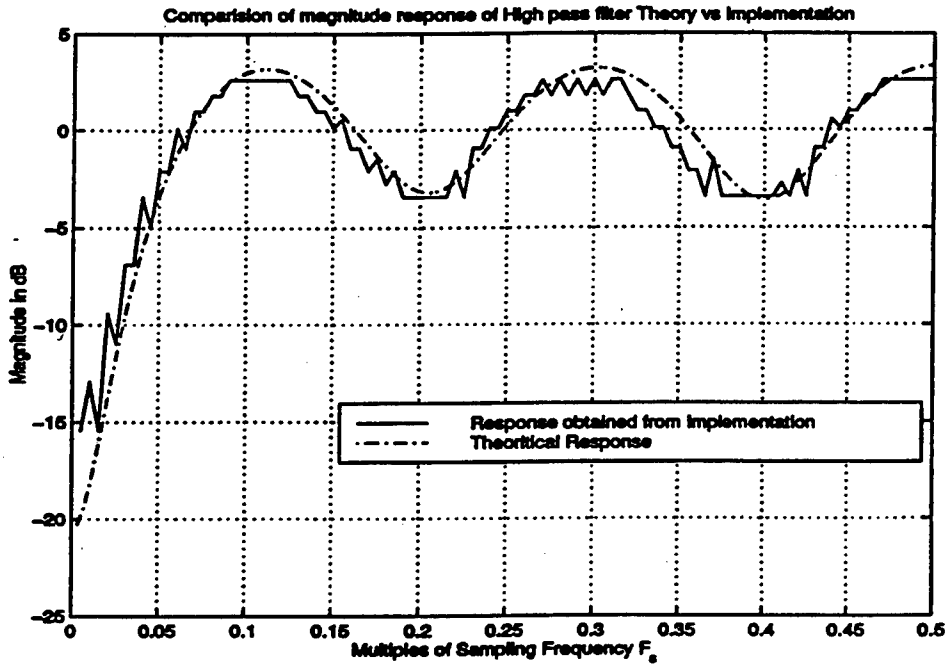


Figure 5.5 - Comparison of Magnitude Response of High Pass Filter

5.1.3 Band Pass Filter

This section describes the design of a band pass filter. The band pass filter is developed using a concatenated high and low pass filter designed in the previous sections. The band

pass filter is designed to have a normalized bandpass from 0.05Fs to 0.1Fs within 3 dB attenuation. The out of band attenuation is 15 dB.

The coefficients of the filter are obtained from the Filter Design System (FDS) of SPW, and the coefficients are expressed as a sum of power-of-two terms. The coefficients obtained and their representations in power-of-two terms are given in Table 5.2. The block diagram of the band pass filter developed by cascading the high pass and low pass filter is shown in Figure 5.6. The I/O FPGA in the design is used for routing the signals to the filters implemented on FPGA2 and FPGA3. The I/O FPGA is also used to convert the offset data generated from the A/D into a two's complement representation since the digital FIR filters are designed for this. The high pass filter is implemented on FPGA2 as shown in the figure and the low pass filter is on FPGA3. Both the filters have 11 taps and operate at a precision of 10 bits.

TABLE 5.2 - COEFFICIENTS FOR LUT MULTIPLICATION

Weight number	Value of the weights	Representation of the weights	Number of CLBs required
W0, W10	-0.03125	2^{-5}	25
W1, W9	0.0703125	$2^{-4}+2^{-7}$	50
W2, W8	0.132815	$2^{-3}+2^{-7}$	50
W3, W7	0.21875	$2^{-2}-2^{-6}$	25
W4, W6	0.234375	$2^{-2}\ 2^{-6}$	30
W5	0.28125	$2^{-2}+2^{-5}$	25

Figure 5.7 compares the theoretical magnitude response of the band-pass to the response observed in the implementation.

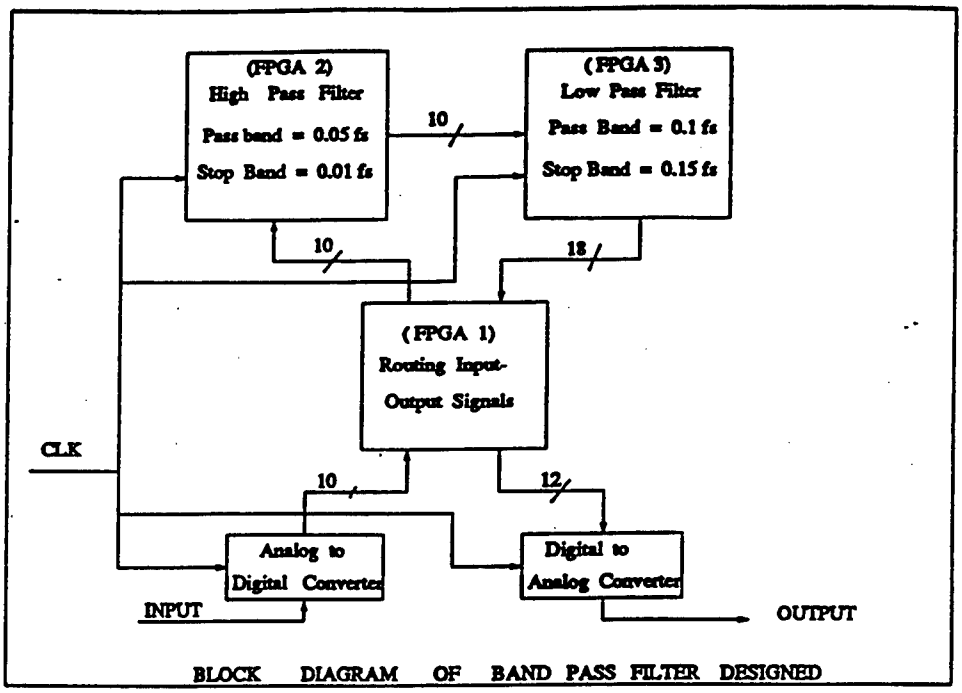


Figure 5.6 - The Block Diagram of Band Pass Filter

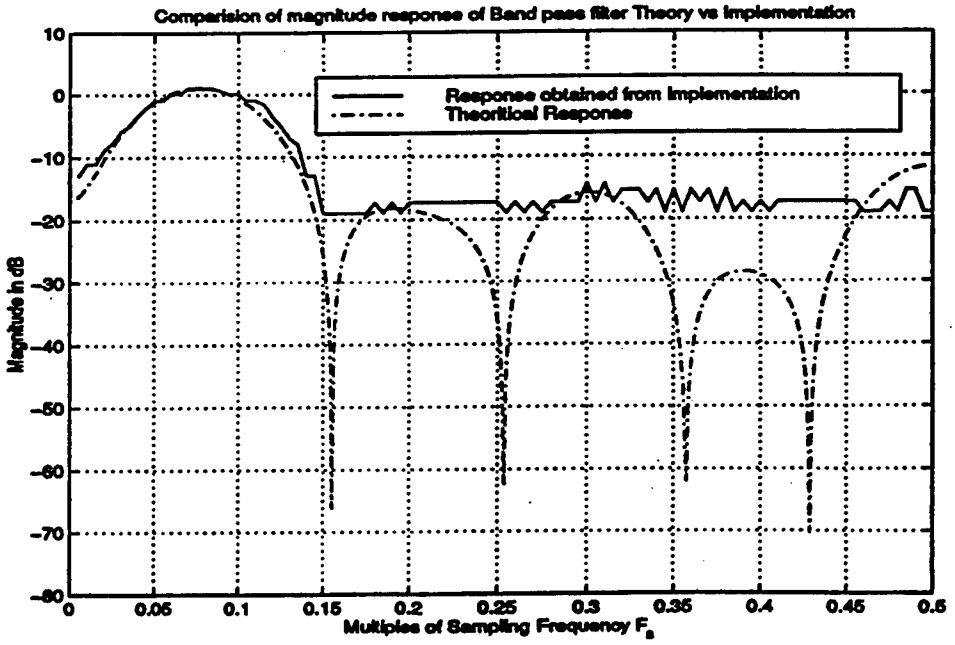


Figure 5.7 - Comparison of Magnitude Response of Band-Pass Filter

5.1.4 Box Car Filter

The Box Car filter is a simple FIR filter in which the coefficients are all unity. The design consists of a direct form FIR structure with twenty five taps. The filter block diagram is shown below. The design consists of twenty four delay elements, 25 taps and the width of each delay element is 10 bits wide. The design also uses 11 bit adders in order to add the delayed samples. The frequency response of the filter has 30dB attenuation in magnitude at multiples of 0.04Fs.

The filter is designed in HDS and simulated in SPW using square wave and sine wave inputs. The filter is implemented on the FPGAs and the results obtained are very similar to that obtained when the system is simulated. The design requires 379 logic blocks (CLBs) on a Xilinx 4013PQ208-4 FPGA and runs at a speed of nearly 20MHz. Figure 5.9 gives a comparison of the theoretical frequency response with the measure response.

