12-1996

Evaluation of Design Tools for Rapid Prototyping of Parallel Signal Processing Algorithms

James C. Savage

EVALUATION OF DESIGN TOOLS FOR RAPID PROTOTYPING OF PARALLEL SIGNAL PROCESSING ALGORITHMS

THESIS

James C. Savage, Captain, USAF

AFIT/GE/ENG/96D-18

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/96D-18

EVALUATION OF DESIGN TOOLS FOR RAPID PROTOTYPING OF PARALLEL SIGNAL PROCESSING ALGORITHMS

THESIS

James C. Savage, Captain, USAF

AFIT/GE/ENG/96D-18

AFIT/GE/ENG/96D-18

EVALUATION OF DESIGN TOOLS FOR RAPID PROTOTYPING OF PARALLEL SIGNAL PROCESSING ALGORITHMS

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Electrical Engineering

James C. Savage, B.S.E.E.

Captain, USAF

December 1996

Approved for public release; distribution unlimited

Acknowledgments

I would like to thank those who have helped me in this research. I thank my thesis advisor, Dr. Gary Lamont, for his effort to find an area of research which not only aligned with my experience, but also provided me the opportunity to investigate other areas previously unfamiliar to me. Dr. Lamont provided the right amount of suggestion, encouragement ,and freedom for me to make the most out of the research experience. I thank my sponsor, Dr. Robert Ewing, for his insight into current Wright Laboratory signal processing needs and the accompanying motivation. I also would like to thank the members of my thesis committee. My topic has taken many twists and turns over the past year, and while I may have lost some along the way, each was helpful at different stages of the journey. The weary travelers of my thesis committee are Dr. Andrew Terzuoli, Lt. Col. Tom Wailes, Dr. Ken Stevens, Dr. Bruce Suter, and Lt. Col. David Gallagher. Thanks also go to Greg Richardson for his last minute software installation efforts.

Many thanks go to the other VLSI lab rats. With the help and camaraderie of Captain Javier Marti, Captain Jeff Butler, Captain George Dalton, and Lieutenant George Rohlke, the VLSI sequence and the endless hours spent staring at a computer screen in the laboratory were almost bearable.

Finally, I must thank my roommates, Lieutenant Ron Schwing and Captain Dwight Smith, for leaving me alone when I needed to get some work done.

Table of Contents

List of Figures

List of Tables

Abstract

Digital signal processing (DSP) has become a popular method for handling not only signal processing, but communications, and control system applications. A DSP application of interest to the Air Force is high-speed avionics processing. The real-time computing requirements of avionics processing exceed the capabilities of current single-chip DSP processors, and parallelization of multiple DSP processors is a solution to handle such requirements. Designing and implementing a parallel DSP algorithm has been a lengthy process often requiring different design tools and extensive programming experience. Through the use of integrated software development tools, rapid prototyping becomes possible by simulating algorithms, generating code for workstations or DSP microprocessors, and generating hardware description language code for hardware synthesis. This research examines the use of one such tool, the Signal Processing WorkSystem (SPW) by the Alta Group of Cadence Design Systems, Inc., and how SPW supports the rapid prototyping process from an avionics algorithm design through simulation and hardware implementation. Throughout this process, SPW is evaluated as an aid to the avionics designer to meet design objectives and evaluate trade-offs to find the best blend of efficiency and effectiveness. By designing a two-dimensional fast Fourier transform algorithm as a specific avionics algorithm and exploring implementation options, SPW is shown to be a viable rapid prototyping solution allowing

an avionics designer to focus on design trade-offs instead of implementation details while using parallelization to meet real-time application requirements.

EVALUATION OF DESIGN TOOLS FOR RAPID PROTOTYPING OF PARALLEL SIGNAL PROCESSING ALGORITHMS

I. Introduction

Digital signal processing (DSP) processors are in wide use in communications, signal processing, and control system applications. These high-speed, single-chip microcomputers are specifically designed for handling computationally intensive tasks in lieu of using conventional microprocessors [1:482]. A DSP application of interest to the Air Force is high-speed avionics. As computing demands of existing and emerging DSP applications continue to increase, the current single-chip DSP processor technology can no longer keep pace. While parallelization of multiple DSP processors is a solution to increase throughput and speed beyond the capabilities of a single processor for the demanding requirements of avionics signal processing, designing and implementing parallel DSP processor systems can be a lengthy process. Rapid prototyping of parallel architectures is possible through the use of software tools such as the Signal Processing WorkSystem (SPW) [2]. The purpose of this thesis effort is to evaluate the rapid prototyping process to design and implement parallel DSP algorithms using SPW and SPW's applicability for high-speed avionics applications.

Background

Real-world DSP is the filtering of signals in real-time [1:3]. Analog to digital conversion[1] takes place either on-chip or off-chip after which manipulation of the digital signal simplifies to computational operations. The digitized results can then be converted back to analog signals via digital to analog converters. To process real-time data, both the converters and the DSP processors must possess the speed and throughput necessary for continuous processing. However, computing requirements are surpassing the capabilities of single-chip DSP processors. For example, algorithms that integrate speech coding/decoding into a multi-media environment require 1-30 million instructions per second (MIPS) which is well within the 30-50 MIPS capability of today's computer workstation [3:269]. Emerging applications, such as video coding/decoding and medical imaging algorithms, require 0.1-10 billion instructions per second [3:269]. Current DSP processors are not capable of handling these requirements. Thus, parallel architectures such as those used for shipboard radar systems may contain as many as 1000 processors to handle the tremendous processing and throughput requirements [4]. Design of such systems can now take advantage of emerging DSP development tools to exploit the capabilities of a DSP processor in parallel architectures.

Of the DSP development tools available today, AFIT and Wright Laboratory (WL) have access to Signal Processing WorkSystem (SPW). SPW is an integrated software environment for developing, simulating, and implementing DSP systems. SPW consists of a number of components to accomplish these tasks as designed and

[1] Analog to digital conversion is converting an analog input consisting of a voltage or current to digital output binary word.

implemented by the Alta Group of Cadence Design Systems [2]. Figure 1 illustrates these components.

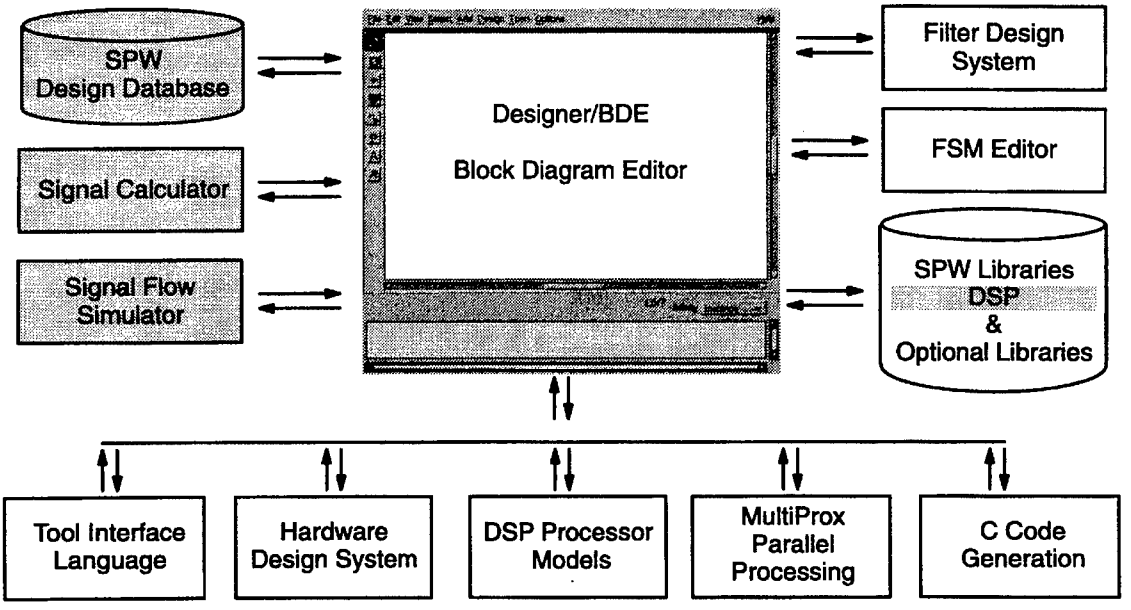

Figure 1 - SPW Components [2:iii]

DSP systems are interactively designed using blocks from the SPW libraries or blocks created using the Filter Design System (FDS) or Finite State Machine (FSM) Editor. System block diagrams are designed in the Block Diagram Editor (BDE) with interconnections to represent data flow. Simulation is performed through the Signal Flow Simulator with signal analysis using the Signal Calculator. The Code Generation System (CGS) option of SPW adds the capability to automatically generate C code for execution on workstations or DSP processors [2]. The addition of the MultiProx (MPX) option gives SPW the capability to partition a design among multiple processors for

multiprocessing simulations or downloading to a development board containing multiple processors [2].

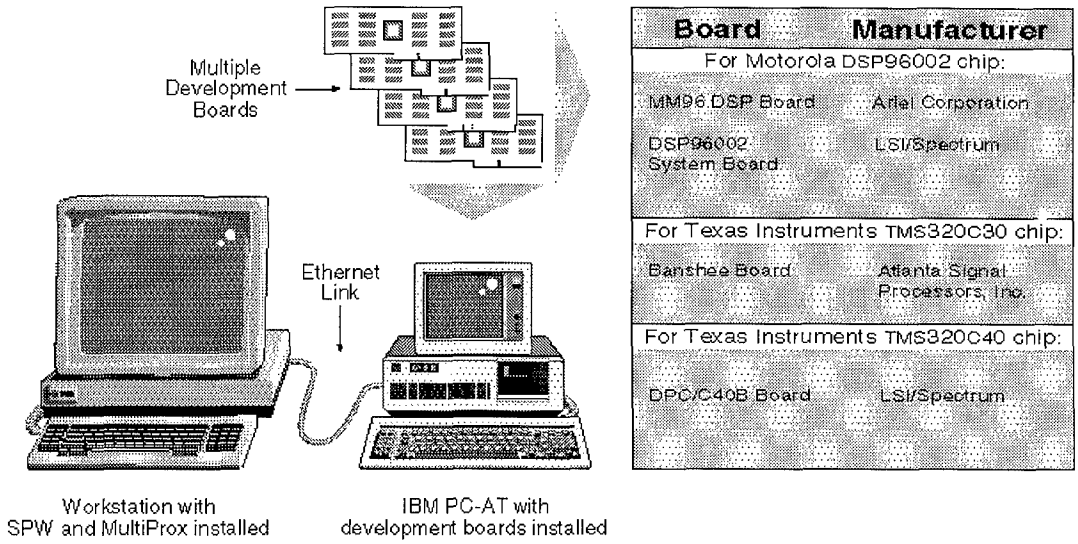

Figure 2 - SPW MultiProx [2]

Another component of SPW, the Hardware Design System (HDS), generates VHDL[2] descriptions with when used with VLSI synthesis tools, can result in a layout for fabrication of an application specific integrated circuit (ASIC) [2,5]. The combination of the tools available within the SPW environment supports rapid prototyping as well as parallel implementations for signal processing.