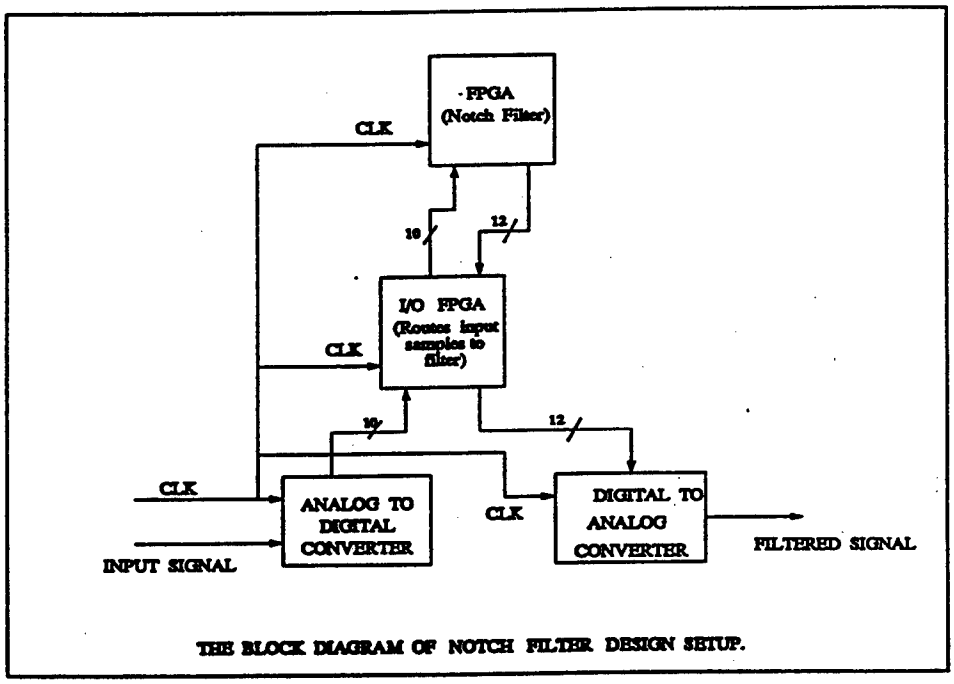


Figure 5.8 - The Block Diagram of Twenty five Tap Box Car Filter

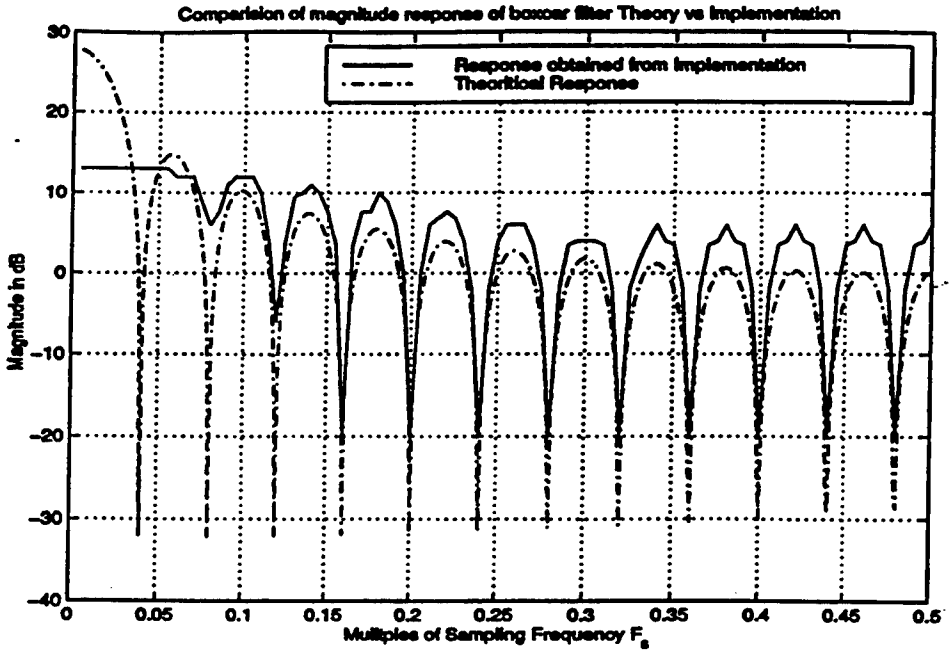


Figure 5.9 - Comparison of Magnitude response of Box Car Filter

5.2 Twelve Tap FIR Filter using LUT Techniques

This section describes the design of a Low Pass 12-tap FIR filter using constant coefficient multipliers using Look Up Table techniques and implemented on a Xilinx 4013 FPGA using the Aptix Mp3 prototyping board..

5.2.1 Design using SPW

The SPW tool provides the means for designing the system schematically and for verifying and simulating the design. SPW is a powerful block oriented software tool suitable for designing any kind of DSP system. The Filter Design System (FDS), which is a part of SPW, was used to obtain the filter's coefficients. First, the filter's frequency characteristics (Low Pass, cutoff frequency etc) were given as input to FDS, which in turn calculates the coefficients and the number of taps needed to meet the desired specifications. The Block

Design Editor (BDE), which is another subsystem tool of SPW, was used to design the filter schematically using standard DSP blocks like adders, multipliers, delay elements etc.

All these blocks are located in the Hardware Design System (HDS) library of SPW, which allows the use of fixed-point arithmetic in the design. The advantage of using fixed-point arithmetic is that we can accurately model the real behavior of the digital system because we don't need to deal with loss of precision when using floating point arithmetic in a bit-limited digital system. After the system is designed schematically, the Signal Calculator System of SPW is invoked to simulate the operation of the design. The Signal Calculator is capable also of generating fixed-point signals which can be applied to the design and verify its real performance.

The FIR filter described in this report has the following characteristics:

Type: Low Pass FIR
Tap length: 12
Cutoff frequency: fc=0.1Fs (Fs = sampling frequency)
Stopband edge: 0.13Fs
Stopband Attenuation : 30 dB
Filter Method: Equiripple/Low Pass
Input Data width: 8 bits
Output Data width: 12 bits
Coefficients: 8 bits

Using the Filter Design System (FDS), which is a part of the SPW design tool, the coefficients of the filter were obtained (in Double precision format):

$b_0 = b_{11} = 0.040473$
$b_1 = b_{10} = 0.075372$
$b_2 = b_9 = 0.11826$
$b_3 = b_8 = 0.16903$
$b_4 = b_7 = 0.20653$
$b_5 = b_6 = 0.22994$

Figure 5.10 below illustrates the block diagram of the design. As shown, the design is implemented as a parallel inverse structure and only 6 Look-Up-Table (LUT) blocks are used because the 12 coefficients are symmetric.

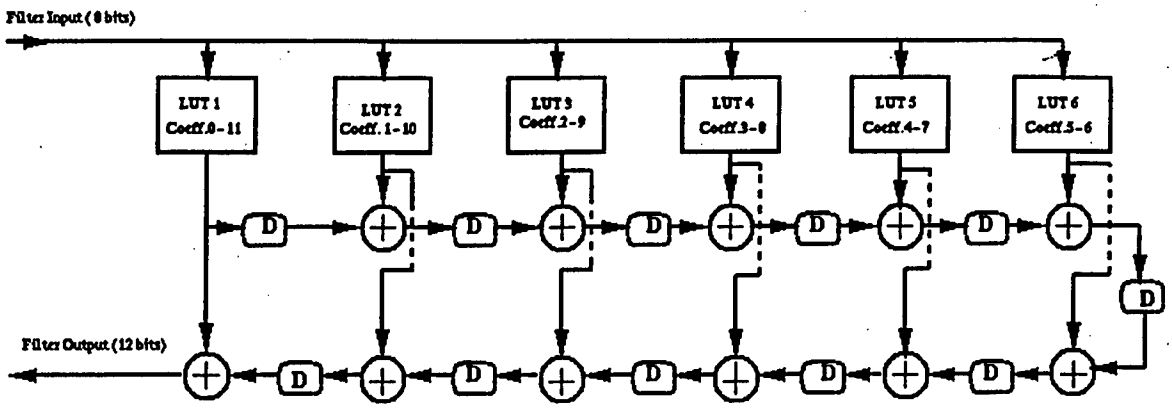


Figure 5.10 - Block diagram of the FIR Filter

As discussed before, the LUT blocks are used instead of regular multipliers. The internal structure of each LUT block is shown in Figure 5.11 below.

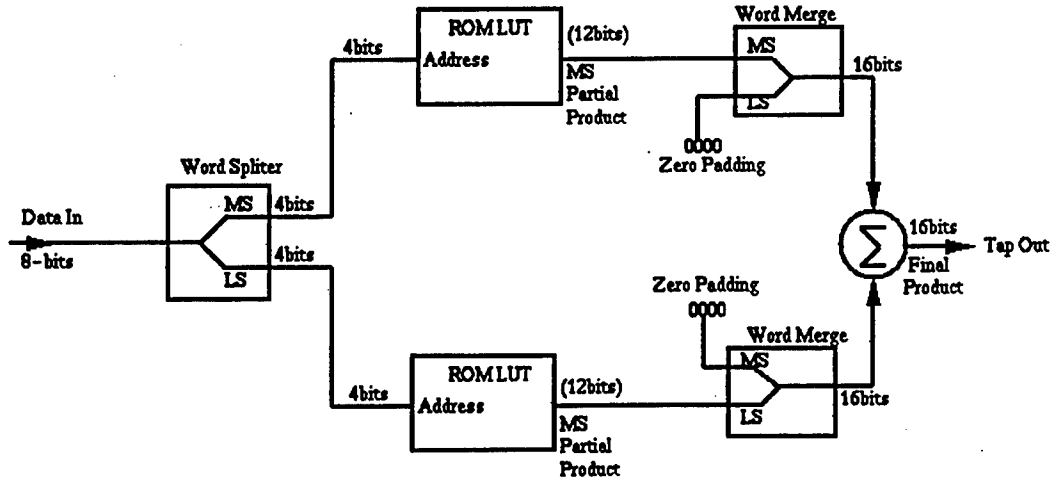


Figure 5.11 - Internal structure of LUT

As shown in Figure 5.11 above, the incoming 8-bit data is split into two segments of 4-bits each. Each 4-bit segment is used to address a ROM Look-Up-Table. So, each LUT block shown in Figure 5.10 it actually contains two ROM LUTs. We could have had only one

ROM LUT and apply all the eight bits of the data on it, but this would be space consuming on the FPGA because it would need a ROM with $2^8 = 256$ memory locations. By splitting the data in two, each ROM has now $2^4 = 16$ memory locations.

The upper (Most Significant) LUT contains all the partial products of the 8-bit coefficient times the most significant 4-bits of the data (i.e., 16 partial products). Similarly, the lower (least significant) LUT contains all the possible partial products of the 8-bit coefficient times the least significant 4-bits of the data. Therefore, at the output of each LUT we have a 12 bit partial product (4bits data + 8bits coefficient). Each 12 bit partial product is appropriately zero padded and then Summed to produce the final product at the output of the Tap.

5.2.2 Filter Performance

The design was implemented on a Xilinx 4013 IO FPGA on the Aptix Mp3 prototyping board. Using the Synopsys and Xilinx tools, the FPGA area used and the max time delay of the filter were obtained as shown on Table 5.3 below.

TABLE 5.3 - AREA AND TIMING PERFORMANCE	
Total number of CLBs used	302
% space of 4013 FPGA used	52%
Max data arrival time (=max delay)	51.65 sec
Max filter clock speed	20 MHz

Table 5.4 below shows a comparison of the filter described in this report with two other filters designed before.

As shown in the tables, the LUT based FIR filter has much better speed performance compared with the two other filters which use regular multipliers in the design. Even the 8-tap filter, which uses much less space than the 12-tap LUT based filter, has bigger time delay in the Data path compared with the 12-tap LUT based design.

TABLE 5.4 - COMPARISON OF AREA AND TIMING PERFORMANCE OF 3 FILTER DESIGNS			
Filter Type	**Total # of CLB's**	**Max Data Delay**	**Max filter clock**
FIR 8-tap with regular multipliers	140 (24%)	87.8 nsec	11.4 MHz
FIR 12-tap with regular multipliers	329(57%)	130.36 nsec	7.5 MHz
FIR12-tap with LUT constant coefficient multipliers	302(52%)	51.65 nsec	20 MHz

Figure 5.12 below shows the theoretical frequency response of the filter obtained using Double Precision arithmetic for the coefficients. On the other hand, Figure 5.13 illustrates the experimental frequency response with fixed point coefficients. From Figure 5.12 we observe that there is a steep roll-off at the cut off frequency (0.1Fs). At this cut off point we can see that the magnitude of the response falls by -3dB from the maximum. This graph was obtained using Double Precision arithmetic for the coefficients which means that it gives us the desired frequency response with the characteristics given from the Filter Design System tool. In Figure 5.13 however, we observe that the real frequency response of the filter differs from the desired theoretical response of Figure 5.12. The -3dB point on the experimental response occurs at a normalized frequency of $0.08F_s$.

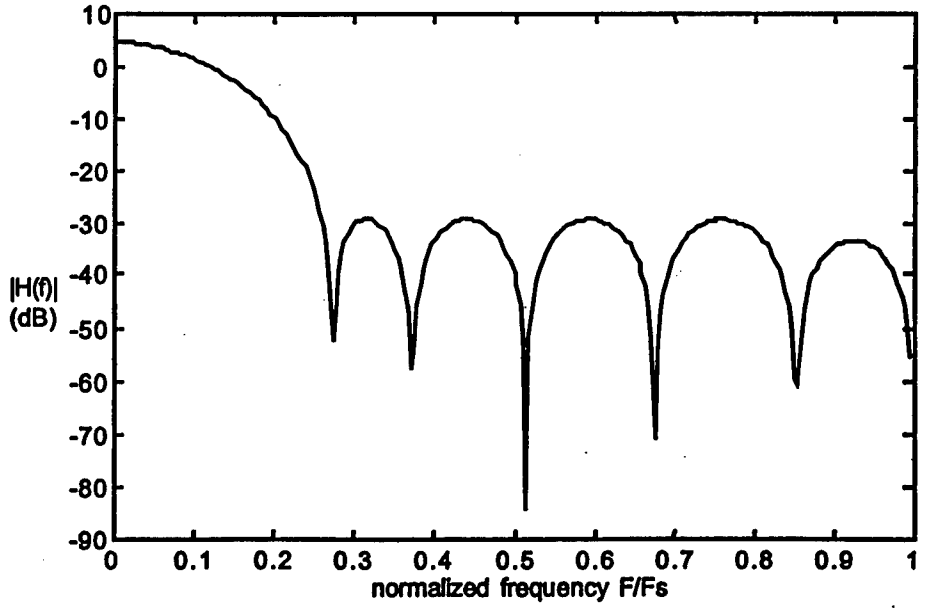


Figure 5.12 - Theoretical Frequency Response

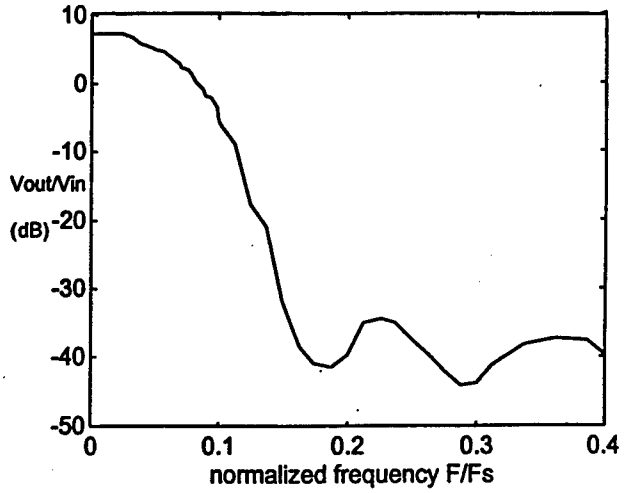


Figure 5.13 - Experimental Frequency Response (Fixed-Point Coefficients)

As shown on Figure 5.13, the -3dB point occurs at around 0.08 Fs and not at 0.1Fs which is the design specification. This happens mainly because of the quantization error introduced on the coefficients. According to our design, the coefficients of the filter are represented by an 8-bit fixed point binary number. Therefore, when the Double Precision coefficient number is represented by the 8-bit fixed point number, there is loss of precision because of quantization. This error can change the filter's characteristics, and especially the frequency response and cut off point, because the original value of the coefficients changes after quantization. For example, coefficient b0 has a value of b0 = 0.040473. When this number is represented with 8-bit fixed point (two's compliment) arithmetic, it becomes 0.0390625 which is the closest approximation to the original number.

Other than that, the frequency response of the filter shows good rejection characteristics and meets the -30dB rejection in the stopband as specified in the design. Compared with filter designs which use regular multipliers, this filter exhibits much better speed performance and area occupation on the FPGA. A maximum delay of 51.65 nsec was obtained which allows the filter to operate on clock speed of 20 MHz. The frequency response of the filter shows good rejection characteristics (-30 dB in the stop band), but due to quantization error introduced on the coefficients, the cut off frequency is shifted from 0.1Fs to 0.08Fs.

This design occupies 52% of a Xilinx 4013 FPGA, which means that there is still enough space on the FPGA to expand the design with more taps. The benefit of this will be better frequency response characteristics with a trade off on delay increase since the data will have to travel in longer paths. Investigation of this expansion is planned for the future. Also, the future work includes investigation of some other design techniques for delay reduction (pipelining for example). Other types of filters are also under investigation (High Pass, Band Pass etc).

5.3 PN Sequence Generator

Binary PN sequence generators are used in Direct Sequence (DS) spread spectrum systems as spreading codes. The sequence length before repetition can be extremely long and is assumed to be random - i.e., the autocorrelation function is an impulse, or nearly so. A PN sequence generator block diagram is shown in Figure 5.14

The circuit consists primarily of shift registers which are implemented using D type flip-flops. The input to the first flip-flop in the shift register is the output of the parity generator, which is implemented using exclusive-or gates. The inputs to the parity generator are the outputs of the flip-flops. The character of the PN sequence generated depends on the number of flip-flops employed and on the selection of which flip-flop outputs are connected to the parity generator. The PN sequence generator is designed using blocks provided by the Hardware Design System (HDS) and simulated by the simulator provided by SPW. For synthesis of the PN sequence generator we require the VHSIC Hardware Description Language (VHDL) code. The detail block diagram of the PN sequence generator developed in HDS is given in Figure 5.15.

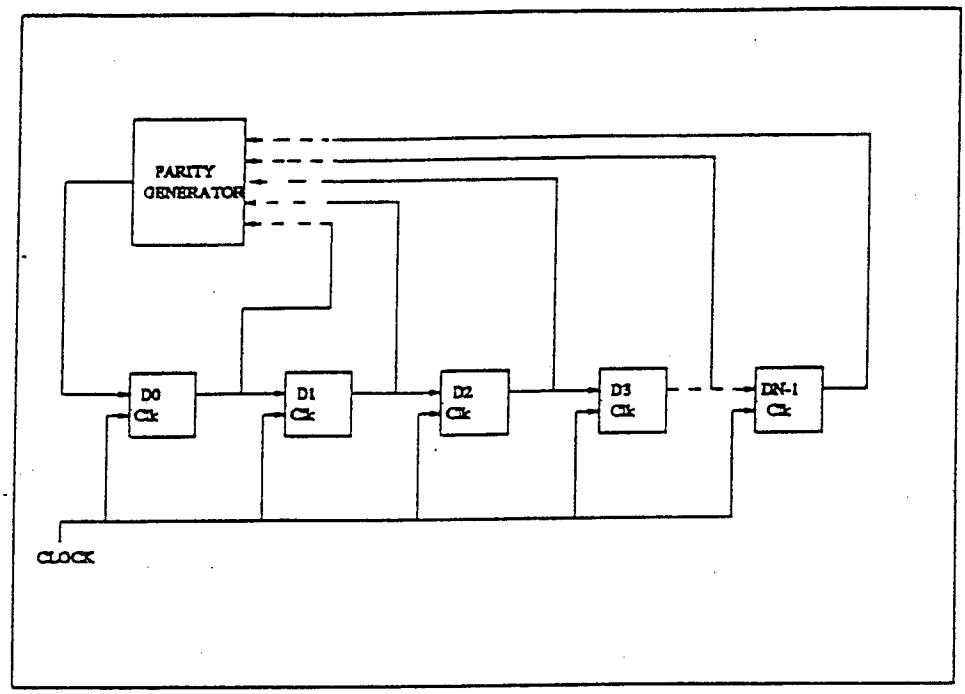


Figure 5.14 - Block Diagram of PN Sequence Generator

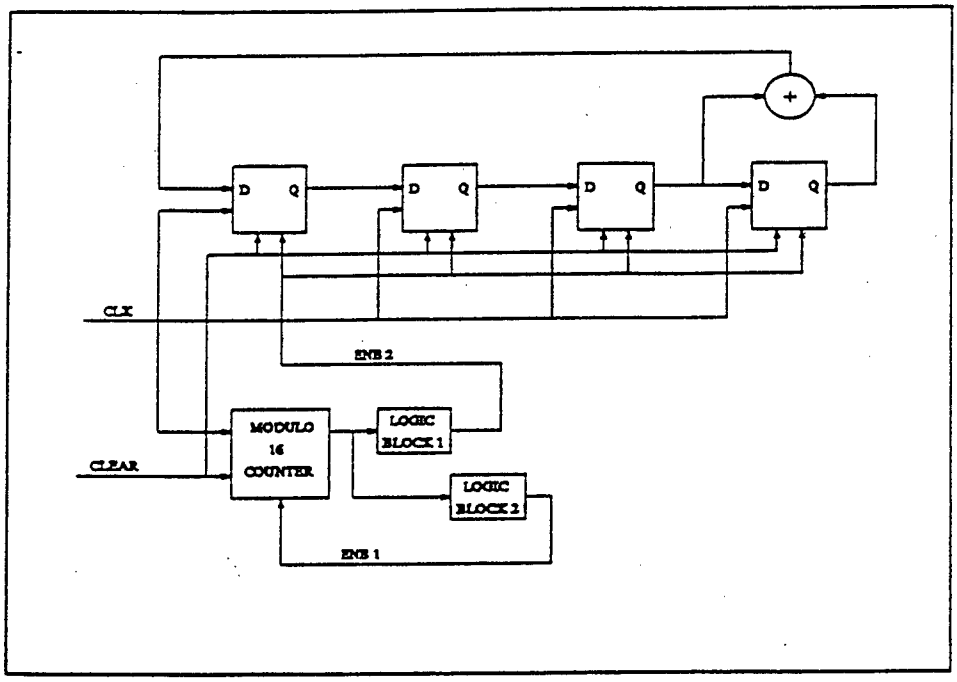


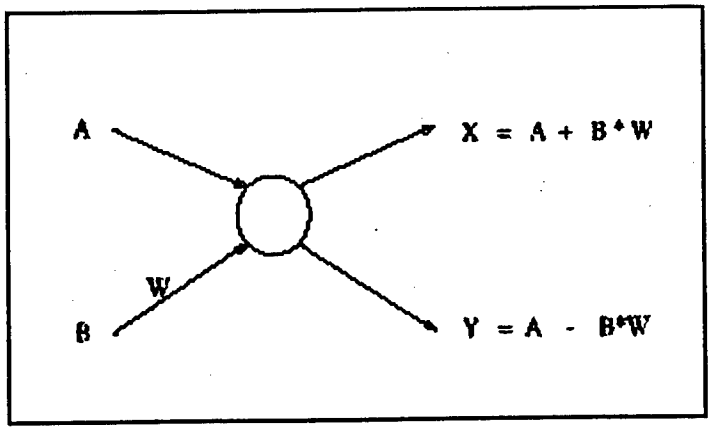
Figure 5.15 - Detail Block Diagram of PN Sequence Generator

The VHDL code generated is used by the synthesis tool to generate the design. In our case we used Synopsys as the tool to synthesize the PN sequence generator. The Synopsys synthesis tool takes the VHDL file as input and complies the code. The FPGA compiler gives a netlist of the design which is used by the Xilinx software to obtain the downloadable bit file. This is the general procedure for synthesis of any design developed in HDS and SPW. The advantage is that changes can be made in the top level design and synthesized quickly using the synthesis tool. The same procedure as described for the low pass filter implementation is used to implement the PN sequence generator on the Aptix MP3 board.