[2] VHDL is the VHSIC Hardware Description Language where VHSIC is Very High Speed Integrated Circuit. VHDL is a standard language for the specification of hardware behavior and structure for very-large scale integration design [5].

Problem Statement

Real-time processing of data is a requirement for avionics signal processing applications. However, the speed and throughput requirements for real-time signal processing currently exceed the capabilities of single-chip DSP processor implementations. Moreover, the current development time from DSP algorithm selection to design simulation to testable hardware implementation often prohibits efforts to make design changes and evaluate trade-offs.

The objective of this thesis investigation is to evaluate the rapid prototyping process from algorithm selection through design, simulation, and hardware test of parallel DSP architectures for avionics signal processing using the SPW environment. This is accomplished by first choosing a general algorithm representative of those used in high-speed avionics applications. The algorithm is then designed at a high-level using functional blocks and interconnections using the SPW environment for simulation and analysis. Simulation and analysis of the simulation results are performed to determine whether or not the algorithm functions and performs as required based upon acceptance criteria. With the use of the code generation capabilities of SPW, porting to a workstation and a development board (Texas Instruments) for hardware/software testing is investigated. Finally the capability of the HDS component of SPW to generate VHDL is evaluated. Throughout the rapid prototyping process, the SPW environment is evaluated for ease of learning, ease of use, and functionality and to what extent SPW allows a designer to evaluate design and implementation trade-offs.

Rationale

Why is it important that signal processing system designers take advantages of emerging software development tools? The primary reason is the time savings involved in rapid prototyping. Instead of spending time writing code[3] which is in most cases an exercise of translating the functions of an abstract system design, a designer can spend more time analyzing the design by making trade-offs in cost and performance to improve efficiency and effectiveness [6]. Through rapid prototyping, the process of design, simulation, and hardware test can be completed efficiently so that more time may be spent refining the algorithm and addressing design issues such as numeric formats and sampling rates rather than being bogged down with low-level implementation details. If necessary, extra performance may be gained by fine tuning the automatically generated code by hand once the design of the algorithm has been decided. Another benefit of rapid prototyping is the ability to test a system in real time on actual hardware which not only speeds up the testing process, but also reflects a better representation of the final product. The time savings achieved through rapid-prototyping allows a designer to spend more time examining and evaluating design trade-offs. These design trade-offs include parallel partitioning strategies and data handling strategies to include input, output, and storage. Other design issues include power consumption, in-circuit testing capability, and redundancy. Rapid prototyping time savings provides more time to be spent examining these design issues.

[3] Code or more appropriately, source code is the form in which a computer program is written by the designer in a formal programming language which is subsequently compiled automatically into a machine-recognizable code [6]

Scope

This research focuses on the capability of SPW to design, simulate, and implement a parallel system of DSP processors using a rapid prototyping methodology. Synthesis is attempted on unmodified VHDL code or portions of the VHDL code which are synthesizable. A selected avionics algorithm is used throughout the process. Design trade-offs of this algorithm are evaluated through the use of parallel processing metrics [7].

Standards

Evaluating computing performance is critical to analyzing different architectural approaches. Performance metrics include run time, speedup, efficiency (time and space), cost and the isoefficiency metric of scalability [7:117-141]. Each of these metrics is defined in Kumar's Introduction to Parallel Computing [7].

Methodology

The following subsections identify and describe the tasks that comprise this research effort:

Literature Review

The literature review, Chapter II, is a continuing process to examine the areas of high-speed avionics signal processing, parallel DSP architectures, and DSP software development tools. The goal of a literature review of high-speed avionics signal processing is an understanding of the classification of avionics applications and the

requisite computing requirements for a particular classification. The literature review of current efforts to parallelize DSP applications enables selection of an architecture to support a particular avionics signal processing applications. A survey of DSP software development tools serves both to identify the most promising environments in use today for rapid prototyping and to allow comparisons with the SPW environment. In an effort to locate information in these areas, the World Wide Web (WWW) and associated WWW search engines are used extensively. The growth of the DSP industry has been accompanied by a proliferation of sites on the internet relating signal processing. Finally, factors to consider while evaluating DSP software development tools such as SPW are described.

Algorithm Selection and Design

In order to demonstrate the process of taking an algorithm through the rapid prototyping process, an algorithm representative of those used in high-speed avionics signal processing is chosen. In addition to avionics signal processing applicability, selection is based upon algorithm complexity so design and test may be completed within the time allotted for this research. Chapter III, Algorithm Selection and Design, provides background on the chosen algorithm, the two-dimensional fast Fourier transform, and describes the design and simulation of the algorithm in SPW. SPW is evaluated for whether or not it provides adequate functionality to allow a designer to make design trade-offs while providing a proper interface.

Design Implementation

Chapter 4, Design Implementation, investigates the three different implementation options available in SPW. From the block diagram design of the algorithm, C code can be generated for execution on workstation. Also, C code may be generated for porting to DSP processors for testing on development boards. The potential of Wright Laboratory's (WL) Texas Instruments Quad C40 DSP320 Development Board is assessed. Finally, hardware synthesis support exists through VHDL code generation. Each of these options is analyzed with appropriate metrics applied.

Materials and Equipment

The following materials and equipment are required:

- Sun Workstation (AFIT VLSI Laboratory)
- Signal Processing WorkSystem (SPW) Version 3.0 (AFIT)
    - SPW Hardware Design System (HDS) option
    - SPW Code Generation System (CGS) MultiProx option
- Synopsys VHDL Tools
- Texas Instruments Quad C40 DSP320 Development Board/PC-OS/2 (WL Avionics Laboratory)

Summary

To evaluate the utility of the Signal Processing WorkSystem, an algorithm representative of high-speed avionics applications is implemented using a rapid prototyping methodology. With the parallel processing support, code generation, and hardware synthesis support of SPW, significant time may be saved in the development process while at the same time, an algorithm can be parallelized to improve upon the capabilities of a single-chip DSP processor.

This chapter describes the problem and describes the potential advantages of using software development tools for rapid prototyping. The reader is assumed to have a basic understanding of computer science and electrical engineering concepts. Chapter II discusses the areas of high-speed avionics signal processing, DSP in general, parallel/distributed DSP design, rapid prototyping through DSP software development tools and software evaluation principles. Chapter III describes the algorithm selection and its design in SPW. Chapter IV covers implementation of the design to include C code generation, and HDL generation for hardware synthesis. Finally, Chapter V draws conclusions and presents recommendations on the use of SPW to support a rapid prototyping methodology for high-speed avionics applications.

II. Literature Review

This chapter provides background information on examples of high-speed avionics signal processing and computational requirements, along with engineering to meet these requirements. This is followed by a discussion of DSP processors. Then, an explanation of how parallelism may be used along with descriptions of several current parallel DSP designs is provided. The rapid prototyping methodology is explained and emergence of software development tools which support this methodology is examined. Finally, factors to consider while evaluating hardware/software development platforms are described.

High-Speed Avionics Signal Processing

DSP applications may be classified based upon relative algorithm complexity, required sample rates, clock rates, and numeric formats [8]. Algorithms specify the arithmetic operations but not how the operations are to be implemented at lower levels of detail. Implementation details, for example, include the sample rate or the rate at which samples are consumed, processed, or produced [8]. The ratio of clock rate to sample rate partially determines the hardware required to implement an algorithm with a given complexity [8]. Numeric formats are also a design issue to evaluate the trade-offs between algorithm simplicity and numerical accuracy. The following graph shows the range of the signal processing applications when considering relative algorithm complexity and sample rates:

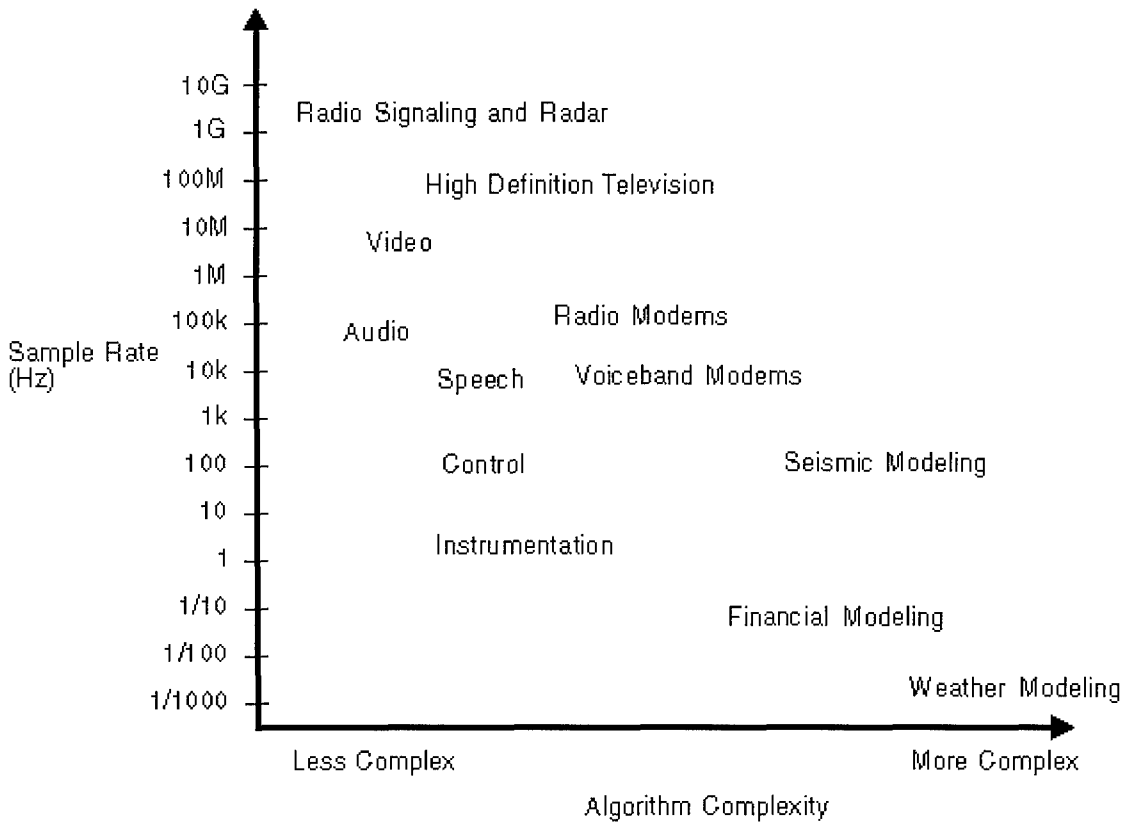

Figure 3 - Relative Sampling Rates and Algorithm Complexities of Signal Processing Applications [8]

High-speed avionics signal processing typically falls in the area of high sample rates with less complex algorithms.