5.4 Design of 8 point FFT

5.4.1 Description of the Algorithm

In the radix-2 decimation in frequency (DIF) FFT algorithm, the frequency decimation passes through total of M stages, where $N = 2^M$ with N/2 2-point DFTs or butterflies per stage, giving a total of $N/2 * \log_2 N$ butterflies per N-point FFT.



The Radix 2 Butterfly Element

Figure 5.16 - The Block Diagram of Radix 2 Butterfly Element

In the case of an 8-point DFT implemented using the radix-2 DIF FFT algorithm, the input samples are processed through three stages. Four butterflies are required per stage, giving a total of twelve butterflies in the radix-2 implementation. Each butterfly is a 2-point DFT of the form depicted in the Figure 5.16 . The inputs A and B are the inputs to the radix-2 DIF FFT butterfly. Multiplication with the twiddle factor W is shown in the Figure 5.16 . The outputs of the radix 2 element are X and Y expressed in terms of inputs A, B and the twiddle factor W.

5.4.2 Details of Implementation

The 8 point FFT is designed using HDS. The design uses the shift and add technique for multiplication of the filter coefficients with the signal samples. The coefficients that are used in the design are rounded so they can be expressed as the sum of two, power-of-two terms. The design makes use of a radix-2 algorithm which includes a standard butterfly element and necessary weights. The block diagram of the eight point FFT is shown in Figure 5.17.

The block diagram gives the overview of the 8 point FFT system. The design makes use of two FPGAs to implement the following two functions: FPGA1 is used for buffering the incoming signal and FPGA2 is the 8 point FFT processor which computes the FFT of the samples provided by the input buffer stage. The detailed implementation of the 8 point processor is shown in Figure 5.18. The figure makes use of the radix 2 DIF algorithm to compute the 8 point FFT.

The element node in the Figure 5.18 is the radix 2 butterfly element except that the outputs of the node element are weighed by the twiddle factors. The multiplication with twiddle factor is implemented using the add and shift method by expressing the twiddle factor in terms of two power-two terms. The block diagram of the node element implemented is shown in Figure 5.19: The requirement is that the 8 point FFT be capable of processing real time data, hence it is necessary to buffer the incoming data bits. To buffer the input data it is necessary to latch the incoming serial data which requires us to demultiplex the data and latch, so that the data is available for processing.

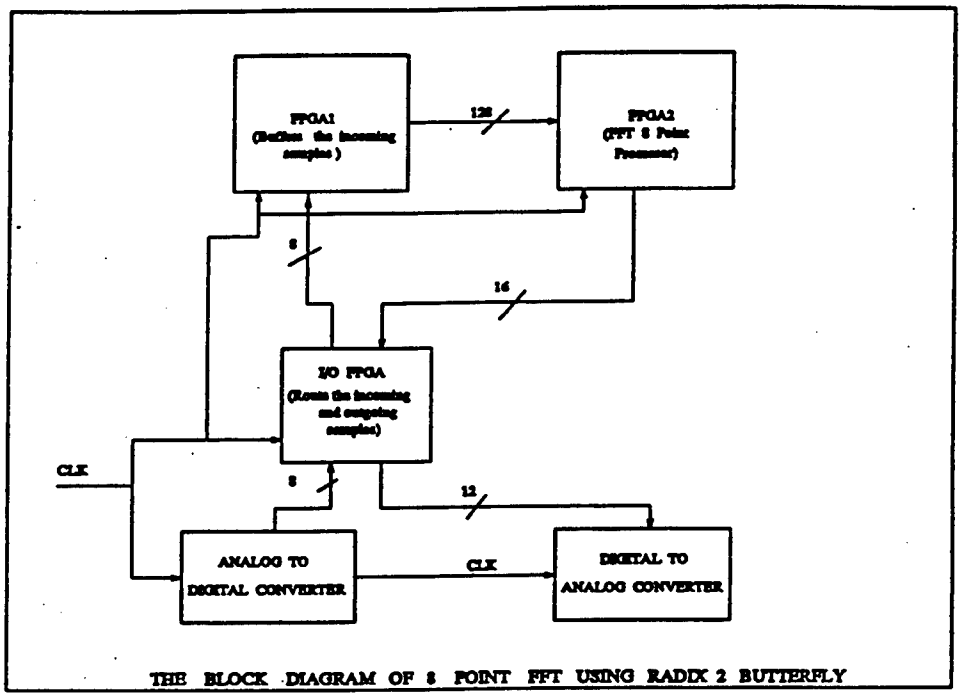


Figure 5.17 - The Block Diagram of Eight point FFT using Radix 2 Algorithm

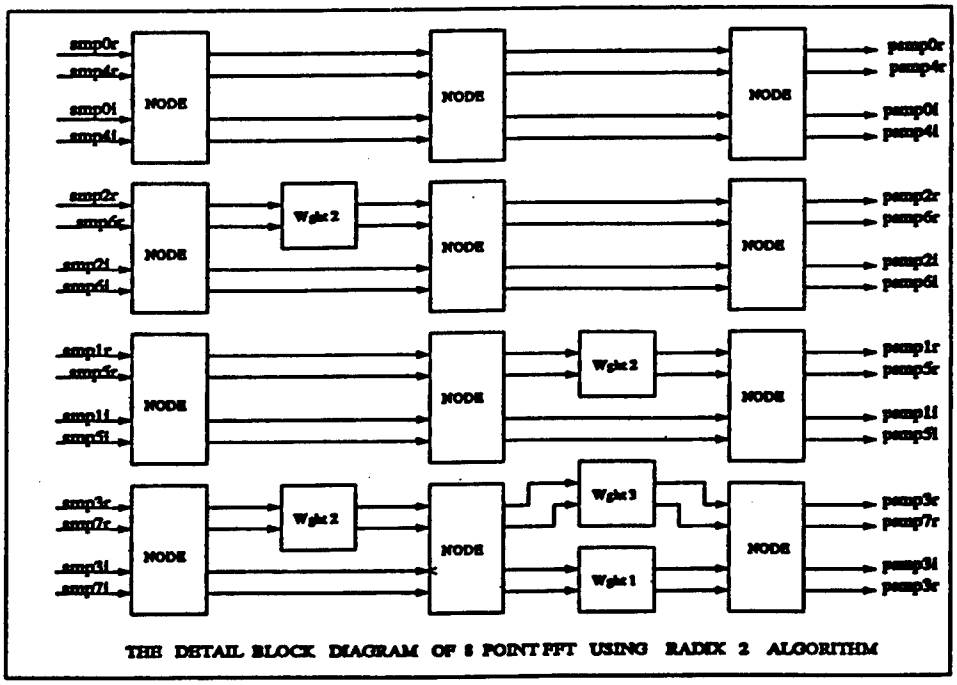


Figure 5.18 - The Detail Block Diagram of 8-point FFT using Radix 2 Algorithm

The block diagram of the buffering stage, which buffers the incoming real time A/D converted signal is shown in Figure 5.20. To implement the buffering stage we need around 250 CLBs on the FPGA. The design of an 8 point FFT requires more than 350 CLBs; therefore, both the buffering stage and 8 point FFT cannot be implemented on a single FPGA, hence we require two FPGAs to implement both of them.

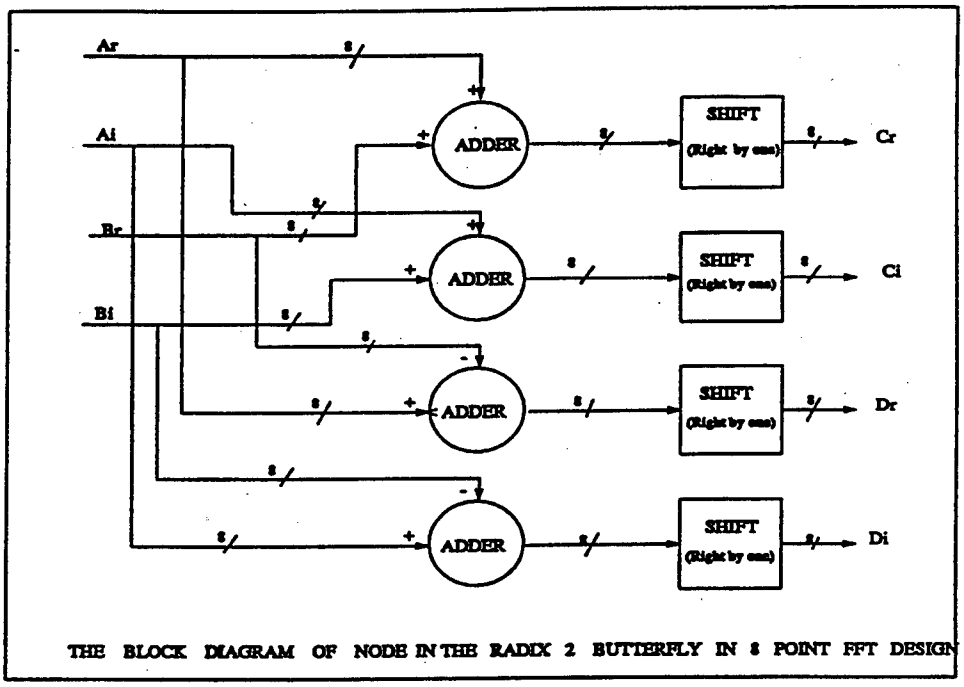


Figure 5.19 - The Block Diagram of Node element used in 8 point FFT

To implement the buffer stage, VHDL code is compiled and simulated in order to verify the design. The VHDL code is used as an input to the Synopsys tool to synthesize the logic.

The 8 point FFT designed in HDS follows a procedure similar to the PN sequence generator. The model of the 8 point FFT developed in HDS and the VHDL code generated by the VHDL link of SPW (HDS) is used for synthesis. The synthesis is accomplished using the Synopsys FPGA compiler which provides a netlist file which is used by the Xilinx Synopsys libraries to provide the downloadable bit file which can be used to implement the FFT on the Xilinx FPGA.