Military avionics signal processing applications include secure communications, sonar processing, image processing, navigation, missile guidance, and radar/sensor processing [1:474]. A DSP application of vital interest to the Air Force in the area of radar/sensor processing is accurate, real-time target recognition through the use of radar cross-section (RCS) images. *Xpatch* [9] is currently used for this purpose, but the real-time computing requirements far exceed the capabilities today. *Xpatchf*, the frequency domain program of *Xpatch*, requires 35 minutes on an Intel i860 computer to calculate the

RCS image of an F-15 aircraft from a single perspective, and a complete image requires 16,800 hours or 700 days of computer time [10:1-2]. While the ultimate goal of real-time RCS prediction some time away, parallelization can offer significant speed-ups in the near term.

The first AFIT research on parallelizing electromagnetic prediction code was conducted by Captain Scott Suhr [11], who demonstrated the speedup possibilities by parallelizing a previously designed serial code. A precursor to *Xpatch*, NECBSC, was modified for execution on an Intel iPSC Hypercube. The results were a speedup of 3.59 on an eight node Intel iPSC/2 over the serial benchmark on the same machine. Speedup on an Intel iPSC/860 was 2.51 due to a faster benchmark, but overall time was reduced by 23 percent [11:xi]. Research continued by Lieutenant Paul Work to parallelize serial ray-tracing code considered factors such as load balancing and decomposition [12:xi], which demonstrated the speedup possibilities using parallel processing with electromagnetic code.

Later research conducted by Captain B. A. Kadrovach focused on the *Xpatch* algorithms by profiling *Xpatchf*, the frequency domain portion of *Xpatch*, to reveal any repetitive functionality and periodicity [10]. This profiling identified the ray-tracing portion of the multi-bounce feature of *Xpatchf* as a candidate for a hardware implementation [10:3-11]. Aspects of the ray-tracing algorithm are computationally independent, enabling many rays to be processed simultaneously. Captain Kadrovach designed a hardware model, called a Voxel Unit (VU), to handle the multi-bounce RCS processing. He envisioned that multiple VUs under the control of a single

microprocessor would be used in parallel to achieve speedups greater than two orders of magnitude when using a network of eight cards each with a four-by-four array of VUs [10: vii,4-2]

The research of Captains Suhr and Kadrovach and Lieutenant Work has shown the advantages parallelism can offer for radar processing. While Captain Suhr demonstrated the speedup possibilities inherent in a serially designed electromagnetic code, Captain Kadrovach focused on a portion of a code and the description of hardware, the VU, to accomplish this portion in parallel. DSP development tools with parallel design support offer the designer the ability to make these design decisions at a high level and automate the implementation.

The fast Fourier transform (FFT) and more specifically, the two-dimensional FFT is another useful algorithm in radar processing applications. Like the ray-tracing algorithm, the 2-D FFT algorithm may benefit from parallelism. Tools such as SPW allow a designer to experiment with different parallel partitioning approaches, data rates, and numeric formats at a block diagram level to determine the best solution for given requirements.

Digital Signal Processing

Real world DSP is the real-time processing of converted analog signals [1:3]. Analog to digital conversion takes place either on-chip or off-chip after which manipulation of the digital signal as bits and bytes simplifies to mathematical operations in a digital computer. The results can then be converted back to analog signals via digital

to analog converters. To process real-time data, DSP processors must possess the speed and throughput necessary for continuous processing to handle incoming data as it becomes available. However, computing requirements now surpass the capabilities of single-chip DSP processors. For example, algorithms that integrate speech coding/decoding into a multi-media environment require 1 to 30 million instructions per second (MIPS) which is well within the 30 to 50 MIPS capability of today's computer workstation [3:269]. Emerging applications, such as video coding/decoding and medical imaging algorithms, will require 0.1 to 10 billion instructions per second [3:269]. Current DSP processors with processing rates from 1 to 30 MIPS are not capable of handling these emerging applications.

Since DSP processors are designed with signal processing in mind, they have capabilities not found in conventional microprocessors. To handle large amounts of data in real time, DSP processors' internal architectures differ from those of conventional microprocessors. For example, the TMS 320 series manufactured by Texas Instruments uses a Harvard-type architecture [16] as illustrated in Figure 4.

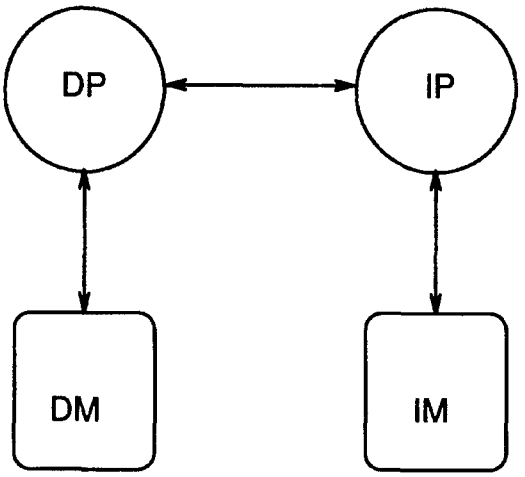

Figure 4 - Harvard Architecture [16:41]

The Harvard architecture has separate address spaces for instructions (IM) and data (DM) to allow for concurrent instruction and data fetching [1:482]. The TMS 320 architecture is different from a conventional microprocessor which must sequentially access instructions and data from a single address space. Another characteristic unique to the DSP processor is the existence of the single-instruction multiply-and-accumulate (MAC) operation [1:482]. DSP algorithms often require a sum-of-products arithmetic operation which is handled by a MAC operation. The use of a MAC operation is illustrated by examining the difference equation of a nonrecursive digital filter of order M:

$$y[n] = \sum B_m * x[n-m] \tag{1}$$

where B_m are filter coefficients [13:614]. This difference equation is an example of the need for a MAC operation. Other features of DSP processors vary depending on the particular manufacturer, but the different internal architecture and presence of the MAC

operation are two characteristics which distinguish the DSP processor from conventional microprocessors.

DSP systems may be classified into different families. These are the Bit-Slice, Word-Slice® (registered trademark of Analog Devices, Inc.), single-chip DSP microcomputers, and DSP microprocessor families. Bit-slice systems were early attempts at DSP parallelism using small but fast subunits to build a required word-length [3:250]. The Bit-Slice approach, which used medium scale integration technology, has since been surpassed by the Word-Slice® approach due to the improvements in very large scale integration (VLSI) technology [3:252]. Word-Slice® systems benefit from fewer components with similar performance. Single-chip DSP microcomputers and microprocessors are designed primarily for individual use, with the single-chip DSP processor the most self-contained [3:252]. An abundance of single-chip DSP applications has led to a very competitive market, and therefore the price-performance ratio for this family of processors is relatively low. Of the four families, the single-chip DSP microcomputer is the best building block for a parallel architecture since it offers the best price-performance ratio using the latest in VLSI technology. Appendix A provides the Pocket Guide to DSP Processors and Cores [14] which provides a sampling of DSP processors and their characteristics to include architectural details, RAM and ROM sizes, and unit prices.

Another characteristic of DSP processors which may be required of a particular application, is low power operation. Portable consumer electronics such as pagers, cellular telephones, personal audio equipment, and laptop computers demand low power

consumption to extend battery life [15]. DSP processors employ several techniques for power reduction. Reducing the supply voltage is one technique since power consumption is proportional to the square of supply voltage [7]. This reduction is possible through the tighter integration of transistors on a chip. Power management features such as sleep/idle modes, clock frequency control, and control over unused peripherals and outputs are also used to reduce power consumption [7]. System and programming techniques to avoid external memory access and unnecessary logic state transitions also help to reduce overall power consumption [7]. The competitive DSP market is forcing manufacturers to use these methods to achieve the longest battery life. While ASICs can be designed to minimize the number of transistors necessary for an application, the existence of power reduction measures on DSP processors makes them attractive for low power applications.

Parallel/Distributed DSP Designs

With parallel processing, speed and throughput of single-chip DSP processors can be surpassed. Simply put, parallel processing techniques may be used to take advantage of the parallelism inherent in many DSP algorithms. That is, the calculations involved in signal processing required are often independent so that the work can be partitioned among two or more separate processors. Parallel processing has long been used to connect multiple microprocessors, and the associated architectures and algorithms developed for microprocessors are applicable to parallel DSP.

Georgia Institute of Technology Digital Signal Multiprocessors

The concepts used in parallel microprocessor architectures may be applied to DSP processors. Continuing research at Georgia Institute of Technology on Digital Signal Multiprocessors (DSMP) systems. At Georgia Tech, two experimental laboratory DSMP systems have been built. The two laboratory systems are known as OSCAR (Optimal Synchronous Cyclo-static Array) and OSCAR-32 [16:293]. These systems represent the first of multiprocessor systems designed for DSP.

The first DSMP prototype at Georgia Tech, OSCAR, was completed in 1986. The OSCAR project was divided into two phases. The first phase, OSCAR I, was a small-scale supercomputer model which consisted of sixteen commercially available processors in a 4x4 rectangular array.

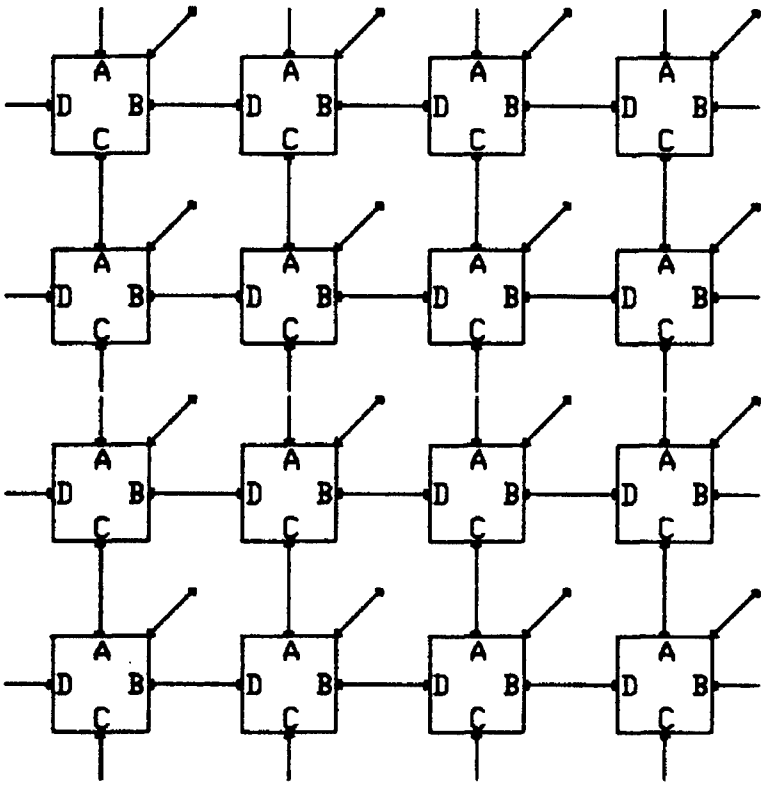

Figure 5 - OSCAR I [16:293]

OSCAR II was to be built using 128 processors, but funding cuts prevented its realization. The OSCAR I was a complex system where each processor contained five fully parallel communication ports, a 32-bit floating-point arithmetic unit, local memory, address generation unit, micro-controller, and a debugger/monitor processor [16:294].