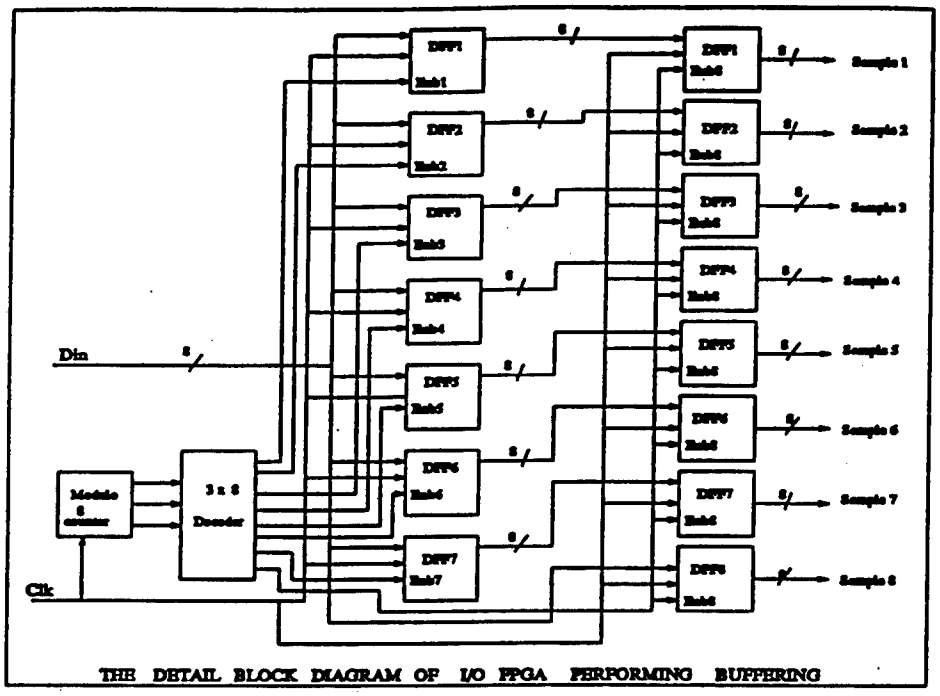


Figure 5.20 - The Block Diagram of Buffering Stage used in 8 point FFT

5.5 Scalable FFTs

As the number of points of FFT implementation increases, it requires buffering of incoming samples and processing of the stored samples. The buffering suggested in the previous section is not practical and requires a memory to store the incoming samples and the intermediate samples that are to be processed .

The implementation can be done in two ways, one method is to make use of the memory provided by the FPGA and second way is to use an external memory. For very large FFTs it is advisable to use external memory. Both methods of implementation make use of the same algorithm except that, the number of points, amount of time and hardware required differ.

5.5.1 Description of the Algorithm

The algorithm used to compute the 16 point FFT is based on the radix 4 FFT. The hardware requirements for the implementation are memory to store the incoming and processed samples, a radix 4 butterfly for processing purposes and a multiplier for performing multiplication of twiddle factors with the samples. The multiplication operation is obtained by shift and add operations on the samples.

We make use of just one complex multiplier for multiplying the coefficients and the samples. A critical component of the design is the implementation of the control logic for synchronizing the memory, radix-4 butterfly element and the multiplier. The radix-4 algorithm is used when the number of the points of FFT required is a power of 4. In this way, the original one dimensional array can be broken down into elementary computations of four point DFTs. The systems developed are 16 point and 64 point FFTs. The block diagram of a 16 point FFT is shown in Figure 5.21 and can be expressed as a flow diagram in a similar manner as the radix-2 flow diagram.

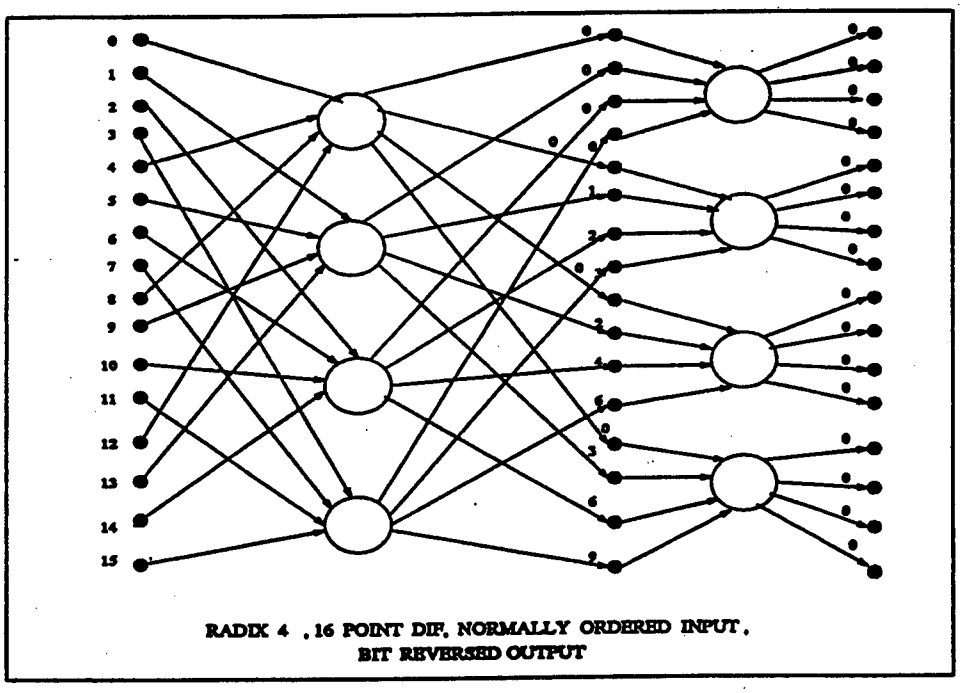


Figure 5.21 - Flow Diagram of 16 Point FFT

5.5.2 General Overview of the System

This section gives the details of how the algorithm is implemented. The block diagram of the design is shown in Figure 5.22. The detailed block diagram shows how the different hardware components are connected. The input samples from the A/D board are interfaced to the MP3 system emulator and are routed with the help of the I/O FPGA to the radix four butterfly element. The control logic is implemented in the I/O FPGA which generates the control signals for the memory, the butterfly element and the complex multiplier.

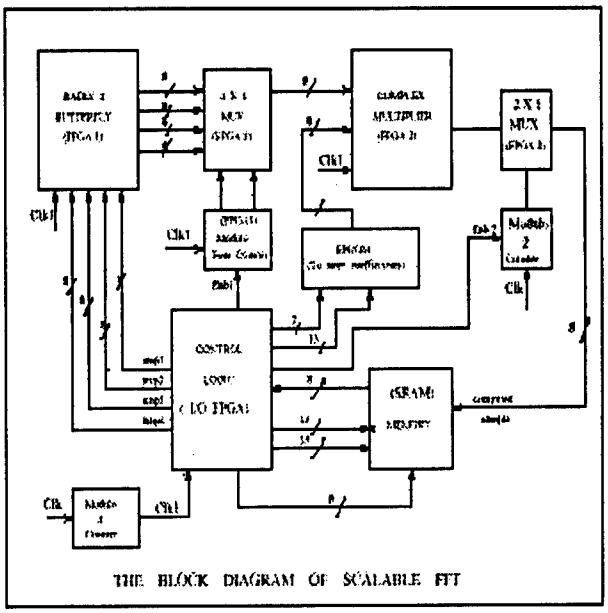


Figure 5.22 - The Block Diagram of the System

The MP3 board, as discussed earlier, is a field programmable circuit board which is used to prototype complex digital systems. The programmable interconnect components on the board provide the programming capability of the board. ASICs, FPGAs and other components can be plugged into the board. The board has a diagnostic plug interface to which diagnostic devices are connected. The hardware components required to implement the FFT algorithm are SRAM, ROM and FPGAs.

The radix-4 butterfly element is a four point DFT implemented on the FPGA. The complex multiplier, which is also implemented on a FPGA, operates on complex inputs and

coefficients and gives a complex output signal. The control signals required to synchronize all of the interconnections between the hardware components are implemented on an I/O FPGA. The advantage of putting the butterfly, complex multiplier and control unit on a FPGA is that the design can be changed. The details of implementation of the control logic, radix 4 butterfly and complex multiplier are discussed in the following sections.

5.5.3 Control Logic Design

Control logic is used to synchronize various components of the design. The control logic generates control signals for the whole design. The main control signals generated by the unit are those used to enable read and write address counters which provide addresses to perform memory read and write operations.

The control unit enables all the read and write counters at appropriate times. Three read and two write counters are required for the entire computation of 16 point FFT. The counters are multiplexed to form a single bus, which is used as the address lines of a second port for the memory. The Figure 5.23 shows various blocks of the control logic unit.

The data buffer unit serves the purpose of latching the incoming data and sending it to the data bus of port one of the memory. The memory control unit generates the read and write addresses for both ports of the memory along with the port enable and read/write enable signals. The memory control unit generates control signals for other parts of the design such as the ROM control unit, Coeff buffer, input data buffer and the buffer used to latch data read from memory. The ROM control unit generates the addresses for the ROM along with the output enable. The ROM unit is used for storing coefficients and the coefficients are accessed by the address generated by the ROM control unit. The Coeff buffer is used to store the real and imaginary parts of the coefficients and route them to the complex multiplier.

The memory control signals buffer is used to buffer all the memory control signals along with the addresses. The buffering is done in order to avoid interfacing problems between two chips. The two chips are the FPGA, which generates the control signals and addresses, and the SRAM. The data read from memory is 8 bits wide and inputs to the

radix 4 element are four samples, each 16 bits wide. Therefore, the data read from the memory needs to be buffered in order to input it to the radix-4 butterfly element. In the following section the memory control unit is discussed in detail.

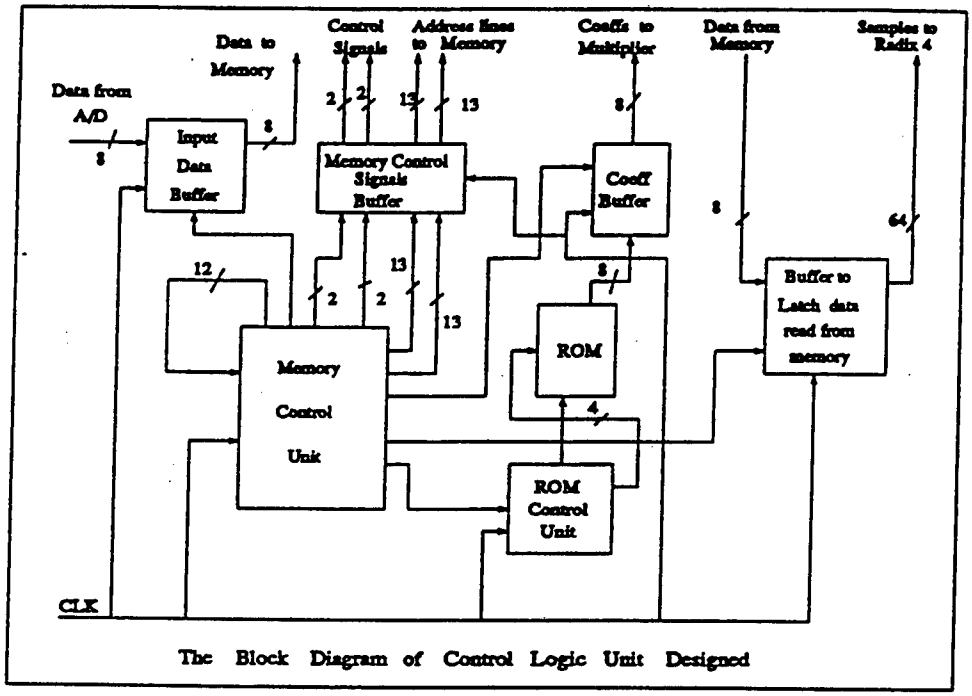


Figure 5.23 - Block diagram of Control Logic Unit

5.5.4 Memory Control Unit

The memory control unit is the main part of the control logic . The purpose of the unit is to generate control signals and addresses for the design. The unit includes the read and write counters to generate the addresses to access the memory. Figure 5.24 gives details of the memory control unit.

In Figure 5.24 the read counters and the write counters generate read/write addresses to access the second port of the memory. The outputs of these counters are multiplexed using a 5 to 1 multiplexer. The logic block takes the outputs of the read/write counters to generate memory control signals and enable signals which are used to enable the counters. The write address block in the figure generates write addresses for port one of the memory chip. A detailed explanation of each block is given in the sections to follow.

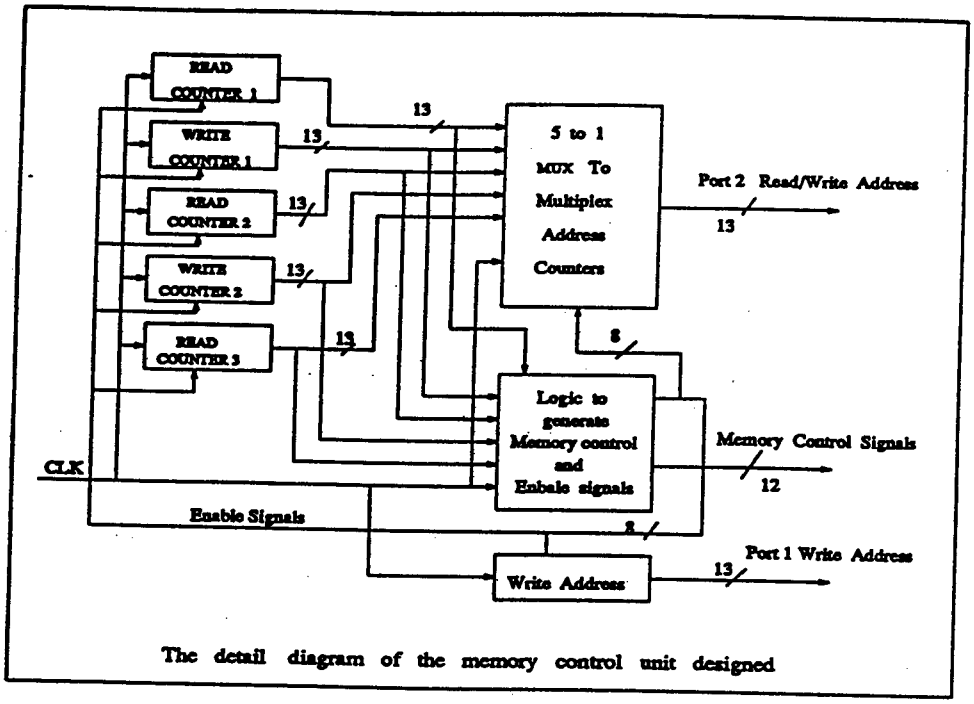


Figure 5.24 - Diagram of memory control logic unit

5.5.4.1 Write Address Counters

This section describes how the write address is generated for storing the incoming samples. Since the design is required to store 16 samples, the write address counter is a modulo 16 counter. The clock required to run this counter is thirty two times slower than the clock required for processing the stored data. To eliminate the problem of clock skew, which occurs when more than two different clocks are used, we derive the slower clock from the faster clock by making use of a divide by five counter. Since a dual port SRAM is to store the incoming samples, one of the ports of the memory is used for writing the data samples obtained from the A/D. Writing data into the memory requires a port enable signal, address bus and write signal. The write address, as mentioned earlier, is a simple modulo 16 counter and the data is written to memory on the rising edge. Hence the write signal for port 1 is obtained from inverting the clock used to run its address counter.

The speed of the clock used for generating the write address is determined by calculating the number of clock cycles required for computing the 16 point FFT. The write address is generated continuously to store samples coming at a constant rate. Hence one

port of the memory is permanently used for writing input data. Since the 16 point FFT of a signal requires writing the computed data twice into memory we require two more sets of write addresses. These address lines are used to access the second port of the memory. The write is executed on the rising edge of the signal, which is obtained by inverting the clock used for generating the write addresses. This clock signal is obtained using a modulo two counter clocked by the system clock. Table 5.5 gives the number of write counters used for designing a 16 point FFT, along with the port accessed, the clock required to generate address counter, the range of the addresses accessed by the counters and the time for which they are enabled. In Table 5.5 Cntr1 is a counter that generates the address for writing the data samples obtained from the A/D into the memory. These addresses access only port one of the memory and write into the memory from address 0 to F in the sequential order. The Cntr1 is always enabled as it has to write the data coming from the A/D.

TABLE 5.5 - DESIGN OF THE 16-POINT FFT

Write counter number	Port number accessed	Clock speed for generating addresses	Addresses accessed	Enable time for write
Cntr 1	1	sys-clk/32	0 - F (hex0	always
Cntr 2	2	sys-clk/2	10 - 2F (hex)	certain time
Cntr 3	3	sys-clk/2	10 - 2F (hex)	certain time

The write counter Cntr2 is used to write back the computed data during the first stage of processing of the data samples. The range of addresses accessed is from 10 - 2F (hex). The final write counter Cntr3 generates an address to write back the data that is computed after stage two. Cntr3 also accesses addresses from 10 - 2F (hex), and both Cntr2 and Cntr3 generate addresses that access the second port of the memory. The difference in the two counters are the times for which they are enabled and the order in which they count.

5.5.4.2 Read Address Counters

The addresses for the memory read are provided by the read counters. There are three read counters to generate a different set of addresses at different times. These read counters are enabled at different times to count and generate the required address lines. According to the algorithm, one read operation from the memory is followed by the processing of the sample from memory and then writing back of the processed sample. Since we are making use of the radix-4 butterfly element, we read four memory locations consecutively. The purpose of the read counters is to generate addressees for these memory locations in the correct order. The number of read counters required for a 16 point FFT computation are three. The first read counter increments such that the sequence of the count generates addresses to read from 0th, 4th, 8th and 12th memory locations.

TABLE 5.6 - DESIGN OF THE 16-POINT FFT (CONTD.)

Write counter number	Port number accessed	Clock speed for generating addresses	Addresses accessed	Enable time for write
Cntr 1	2	sys-clk/4	0 - F (hex0	4
Cntr 2	2	sys-clk/4	10 - 2F (hex)	8
Cntr 3	2	sys-clk/2	10 - 2F (hex)	32

After the samples are read from memory they are buffered and input to the radix-4 butterfly element. The read counters are enabled at appropriate times as mentioned earlier. The first read counter is enabled only after the 12th input sample is written into the memory - in fact the counter is enabled one cycle after the 12th data sample is written into the memory. Four read operations are followed by a DFT computation of four samples read from memory, which is followed by the multiplication of the computed samples with the appropriate coefficients. Table 5.6 provides the number of read counters required, the port number accessed by the addresses generated by the counter, the speed of the clock required to drive the three synchronous counters in terms of the system clock, the range of addresses accessed and the number of read operations performed during a read cycle. The Cntr1 counter generates the addresses to read from port two of the memory during the first stage

of processing. Counter Cntr2 is used to read the intermediate computed data written after the first stage of processing; and the final counter Cntr3 is used to read the processed samples that are stored after the second stage of processing. Since it is a 16 point FFT there are only two stages of processing required and hence the final counter reads the computed FFT of the signal.

5.5.5 Design of Radix 4 Butterfly Element

The flow diagram of the radix 4 butterfly element is shown in Figure 5.25. The element inputs four samples and outputs four samples. The algorithm is as follows: the four input samples shown in the figure are multiplied by certain weights and summed to give a single output sample. We require four adders to sum the input samples.

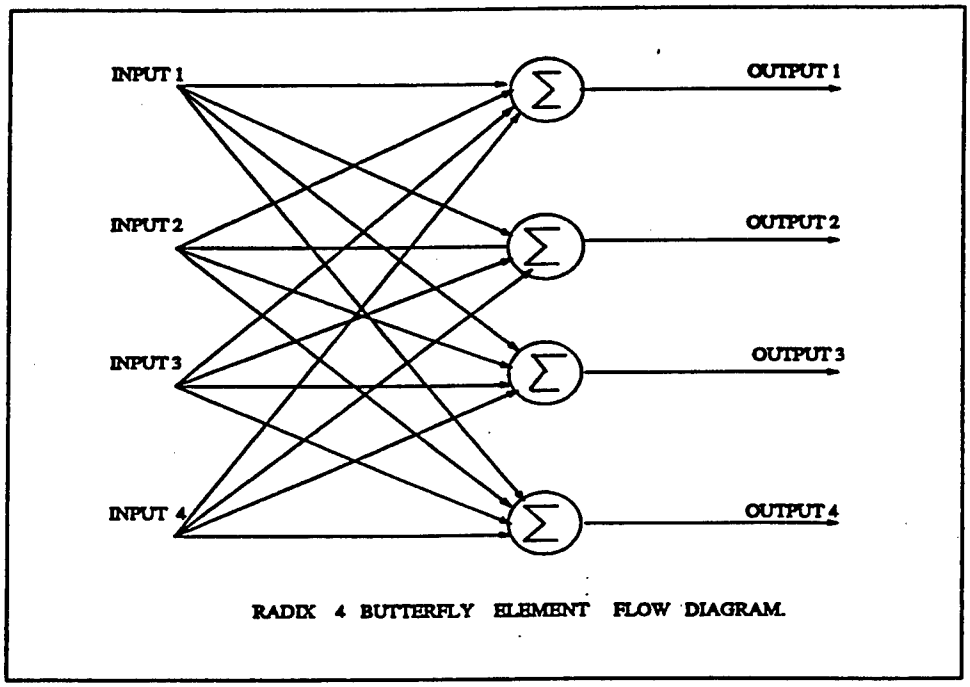


Figure 5.25 - The Flow Diagram of Radix 4 Butterfly Element

The design is pipelined by using flip flops in the path between the inputs and outputs. This helps in increasing the operating clock frequency. The design of the Radix 4 Butterfly element is designed using HDS. The SPW tool generates the VHDL code for the butterfly

element. The code generated is synthesized by the Synopsys FPGA compiler to provide the netlist of the design for configuring the FPGA.

5.5.6 Design of Complex Multiplier

The block diagram of the complex multiplier is shown in Figure 5.26. The multiplier takes the real and imaginary part of the sample as input, along with complex coefficient, which is stored in the ROM. The complex multiplier is designed in HDS section of SPW.