The second DSMP prototype, OSCAR-32, was built at Georgia Tech in 1987. OSCAR-32 is a reconfigurable ring structure of 32-bit processors with each processor resident on a constituent processor boards (CPB). Up to sixteen of the CPBs can be connected to form a ring of processors.

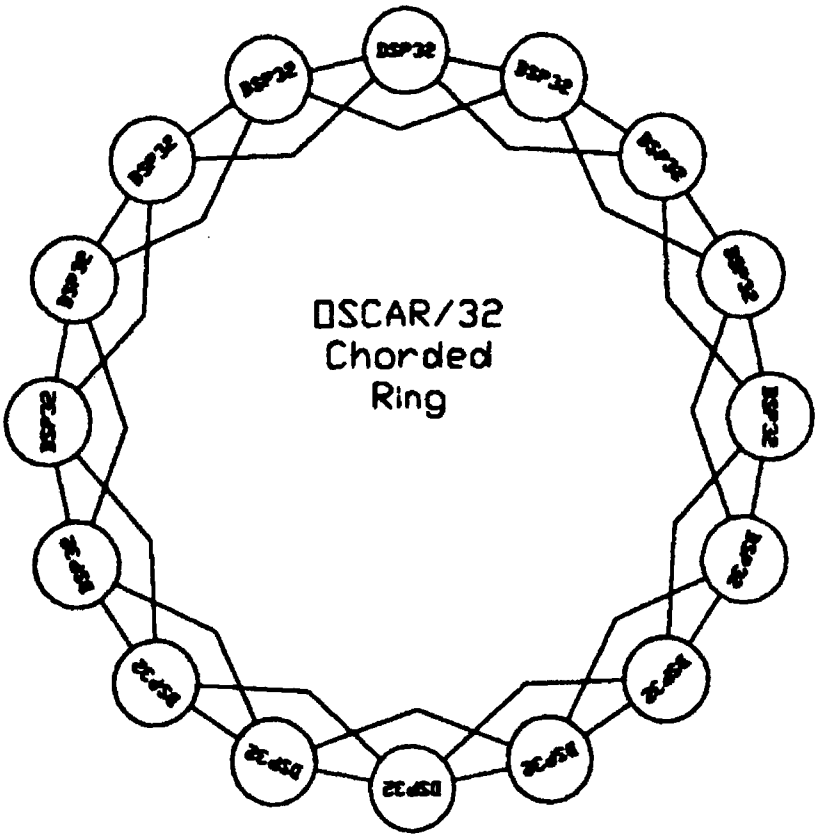

Figure 6 - OSCAR-32 [16:301]

The OSCAR-32 uses the AT&T WE-DSP32 floating-point signal processing microcomputer which provides high-speed processing at the expense of heavy timing penalties for inter-processor communication [16:297].

The experimental DSMPs at Georgia Tech are examples of the use of parallel processing concepts of architecture and algorithm design for signal processing applications. These machines had general purpose signal processing in mind. However,

even within the scope of signal processing, there exists a variety of applications which could benefit from different architectures.

DSP Multiprocessor Architectures

A paper by Bier and Lee [17] describes several abstract multiprocessor architectures for real-time DSP. Each of the architectures contain a memory shared by the individual processors along with a controller processor which grant access to a bus connecting the processors to the shared memory. The Gated-Shared-Memory Architecture uses a gate keeper as a hardware implementation of the semaphore concept [17:299]. To avoid the use of a gate keeper and its complexity, a central controller, called the MOMA (Maintains Ordered Memory Accesses), is used in an Ordered Shared-Memory Architecture.

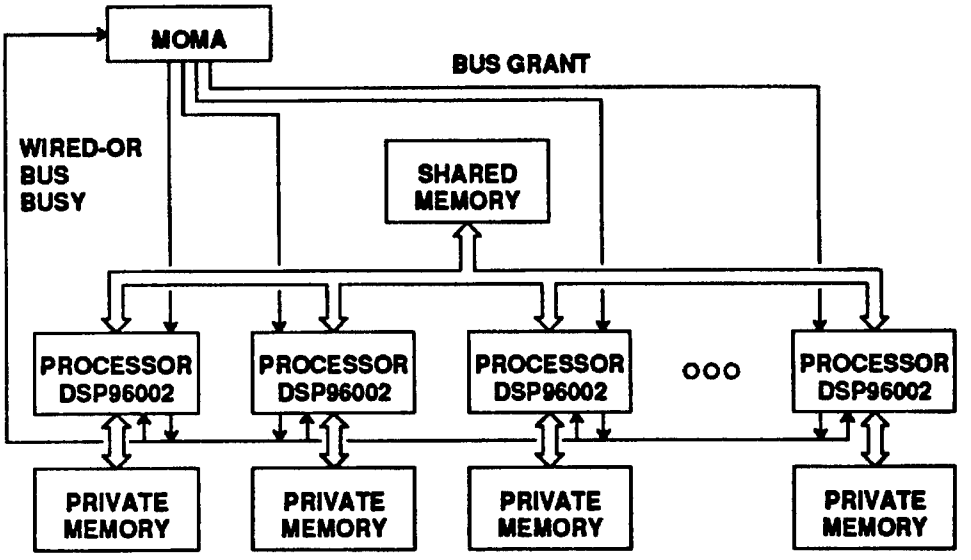

Figure 7 - Bier and Lee Architecture [17:300]

The MOMA takes advantage of a-priori knowledge of shared memory accesses by the processors. The MOMA grants access to the bus and the shared memory in a prespecified order. The main advantage of this scheme is that no explicit hardware or software is required to resolve memory access issues [17:299-300].

Another multiprocessor architecture for DSP is described by Baraniecki and Baraniecki. The architecture is composed of 'N-Clusters' each consisting of 1 to M processor elements. The processors within a cluster share a common main memory for application programs and a common database memory. Memory accesses are handled by two interconnection chips.

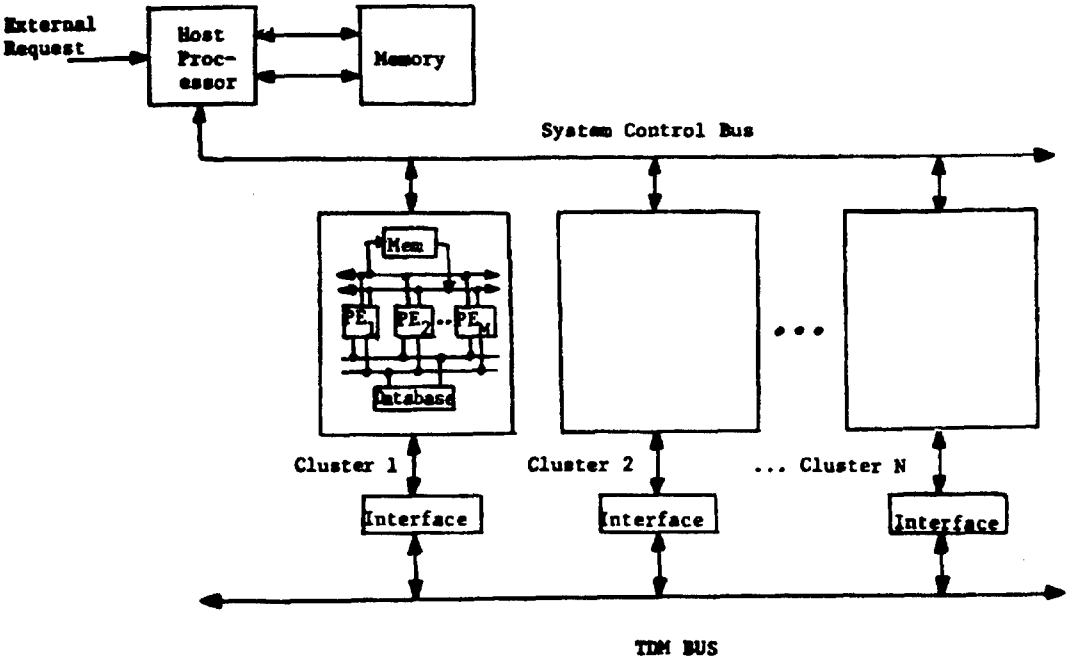

Figure 8 - N-Clusters Architecture [18:94]

The entire system is controlled by a single processor known as the host with a control and data bus interconnecting the different clusters [18:90]. While this architecture has the flexibility to handle a wide range of applications through various partitioning schemes among the different clusters, the repeatability and limited communication requirements of ray-tracing does not require such a complex architecture.

Two hardware examples of parallel DSP systems use dual port memories to handle inter-processor communication.

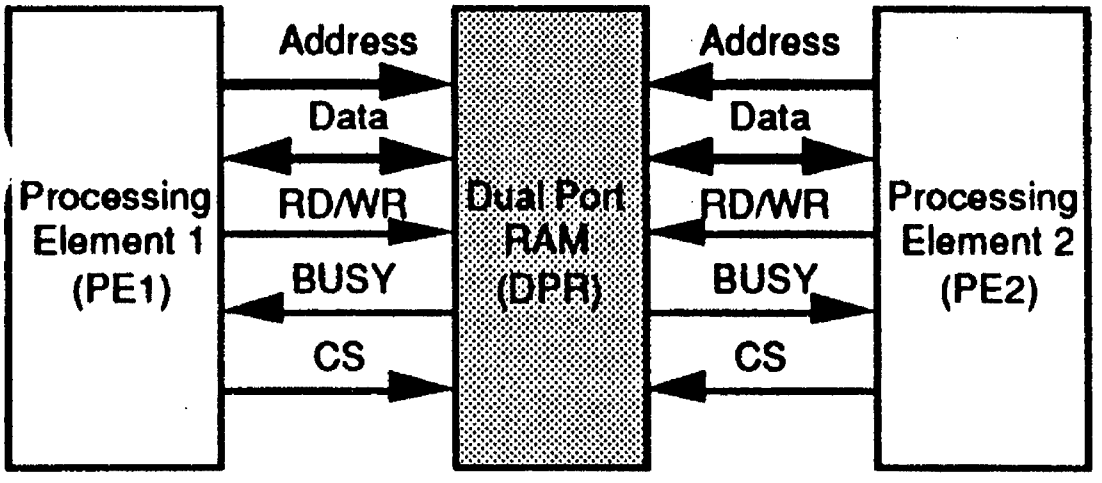

Figure 9 - AdEPAR Dual Port RAM [19:150]

The Advanced Educational Parallel (AdEPAR) DSP system uses boards containing TMS320C25 DSP processors hosted by an IBM PC. The PC host handles communication between processors or from the processors to the host by dual port memory for simplicity, speed, modularity, and configurability [19:149]. A system used for image processing described in [20:494] also makes use of a dual port memory

configuration for message passing and intermediate storage. While both examples use this configuration for higher performance/cost ratios, the number of processors and the amount of communication could hinder performance as processors not adjacent pass messages through intermediate processors. This approach does offer a simple solution for interconnection of multiple processors.

Rapid Prototyping and DSP Development Tools

The rapid growth in the DSP industry over past decade has seen equal growth in the number of software tools and what these tools offer the signal processing system designer. The most sophisticated of these environments allow a designer to realize an algorithm in a hardware prototype or even a VLSI layout in a fraction of the time once required. Benefits of using these tools include version control and automatic design rule checking along with simulation, test data generation, software generation, and hardware synthesis.

A simple methodology for the development of a DSP system can be defined by the design, simulation, and implementation stages. In design, a system based upon a specification is designed at a high level usually graphically. Simulation includes test generation to determine if the design is functioning as intended. Implementation can consist of software generation for a workstation, a network of workstations, or a DSP chip. Implementation can also include generation of hardware description language code like VHDL to be used with synthesis tools for the fabrication of a custom chip. Sophisticated tools provide the means to accomplish each of these stages.

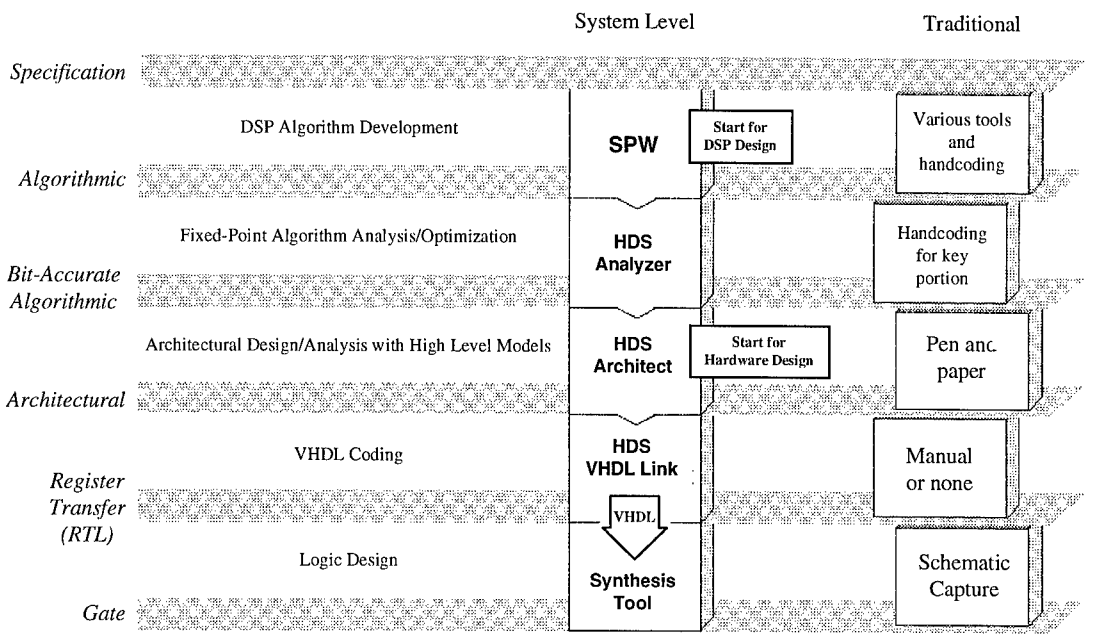

Figure 10 - SPW Development Process

Block diagrams are the most common way to represent a signal processing system. Block diagrams have long been used for documentation and the use of these diagrams in software tools provides a way to describe, document, and analyze a signal processing system. A system block diagram is represented as a network of transfer functions, data sources and sinks, and control functions. The following block diagram, included as part of the SPW tutorials, represents a notch filter[4].

[4] A notch filter is used to attenuate frequencies within a specified band.

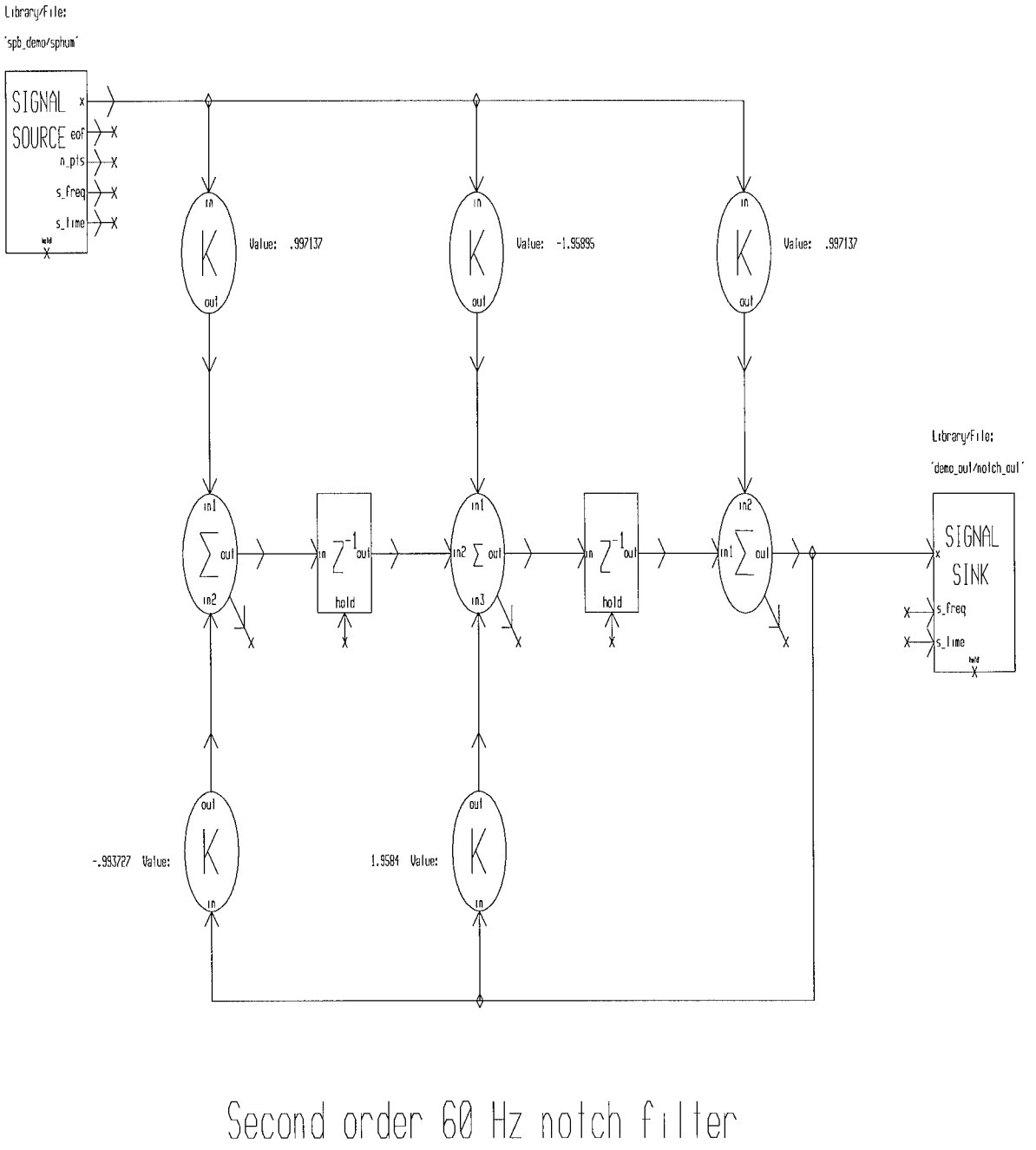

Figure 11 - Sample Block Diagram of the Notch Filter [2]

These blocks are interconnected with lines or arrows to represent data movement. The block diagram can also be referred to as a dataflow graph. Each block in the system has an underlying computational model to define the block's behavior [21].

Simulation of a system is performed using either a synchronous dataflow (SDF) or dynamic dataflow(DDF) model. In SDF, each block consumes or produces a fixed number of samples prior to execution, whereas a DDF block can consume and produce varying numbers of samples on its inputs and outputs [21]. The DDF simulation model is supported by Mentor Graphics' DSP Station and COSSAP. SPW uses the SDF model for simulation. For the purposes of analyzing the data produced in a simulation, the capability data plotting and parameter changing during simulation is advantageous. SPW includes the Signal Calculator for analysis purposes.

While most block diagram based software tools offer simulation capabilities, few have the capability to generate software code and even fewer support hardware synthesis. Tools with hardware synthesis support are the most expensive with costs in excess of $30K compared with PC software tools without synthesis support costing from $500 to $5K [21]. In addition to SPW's HDS tool, DSP Station and COSSAP also support hardware synthesis. Synthesis support can be in the form of either a register-transfer-level (RTL) design or a behavioral level design. All three tools support RTL designs, but only DSP Station and COSSAP produce behavioral descriptions. RTL designs are expressed in terms of RTL elements such as registers, multipliers, and shifters while a behavioral level design dictates nothing about implementation and allows the synthesis tool to handle details [21].

A few software tools support simulation and software generation for parallel systems. These include SPW's MultiProx, Pegasus from Jovian Systems, and RIPPEN from ORINCON Technologies. MultiProx allows a designer to partition a design for

multiple processors. Individual code can then be generated for each processor in the system along with the communication required among the processors in the system. Real-time testing on a development board containing multiple DSPs can be monitored to analyze load balancing. Table 1 contains the uniform resource locators (URLs) for block diagram based signal processing development tools where additional information regarding these products may be obtained.

Table 1 - URLs of publishers of block diagram based signal processing tools

Publisher	Software Package	Publisher URL
Cadence/Alta Group	Signal Processing WorkSystem	http://www.altagroup.com/
Hyperception	Hypersignal-Windows Block Diagram	http://www.hyperception.com/
Jovian	Pegasus Parallel Processing Design Environment	http://www.jovian.com/jovian/
Mentor Graphics	DSP Station	http://www.mentorg.com/
Orincon	Rippen	http://www.ppgsoft.com/rippen.html
Signalogic	DSPower	http://www.signalogic.com/
Synopsys/CADIS GmbH.	COSSAP	http://www.synopsys.com/
U.C. Berkeley	Ptolemy	http://www.ptolemy.berkeley.edu/

Of the tools available for signal processing application development, SPW offers the most promise for rapid prototyping. Algorithm design begins using intuitive, graphical block diagrams as a system specification. Next, the algorithm's block diagram can be simulated for proper operation. Implementation can take the form of either code generation for workstations and off-the-shelf DSP processors or VHDL code for synthesis and the fabrication of a custom VLSI chip.