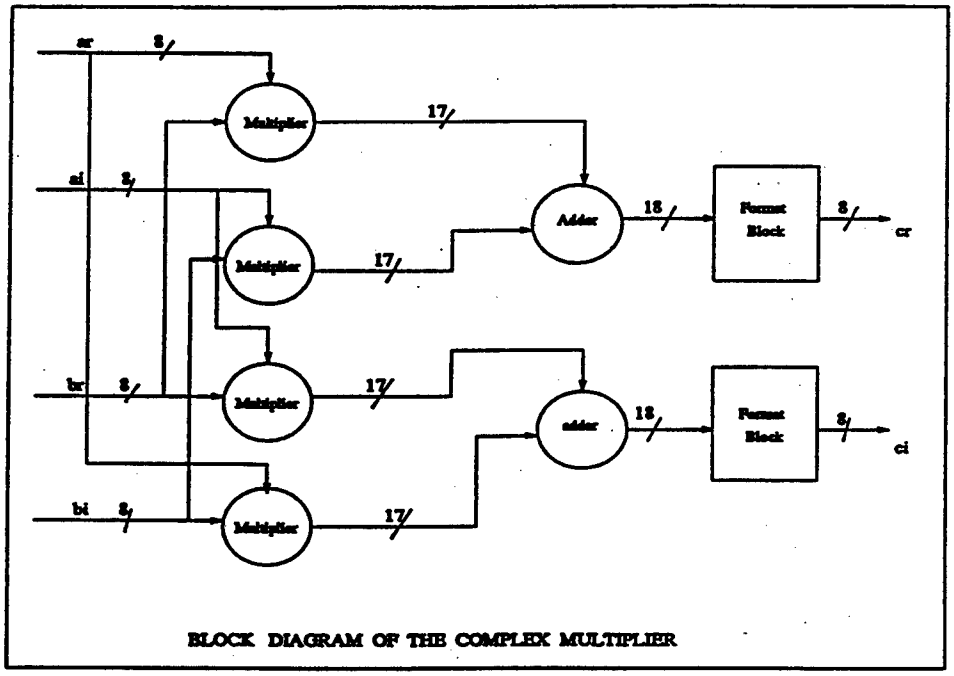


Figure 5.26 - The Block Diagram of Complex Multiplier

The same procedure described in the previous section is carried out for implementing the multiplier on the FPGA. There are two clock cycles available for computing the complex multiplication of the inputs and the coefficients. A pipelined architecture is used in the data path which helps to increase the operating clock frequency of the multiplier. The product of the sample and coefficient provides real and imaginary parts each of which are 8 bits wide. Since the width of memory bus is only 8 bits, we require two clock cycles to write the complex product. The real and imaginary parts are multiplexed using a two to one

multiplexer, with the select line provided by a modulo two counter which is clocked four times slower than the system clock. The two to one multiplexer, the modulo two counter, and the complex multiplier are designed as a single module. The speed of operation of the module is determined by the delay involved in designing the complex multiplier. The time required to perform the complex multiplication is 105ns. The data path delay in the multiplier limits the frequency of operation. Reducing this delay helps in achieving higher speeds of operation.

5.5.7 Read Only Memory Design

The coefficients used during the computation of the FFT need to be stored since they are used repeatedly for processing of the signal samples. The coefficients are stored in a ROM since the coefficients do not change during computation of the FFT. The control logic and ROM to store the coefficients are designed to target a single FPGA.

The design of the ROM consists of a look up table of 14 addresses, since there are only 7 different coefficients required for computing a 16 point FFT using radix 4 algorithm. Access to the table is accomplished with the help of integers and requires a bit vector to integer conversion. The diagram showing the design of the 14 x 8 ROM is shown in Figure 5.27.

The inputs to the ROM module are the clock, the output enable and four address lines. The output of the ROM module is an 8 bit wide coefficient. Two memory locations are used to store the 16 bit complex coefficient. The address lines are given to the bit vector and decoder section of the module which converts the binary address into integers to access the look up table, in which the coefficients are stored. The output section of the ROM module consists of a sequential element which is enabled by the input and triggered by an external clock. The coefficient is written to the output data bus only on the rising edge of the clock and when the enable signal is high - this helps in interfacing the ROM with the control logic unit.

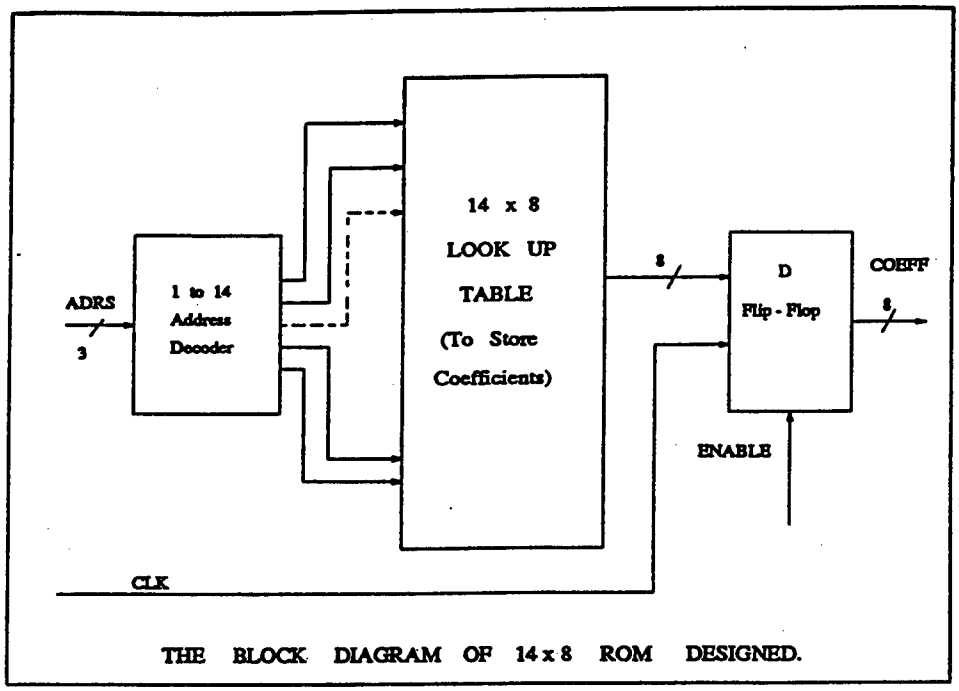


Figure 5.27 - The Block Diagram of Read Only Memory

5.6 Implementation of 64 Point FFT

The algorithm used here is the same as the one, used for implementing 16 point FFT with a few hardware changes. The design uses SRAM, a radix 4 butterfly element, a complex multiplier and control logic. The control logic design differs slightly from the previous design in that there are more samples to process; requiring changes in the generation of read and write addresses. Since a standard memory chip is addressed, care is taken to generate appropriate read and write enable.

Since the design makes use of the same radix-4 butterfly element and complex multiplier, the real challenge of the design is to develop the necessary control logic to access the memory. A different sized ROM is also required to store the coefficients which are different from the coefficients used in the 16 point FFT design. As more computations are necessary in the case of the 64 point FFT, the design runs slower than the 16 point FFT. Therefore, as the number of points in the FFT computation increases, the time required for the computation also increases. The focus of the design is mainly on the control logic unit, therefore, the next section describes with the design details of the control logic required to integrate SRAM, ROM, radix-4 butterfly element and complex multiplier.

5.6.1 Control Unit Design

The control unit can be broken up into several sub units. One important sub unit is the control unit for memory which generates the read and write addresses used to access the memory locations of the SRAM. The read and write addresses are 13 bits wide and the design requires four phases of read operations and three phases of write operations. Counters are designed for generating the read and write addresses for the read and write operations respectively. The SRAM used is a dual port memory, which means that write and read operations can be done simultaneously but to different memory locations. According to the design requirements, the data is received continuously from the A/D, so one port of the memory is used exclusively for writing the incoming data. The other port of memory is used for reading the buffered input and writing back the intermediate computed samples. To do this we multiplex the outputs of the read and write counters to obtain a single bus for addressing the second port of the memory. The read counters are enabled for reading eight memory locations and are disabled for the time duration of the computation and time required for writing the eight intermediate computed samples back into memory. In the same manner the write counters are enabled for writing eight samples. During the three read and write phases, we carry out the same logic except during the fourth read phase, when the FFT is computed, and during which 128 memory locations are read.

For the read operation to be accomplished from the SRAM, we need the read signal to be high, the corresponding port enable to be low and the appropriate address required to access the memory location. The write operation is slightly different, as the data for a specific memory location is written on the rising edge of the write signal, hence care should be taken not to violate the setup and hold time requirements of the device. The control unit is modeled in VHDL, using behavioral and data flow styles. The code is simulated using the Mentor Graphics simulation tool to test the functionality of the control unit.

ROM address generation is another sub unit function. The ROM holds the coefficients which are used as inputs to the complex multiplier along with the outputs of the radix-4 butterfly element. The address bus is 6 bits wide and it is generated by the

control unit. The unit also generates the chip enable, output enable and read signal for the ROM. The coefficients stored are the twiddle factors which are used in the multiplication operation with the output of the radix 4 element.

The buffer control unit is another sub unit which is used to buffer the samples read from the memory and send them to the radix-4 butterfly element. Since we can read only one sample from memory at a time, and the radix-4 element requires four sample, this unit buffers the samples and then send the samples to the butterfly element.

Appendix A

Details of The Design Flow

This chapter of the appendix gives the details of a simple design to illustrate the design flow adopted.

A.1 Description of The Design

The first step in the flow is to describe the design using a language or a schematic editor. The example code below is for a simple PN sequence generator. The number of delay elements used is 4 and hence the length of the codes that can be generated is $2^4 - 1$.

```
-- Define External LIbraries

LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
LIBRARY SYNOPSYS;
USE SYNOPSYS.ATTRIBUTES.ALL;
```

```
ENTITY pnseq IS
     PORT( clear,Clock : IN BIT;
           pnout : OUT BIT;
           clkout : OUT BIT);
     END pnseq;

ARCHITECTURE pn_behav OF pnseq IS

      SIGNAL cnt4,cout,dout : BIT_VECTOR (3 DOWNTO 0);
      SIGNAL ENB1,ENB2 : BIT ;

    BEGIN

       pro1 : PROCESS(clear,Clock,ENB1)
              BEGIN
                 IF(clear='1') THEN
                   cnt4 <= "0000";
                     ELSIF(Clock'event AND Clock='1') THEN
                         IF(ENB1='1') THEN
            cnt4(0) <= NOT (cnt4(0));
            cnt4(1) <= cnt4(1) XOR cnt4(0);
            cnt4(2) <= (cnt4(2) XOR (cnt4(1) AND cnt4(0)));
            cnt4(3) <= (cnt4(3) XOR (cnt4(2) AND cnt4(1) AND
                                cnt4(0)));
                 END IF;
                     END IF;
```

```
        END PROCESS pro1;

      cout <= cnt4;

pro2 : PROCESS(cout)
       BEGIN
    ENB1 <= NOT (cout(0) AND cout(1) AND
                              cout(2) AND cout(3));
    ENB2 <= (cout(3) AND (NOT cout(2))
                AND cout(1) AND (NOT cout(0)));
       END PROCESS pro2;

pro3 : PROCESS(clear,Clock,ENB2)
       BEGIN
          IF(clear='1') THEN
            dout <= "0000";
          ELSIF(Clock'event AND Clock='1') THEN
              IF(ENB2='1') THEN
            dout <= "1000";
                 ELSIF(ENB2='0') THEN
                     dout(0) <= dout(3) XOR dout(2);
                     dout(1) <= dout(0);
                     dout(2) <= dout(1);
                     dout(3) <= dout(2);
              END IF;
        END IF;
```

```
        END PROCESS pro3;

            clkout <= Clock;
            pnout <= dout(3);

END pn_behav ;
```