Software Evaluation Factors

A well designed software package is one which strikes a balance between ease of learning, ease of use, and functionality [22:13]. To evaluate a software package, each of these factors must be kept in mind while learning to use and eventually using a software package to perform tasks it was designed to perform. Ease of learning is the extent to which a new user can become proficient with the software with minimal training and practice [22:13]. What is the quality of the introductory tutorials? How long does it take to gain proficiency with the software? Ease of use is the extent to which the software allows an experienced user to perform tasks with minimal effort [22:13]. Are there shortcuts which allows an experienced user to perform tasks more quickly? Functionality is the extent of different capabilities the software provides [22:13]. Does the software provide all the necessary functions and sufficient options to tailor those functions? Is the user protected from complexity while at the same time given sufficient capabilities to keep from outgrowing the system? While evaluating software, these factors and the techniques to optimize them must be kept in mind.

In his book on computer interface design guidelines, Brown describes four techniques for optimizing the ease of learning, ease of use, and functionality of a software package [22:14]. The first technique is to design for novices, experts, and intermittent users alike. Menus should be available for the novices and intermittent users, but shortcuts such as keystroke combinations should be available for experienced users. The second technique is to avoid excess functionality. Functions should be prioritized by estimated frequency of use so that the more used are easiest to perform while seldom used functions are accessible through secondary paths or eliminated entirely. The third

technique is to provide multiple paths through the use of menu bypass, stacking or type-ahead techniques, and user-defined macros to handle the same task. The fourth technique is to design for progressive disclosure and graceful evolution by making basic functions simple to learn and frequent tasks quick to perform, encouraging experimentation by minimize consequences through reversible actions, and using defaults to minimize the user choices to produce the most likely outcome [22:15-17].

There are various other design details to consider when evaluating a software package. For example, the appropriate use of color allows a user to locate or identify classes of information with greater speed and reliability [22:66]. Icons, when used correctly, can simply task selection. Error messages should allow a user to learn what was done incorrectly and how to go about correcting the error [23:ix]. On-line documentation should not just be an electronic version of what is available from the printed manuals, but it should supplement the manuals by being content-sensitive. A well designed software package uses all of these elements to contribute to the ease of learning, ease of use, and functionality of a system.

Summary

This literature review provides background information on high-speed avionics signal processing applications. Research has shown the advantages of parallelism and the parallelization possibilities existing in the avionics application, *Xpatch.* Algorithms such as the FFT can also exploit the advantages of parallelism. The unique characteristics of the single-chip DSP microcomputer make it an ideal candidate to serve as a building block for a parallel architecture. Careful analysis of the architectures described herein is

required to determine their applicability to a particular algorithm. An overview of different DSP development environments was provided along with methods to evaluate them. Environments such as SPW allow a designer to design, simulate, and implement a DSP algorithm. After an explanation of the algorithm selected, the next chapter describes an algorithm design using SPW.

III. Algorithm Selection and Design

This chapter begins with an explanation of the algorithm selection which is representative of algorithms encountered in high-speed avionics applications. The selected algorithm serves as a basis from which to evaluate SPW as a tool to support rapid prototyping or parallel signal processing algorithms. The explanation of the algorithm selection is followed by an introduction to the selected algorithm, the 2D fast Fourier transform (FFT), beginning with the fundamentals of continuous Fourier transform and ending with the parallel multidimensional FFT. A description of the SPW 2D FFT design follows.

Algorithm Selection

To adequately evaluate whether or not SPW properly blends ease of learning, ease of use, and functionality, the software must be sufficiently exercised. That is, an algorithm must be chosen which represents a class of problem which might be used in an avionics application. Moreover, the chosen algorithm must have a level of complexity to sufficiently test the simulation and code generation capabilities of SPW. Toward these aims, the 2D FFT has been selected as the algorithm to implement with SPW.

The 2D FFT has found use in a variety of applications. Applications include tomography, data compression and picture processing [24:216]. Also, the 2D FFT is used for two-dimensional waveforms encountered in geophysical arrays, gravity and magnetic data, and antenna analysis [25:232]. Radar applications include cross-section measurement, moving target indicators, synthetic aperture, Doppler processing, pulse

compression and clutter rejection [25:2-3]. Any of the aforementioned applications can benefit from possible speed and accuracy improvements. However, in real-time applications where data post-processing is not an option, speed becomes a hard requirement.

Xpatch algorithms were not chosen to design and implement in SPW for several reasons. When this research was initiated, the offices responsible for *xpatch* maintenance were in the midst of reorganization. This reorganization hindered efforts to receive foresight into the future of *xpatch*. The restricted nature of the *xpatch* code itself also limited accessibility. Interest expressed by both the sponsor, Wright Laboratory, and Rome Laboratories as well as the universal applicability of the 2D FFT to applications other than radar led to the ultimate selection.

Introduction to the Fast Fourier Transform

Fourier Transform

Essentially, the Fourier transform of a waveform is the decomposition of that waveform into sinusoids of varying frequencies which must sum to the original waveform [25:4]. Mathematically stated, the Fourier integral is defined as

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{-j2\pi ft} dt \tag{2}$$

where *h(t)* is the time domain waveform decomposed into a sum of sinusoids of varying frequencies, and *H(f)*, the frequency domain, is the Fourier transform of *h(t)* if the integral exists for every value of *f* [25:9]. Traditionally, lowercase letters represent functions of

time while uppercase letter represent functions of frequency. The Fourier transform is a continuous function. For computer calculations the Fourier transform must be discretized resulting in the discrete Fourier transform (DFT).

Discrete Fourier Transform

The DFT approximates the continuous Fourier transform by representing the waveform to be decomposed as a set of regularly sampled points rather than a continuous waveform. Thus, the DFT is a linear transformation which maps a set of regularly sampled points from a cycle of a periodic signal onto an equal number of points to represent a signal's frequency spectrum [7:377]. The DFT approximation can unambiguously determine frequencies within a range as defined by the Shannon's sampling theorem [26]. This is also known as the Nyquist rate where the frequencies contained in the signal are all less than half the rate at which the points are sampled [27:9]. However, when accounting for real-world restrictions such as finite word lengths and associated quantization error, rates as high as 10-to-1 are suggested. Mathematically, the DFT is stated as [27:10]

$$A(k) = \sum_{n=0}^{N-1} a(n) * e^{-j2\pi nk/N} \tag{3}$$

where

$$e^{-j2\pi nk/N} = \cos(2\pi nk/N) - j\sin(2\pi nk/N)$$

N: number of complex data points

n: input point index

k: output point index

a(n): discretized input signal

While this discretized approximation of the Fourier transform can be handled by digital computer, the number of operations can be reduced as found by Cooley and Tukey in 1965 to form the FFT.

Fast Fourier Transform

The algorithm Cooley and Tukey devised in 1965 and its subsequent variations which compute the DFT of an n-point series in O(nlogn) operations are collectively referred to as the FFT [7:377]. The FFT allows Fourier analysis of signals through the use of digital hardware and computers instead of analog filter banks and spectrum analyzers [28:60]. The FFT is essentially a recursive algorithm for computing the DFT [29:231]. An excellent explanation of the development of the FFT is provided in [25:132]. Here, an example shows how the matrix factorization process introduces zeros into the factored matrix thus eliminating the need for some multiplication operations. Essentially, the efficiency of the FFT is based on the capitalizing on the symmetry and periodicity attributes of the complex phase portion of the DFT calculations [24:217].

The FFT offers an improvement over the DFT with reduced computational load. The computational load is reduced from $4N^2$ additions and $4N^2$ multiplications for the DFT to 2N additions and $N\log_2(N)$ multiplications for the FFT [27:28]. Reduced quantization noise is an indirect benefit from the fewer number of calculations required. Quantization noise is reduced since fewer multiplications are performed and therefore there are fewer times where the multiplication result must be rounded off [27:28].

The FFT has two disadvantages. The reorganization of data and the computation reduction necessitates computation of all of the output frequencies even if only a few are required [27:28]. While the DFT outputs one output frequency at a time, often all frequencies are needed and the computational savings compensate for this weakness [27:28]. Another disadvantage to FFT algorithms are their inability to handle inputs varying numbers of points. That is, the number of input points is fixed. To overcome this, zero padding is can be used for signal with fewer samples than the FFT algorithm [27:14]. While adding zeros to a signal allows for variable data collection lengths for a given FFT algorithm, the real and imaginary responses are affected [27:15]. The affects of this can be minimized through the use of weighting (or window) functions [27:35].

Multidimensional Fast Fourier Transform

The multidimensional FFT extends the single dimensional FFT to two or more dimensions. A 2D signal is a function $h(x,y)$ of two variables x and y to describe 2D waveforms such as images [25:232]. Video, for example, offers a third dimension of time. Assuming the 2D signal is periodic in all dimensions, in the continuous space, the 2D Fourier transform is given by

$$H(u,v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} h(x,y) e^{-j2\pi(ux+vy)} dxdy \tag{4}$$

where $h(x,y)$ is the 2D function and $H(u,v)$ is the 2D transform of $h(x,y)$ [25:232-233]. Similar to the 1D transform, the 2D (N_1 x N_2) discrete Fourier transform is given by [27:74]

$$A(k_1,k_2)=\sum_{n_1=0}^{N_1-1}\sum_{n_2-1}^{N_2-1}a(n_1,n_2)e^{-j2\pi[n_1k_1/N_1+n_2k_2/N_2]} \tag{5}$$

where

$$e^{-j2\pi[n_1k_1/N_1+n_2k_2/N_2]}=\cos(2\pi(n_1k_1/N_1+n_2k_2/N_2))-j\sin(2\pi(n_1k_1/N_1+n_2k_2/N_2))$$

N_1 x N_2: size of discretized input signal

n_1, n_2: input point indices

k_1, k_2: output point indices

and $a(n_1,n_2)$ and $A(k_1,k_2)$ are the discretized input and output signals respectively. This separability of the summations in terms of n_1 and n_2 leads to the row-column decomposition of the 2D Fourier transform.

Multidimensional transform implementations can be hampered by the volume of sampling data required. To reduce the amount of sampling points required, different sampling geometries may be employed. The straightforward, uniform rectangular sampling pattern is most often used for simplicity of use and implementation. This consists of periodic sampling in rectangular coordinates as illustrated in Figure 12 [28:36].

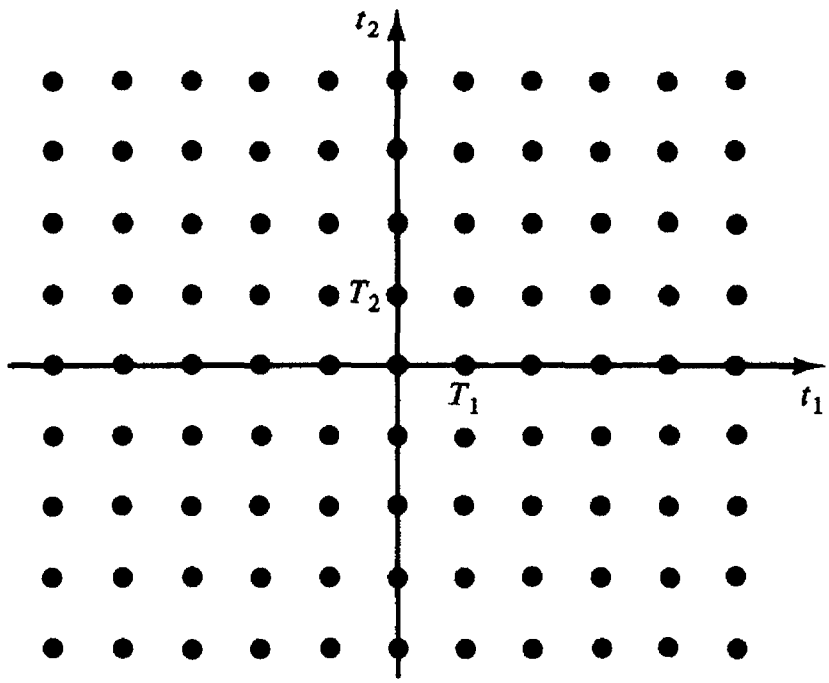

Figure 12 - Rectangular sampling grid [28:36]

This is also known as the sampling matrix or lattice [28:266]. It can be shown that for higher and higher dimensions of transforms, the rectangular sampling scheme does not provide the most efficient sampling method in terms of the number of samples needed to represent a multidimensional signal. Figure 13 shows a hexagonal sampling grid:

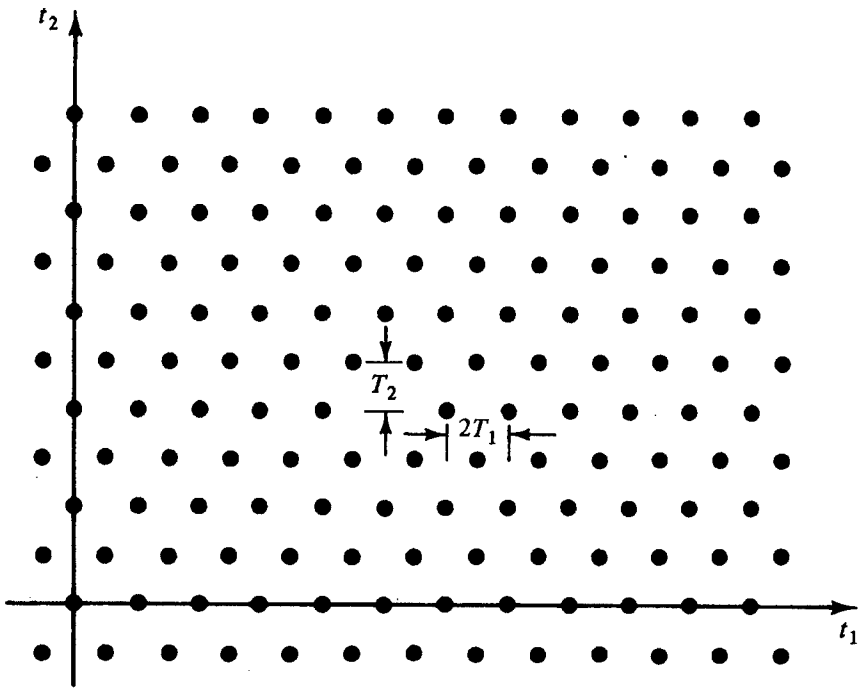

Figure 13 - Hexagonal sampling grid [28:44]

The efficiency gains of a hexagonal sampling scheme (hyperspherical) over a rectangular sampling scheme (cubic) is quite substantial with increasing dimensions of transforms as shown in Table 2.

Table 2 - Ratio of efficiency of an M-dimensional cubic lattice to a hyperspherical lattice [28:47]

M	Efficiency
1	1.000
2	0.866
3	0.705
4	0.499
5	0.353
6	0.217
7	0.125
8	0.062

Multidimensional DFTs are mathematically represented using matrices in order to represent the periodicities due to both the sampling lattice and the signal to Fourier transform [30:45]. Refer to Appendix B for more information.

Row-Column Decomposition

The most natural method of calculating the 2D Fourier transform is the row-column decomposition method due to the structure. This method uses the separability of the 2D Fourier transform to decompose the problem into two sets of 1D transforms. By factoring the exponential term

$$e^{-j2\pi[n_1k_1/N_1+n_2k_2/N_2]} = e^{-j2\pi(n_1k_1/N_1)} * e^{-j2\pi(n_2k_2/N_2)} \tag{6}$$

the two summations may be separated to give

$$\sum_{n_1=0}^{N_1-1}\left[\sum_{n_2=0}^{N_2-1} a(n_1,n_2) * e^{-j2\pi(n_2k_2/N_2)}\right] * e^{-j2\pi(n_1k_1/N_1)} \tag{7}$$

This results in taking a 1D Fourier transform in the n_2 dimension followed by a 1D Fourier transform in the n_1 dimension or a row-column decomposition. Of course, this can be accomplished in either order. An excellent graphical development of the 2D Fourier transform is given in Brigham [25:241]. The following figure illustrates the row-column decomposition FFT:

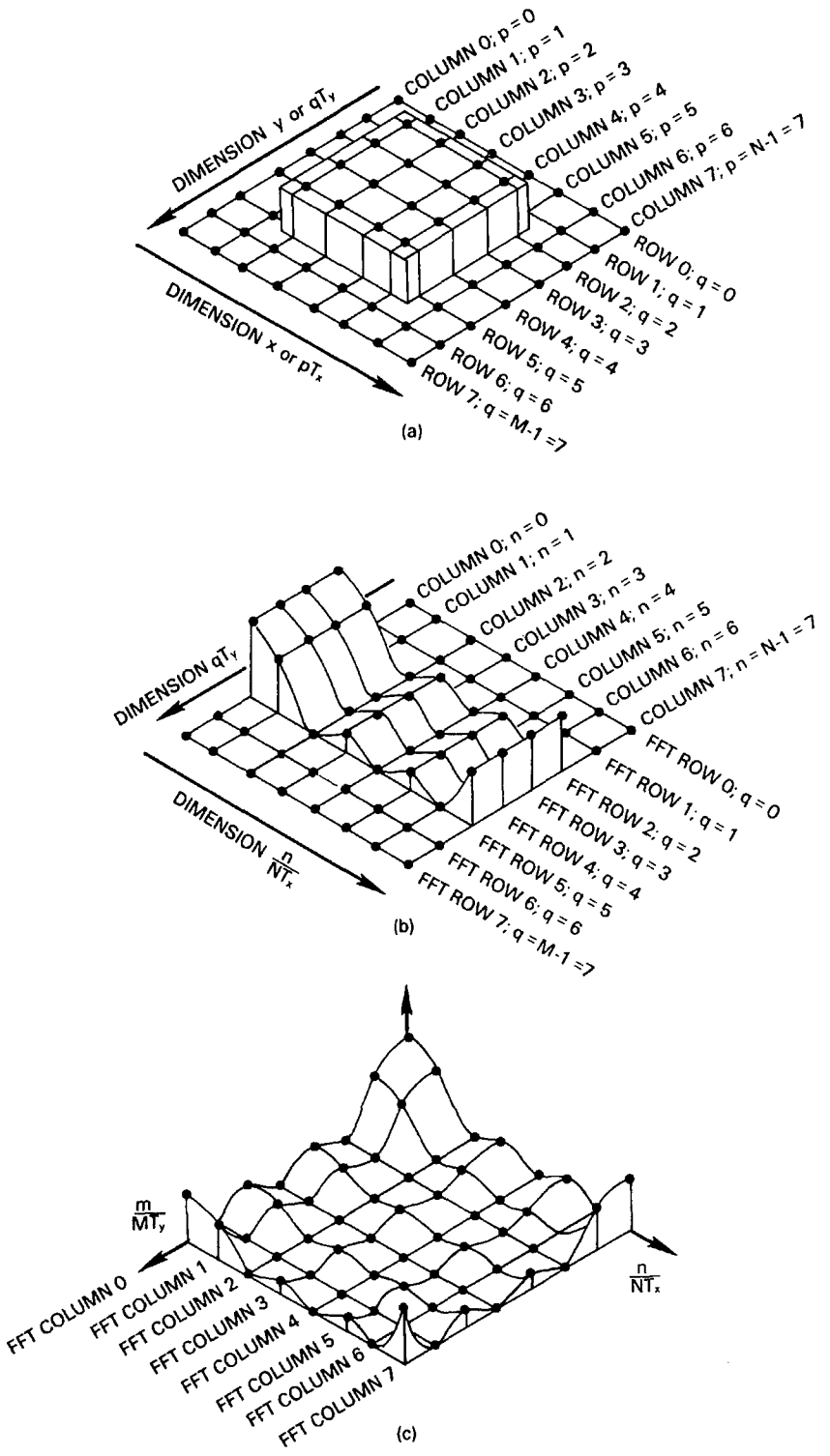

Figure 14 - Graphical development of the 2D FFT with row-column decomposition [25:242]

Figure 14 (c) gives the unordered 2D FFT result. Conventional viewing would require rearranging the FFT results. If the FFT results were divided into quadrants, rearranging would be accomplished by performing a right circular shift through two quadrants [25:244] as illustrated in Figure 15.

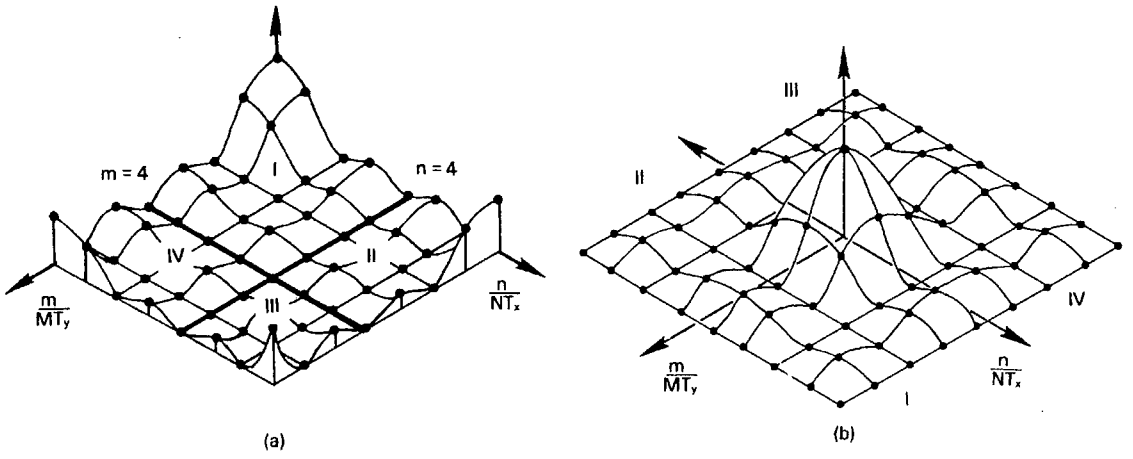

Figure 15 - 2D FFT reorganization for conventional viewing [25:245]

The row-column decomposition has an advantage in its simplicity. The decomposition is intuitive and easy to visualize. Also, a 2D transform algorithm can be constructed out of 1D transform algorithms that may already be provided. 1D algorithms are readily available and highly optimized for different computational machines. For example, the CLASSPACK Signal Processing Library contains optimized C routines to perform FFTs [31]. Thus, construction of a 2D algorithm can benefit from their computational efficiency.