A.2 Simulation and Synthesis of the design

The second step in the flow is to simulate the design. If the design is developed using the SPW schematic editor, the design is verified in the simulation environment provided by SPW. If the design is described using a hardware description language, the design is simulated using Synopsys simulation environment. Once the design is verified functionally, then it is synthesized using the Synopsys FPGA compiler. Below is the script file used for synthesizing the PN sequence generator.

```
/* Set the top level module names for the design */

   TOP = pnseq

/* set the designer and company name for documentation */

   designer = "rao"
   company = "ITTC"
```

```
/* Analyze and Elaborate the design file and specify the
                        design format */

   analyze -format vhdl TOP  + ".vhd"
   elaborate TOP

/* Set the current design to the top level */

   current_design TOP

/* Add pads to all ports , change the default
                        slew rate to SLOW */

   set_port_is_pad {clear,Clock,pnout,clkout}
   uniquify
   insert_pads

/* set the timing constraints */

   create_clock Clock -period 50

/* Compile Design */

   compile -map_effort med

/* Save Design report file */
```

```
    report_fpga > TOP + ".fpga"
    report_timing > TOP + ".timing"

/* Write out the design to a DB file */

    write -format db -hierarchy -output TOP + ".db"

/* Replace CLBs and IOBs with gates */

    replace_fpga

/*  Set the part type */

    set_attribute TOP "part" -type string "4013pg208-4"

/* Save design in the XNF format as <design>.sxnf  */

    write -format xnf -hierarchy -output TOP + ".sxnf"

/* Exit the compiler  */

  exit
```

A.3 Implementation of the design

As mentioned earlier, the Aptix MP3 board is used for implementing all the designs developed. The Aptix MP3 hardware platform is supported by software to accomplish the reprogramming. The input for the software is a top level netlist of the design in a standard XNF or SCICARD format. The file below gives the top-level netlist file for the PN-sequence generator in the XNF format.

```
LCANET, 4

SYM, OSC, OSC, =REFDES=CLOCK1, =PKG_TYPE=DIP4_3
PIN, CLK, O, ioclk,, =#=1
PIN, GND, I, GND,, =#=2
PIN, VCC, I, VCC,, =#=3
PIN, GND, I, GND,, =#=4
END

SYM,IFIL,4013PQ208-4,=REFDES=FPGA1,FILE=pnseq1.xnf,
              =PKG_TYPE=MP_XC_MQ208
PIN, IRESET, I, buf2pad3
PIN, IOutData, O, IOutData
END

SYM,QFIL,4013PQ208-4,=REFDES=FPGA2,FILE=pnseq2.xnf,
 =PKG_TYPE=MP_XC_MQ208
PIN, QRESET, I, buf2pad3
```

```
PIN, QOutData, O, QOutData
END

SYM,IOFPGA,4013PG223,=REFDES=IOFPGA,FILE=pnckio.xnf,
   =PKG_TYPE=APMP3_XX1IO
PIN, buf2pad1, I, IOutData
PIN, buf2pad2, I, QOutData
PIN, buf2pad3, O, buf2pad3
PIN, ioclk,    I, ioclk
END

PWR, 0, GND
PWR, 1, VCC
EOF
```

Appendix B

Details of Design of Control Logic Unit

B.1 Memory Interface Signals

This section gives the details of the signals used for interfacing with the standard SRAM chip.

B.1.1 Port 1 address, enable & read/write signals

As the SRAM used in our case is a dual port, that is there are two ports available for reading and writing. The address generated for port one are 0-F(hex). The port is used only for writing the incoming data. The enable signal for port 1 is always low, as data is coming in continously. The data coming in is to be written into the memory and is done on a rising edge. The write signal is obtained from inverting the clock used for driving the write address counter that generates address for port 1 of memory.

B.1.2 Port 2 address, enable & read/write signals

The address of second port of memory is obtained by multiplexing five address counters. The five address counters generate read and write address, that are used access data in the memory or write data into the memory. The details of the counters are given below :

- The first read counter reads the data that is stored in the memory for processing the data in first stage. The sequece in which the counter counts is as follows, all the address are in hex representation:
  00, 04, 08, 0C, 01, 05, 09, 0D, 02, 06, 0A, 0E, 03, 07, 0B & 0F.
  During the first read only four samples are read at a time, hence there are four read cycles in which four samples are read each time.

- The first write counter writes the data that is obtained after the processing through first stage. The sequence in which the counter counts is as given below and all the address are in hex :
  10, 11, 18, 19, 20, 21, 28, 29, 12, 13, 1A, 1B, 22, 23, 2A, 2B, 14, 15,
  1C, 1D, 24, 25, 26, 2D, 16, 17, 1E, 1F, 26, 27, 2E & 2F.
  During the write, eight samples are written one after another and there are four such write cycles.

- The second read counter reads data from the memory for processing data in second stage. The address generated by the counter are as follows : 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 2A, 2B, 2C, 2D, 2E, & 2F.
  In the second read phase eight samples are read one after another for processing. They are a total of four read cycles each reading eight samples.

- The write counter generates addresses to write the data processed in the second stage. The sequence of counting is identical to the second read count. The sequence is as follows:
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 1A, 1B, 1C, 1D, 1E, 1F,
  20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 2A, 2B, 2C, 2D, 2E, 2F
  The second write counter writes eight samples in the corresponding locations

- The final counter is used to read the samples whose DFT has been computed. The counter reads 32 memory locations continuously. The sequence in which the counter reads the samples is as follows:
  10, 11, 18, 19, 20, 21, 28, 29, 12, 13, 1A, 1B, 22, 23, 2A, 2B, 14, 15,
  1C, 1D, 24, 25, 26, 2D, 16, 17, 1E, 1F, 26, 27, 2E, 2F

B.1.3 ROM enable and address signals

The ROM used to store coefficients consists of 14 memory locations, each 8 bits wide. The enable and address lines are used to access the coefficients stored in the ROM. The signals are described below:

- The coefficients stored in memory are written to the output bus on the rising edge of the clock. The clock for the ROM is derived from a modulo-2 counter..

- The output of the ROM is written to the output when the output enable signal is high.

- The address bus is 4 bits wide and is used to access the look-up table. As mentioned earlier, the depth of the look-up table is 14 and the coefficients stored in the table are 8 bits wide.

Appendix C

Bibliography

[1] Digital Signal Processing in FLEX Devices, ALTERA Product Information Bulletin 23, See *http://www.altera.com*.

[2] Leo Petropoulos, "Replace Digital Signal Processors with HCPLDs," *Electronic Design*, September 5, 1995, pp. 99 - 104.

[3] Steven Knapp, "Using Programmable Logic to Accelerate DSP Functions", Xilinx Application Note, 1995.

[4] Eric. L. Upton and Thomas J. Kolze, "Reconfigurable Modems Serve as Multi-Application Communications Node Integrators," *1993 Conference of the American Institute of Aeronautics and Astronautics*, pp. 1 - 3.

[5] Rupert Baines, "The DSP Bottleneck," IEEE Communications Magazine, May 1995, pp. 46 - 54.

[6] Russell Petersen and Brad Hutchings, "An Assessment of the Suitability of FPGA Based Systems for Use in Digital Signal Processing," *Field Programmable Logic and Applications: Proceedings of the 5th International Workshop,* FPL-95, Oxford, United Kingdom, August/September 1995, pp. 293 - 302.

[7] A. Lawrence, A. Kay, W. Luk, T. Nomura, "Using Reconfigurable Hardware to Speed up Product Development and Performance," *Field Programmable Logic and Applications: Proceedings of the 5th International Workshop,* FPL-95, Oxford, United Kingdom, August/September 1995, pp. 111 - 117.

[8] S. Kotta and S. Simanapalli, "Rapid Prototyping of a Digital Signal Processor," *Field Programmable Logic and Applications: Proceedings of the 5th International Workshop,* FPL-95, Oxford, United Kingdom, August/September 1995, pp. 844 - 847.

[9] Paul Dunn, "A Configurable Logic Processor for Machine Vision," *Field Programmable Logic and Applications Proceedings of the 5th International Workshop,* FPL-95, Oxford, United Kingdom, August/September 1995, pp. 68 - 77.

[10] L.E. Turner and P.J.W Graumann, "Rapid Hardware Prototyping of Digital Signal Processing Systems using Field Programmable Gate Arrays," *Field Programmable Logic and Applications Proceedings of the 5th International Workshop,* FPL-95, Oxford, United Kingdom, August/September 1995, pp. 129-138.

[11] Joe Mitola, "The Software Radio Architecture," IEEE Communications Magazine, May 1995, pp. 26 - 38.

[12] Bernie New, "A Distributed Arithmetic Approach to Designing Scalable DSP Chips," EDN, August

[13] Satish Mohankrishnan and Joseph B. Evans, "Automatic Implementation of FIR Filters on Field Programmable Gate Arrays", IEEE Signal Processing Letters, March 1995.

[14] Joseph B. Evans, "Efficient FIR Filter Architecture Suitable for FPGA Implementation", IEEE Trans. Circuit & Systems, July 1994.

[15] Chi-Jui, Satish Mohanakrishnan and Joseph B. Evans, "FPGA Implementation of Digital Filters", Proceedings of the. 1993 International. Conference on. Signal Processing. Applications. & Technology.

[16] Joseph B. Evans, "An Efficient FIR Filter Architecture", Proceedings of the 1993 IEEE Int. Symposium on Circuit & Systems.

[17] Satish Mohanakrishnan and Joesph B. Evans, "A Framework for the Design of High Speed FIR Filters on FPGAs", Proceedings of the. 1994 International. Conference on Signal Processing. Applications & Technology

[18] Yong Ching Lim, Joseph B. Evans, Bede Liu, "An Efficient Bit-Serial FIR Filter Architecture", Circuit Systems and Signal Processing, May 1995.

[19] Henry Verheyen and Greg Lara, "Rapid Prototyping Using System Emulation Technology for DSP Design Validation", An article by Aptix Corporation, May 1996. 73

[20] Wayne Wilson and Michel Courtoy, "Rapid Prototyping of Telecommunication ASICs with FPGAs", An article by QUALCOMM Inc and Aptix Corporation, 1996.

[21] Michel Courtoy, Aptix Corporation, "RAPID Verification of ASIC-Based Designs", Wireless Systems Design, October 1996.

•U.S. GOVERNMENT PRINTING OFFICE: 1999-610-130-81070

MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)

The advancement and application of information systems science and technology for aerospace command and control and its transition to air, space, and ground systems to meet customer needs in the areas of Global Awareness, Dynamic Planning and Execution, and Global Information Exchange is the focus of this AFRL organization. The directorate's areas of investigation include a broad spectrum of information and fusion, communication, collaborative environment and modeling and simulation, defensive information warfare, and intelligent information systems technologies.