Vector-Radix Algorithms

A 1D FFT achieves computational efficiency through the "divide and conquer" strategy where each transform length is recursively divided by a power of 2 into smaller transform lengths [28:76]. Like the 1D FFT, the 2D vector-radix FFT decomposes a 2D DFT into successively smaller 2D DFTs until only trivial 2D DFTs remain [28:76]. This basic structure of the algorithm is commonly called a butterfly. The decimation-in-time version of the vector-radix algorithm is accomplished by expressing a (N_1 x N_2)-point DFT in terms of four $N_1/2$ x $N_2/2$ DFTs represented by four summations [28:77]:

$$S_{00}(k_1,k_2) = \sum_{m_1=0}^{N/2-1}\sum_{m_2=0}^{N/2-1} x(2m_1,2m_2)W_N^{2m_1k_1+2m_2k_2} \tag{8}$$

$$S_{01}(k_1,k_2) = \sum_{m_1=0}^{N/2-1}\sum_{m_2=0}^{N/2-1} x(2m_1,2m_2+1)W_N^{2m_1k_1+2m_2k_2} \tag{9}$$

$$S_{10}(k_1,k_2) = \sum_{m_1=0}^{N/2-1}\sum_{m_2=0}^{N/2-1} x(2m_1+1,2m_2)W_N^{2m_1k_1+2m_2k_2} \tag{10}$$

$$S_{11}(k_1,k_2) = \sum_{m_1=0}^{N/2-1}\sum_{m_2=0}^{N/2-1} x(2m_1+1,2m_2+1)W_N^{2m_1k_1+2m_2k_2} \tag{11}$$

One summation handles data with both even indices, a second handles data with even and odd indices, a third handles data with odd and even indices, and a fourth handles data with two odd indices [28:77]. Figure 16 illustrates a single radix-(2 x 2) butterfly:

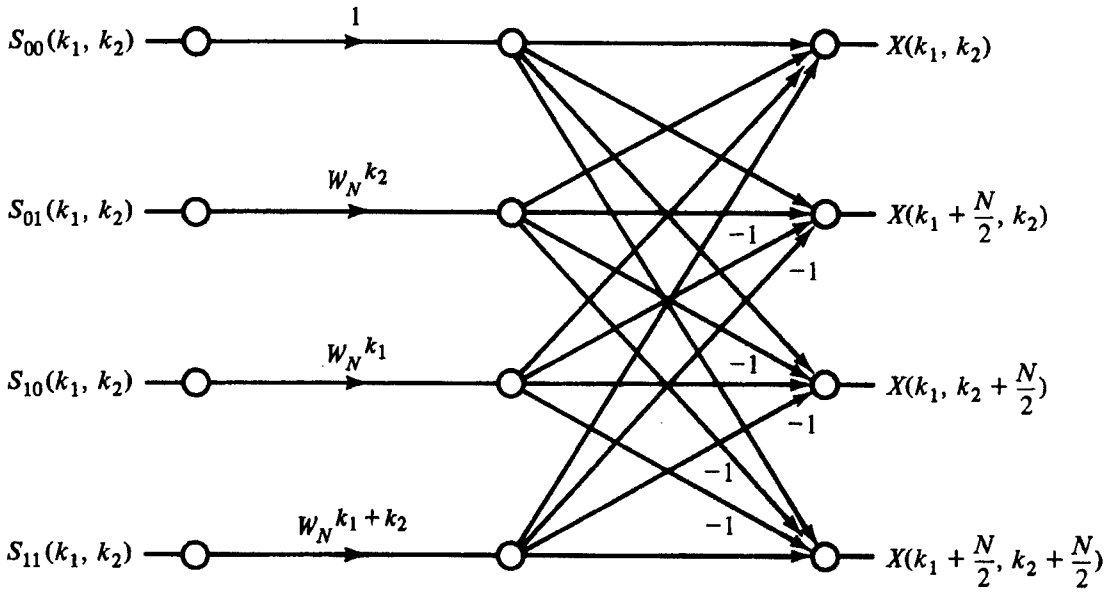

Figure 16 - Isolated Radix-(2 x 2) Butterfly [28:78]

Figure 17 illustrates how a larger FFT is constructed from smaller FFTs:

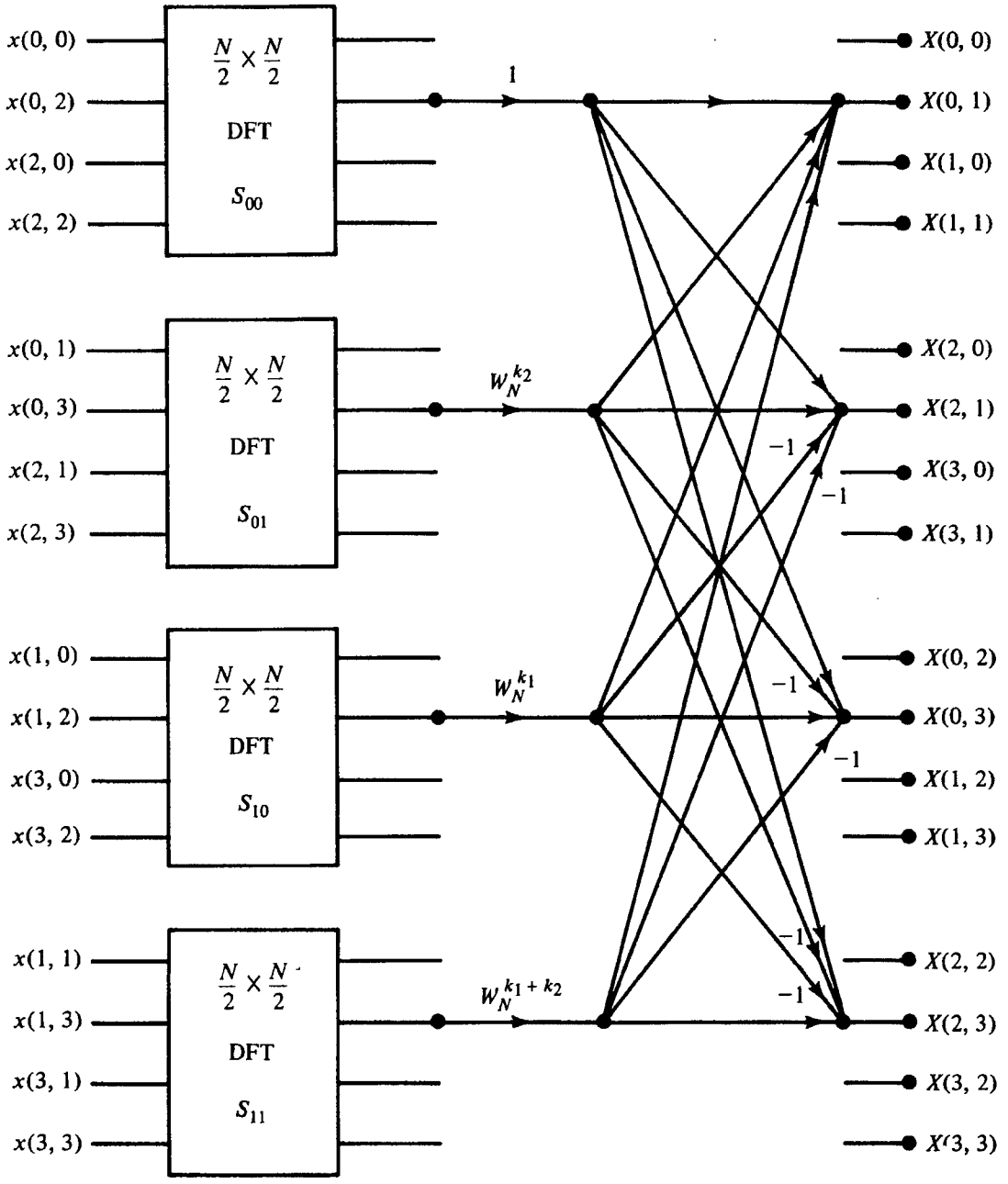

Figure 17 - Radix-(4 x 4) FFT built upon radix-(2 x 2) FFTs (only one of the four butterflies is shown in the second column) [28:79]

Refer to Dudgeon [28] for a complete derivation of the vector-radix FFT.

Computational savings are the primary advantage of the vector-radix algorithm over the row-column decomposition. The number of complex multiplications for the row-column decomposition is

$$C_{r/cFFT} = \frac{M}{2} N^M \log_2 N \tag{12}$$

where

M = number of dimensions

N = number of complex points

and the number of complex multiplications for the vector-radix algorithms is given by

$$C_{vr(2x2)} = \frac{2^M - 1}{2^M} N^M \log_2 N \tag{13}$$

Table 3 shows the computational savings in terms of complex multiplications required for vector-radix (C_{vr}) and row-column ($C_{r/c}$) multidimensional FFTs:

Table 3 - Comparison of number of complex multiplications required for M-dimensional FFT algorithms [28:82]

M	$C_{vr(2x2)}/C_{r/c\ FFT}$
2	0.75
3	0.58
4	0.47
5	0.39

The vector-radix algorithm requires fewer and fewer complex multiplications as compared to the row-column algorithm as the number of dimensions increases. The

number of complex additions required is the same for both methods and is given as [28:82]

$$A_{r/cFFT} = A_{vr(2x2)} = MN^2 \log_2 N \tag{14}$$

Though each approach requires the same number of complex additions, the vector radix does offer savings in the number of complex multiplications required.

Parallel Two-Dimensional Fast Fourier Transform

Parallel Row-Column Decomposition Fast Fourier Transform

A 2D FFT with row-column decomposition can be accomplished in parallel by either performing parallel 1D FFTs serially, performing serial FFTs in parallel, or by a combination thereof. Choosing among the approaches depends largely on the resources available, for example, the number of processors and the availability of existing code. Two algorithms for performing parallel 1D FFTs are described in Kumar [7]. They are the binary-exchange algorithm and the transpose algorithm.

Binary-Exchange Algorithm

The binary-exchange algorithm is described in Kumar for a hypercube for one or multiple elements per processor and a mesh architecture. The FFT structure lends itself to the hypercube architecture since the required communications use the added connectivity of the hypercube topology efficiently. For the one element per processor approach, the FFT is cost-optimal with a processor-time product of $\Theta(n \log n)$, equal to the complexity of a serial n-point FFT [7:383]. Since, this approach is usually not

feasible for a large number of points, the multiple elements per processor approach must be taken as shown in Figure 18.

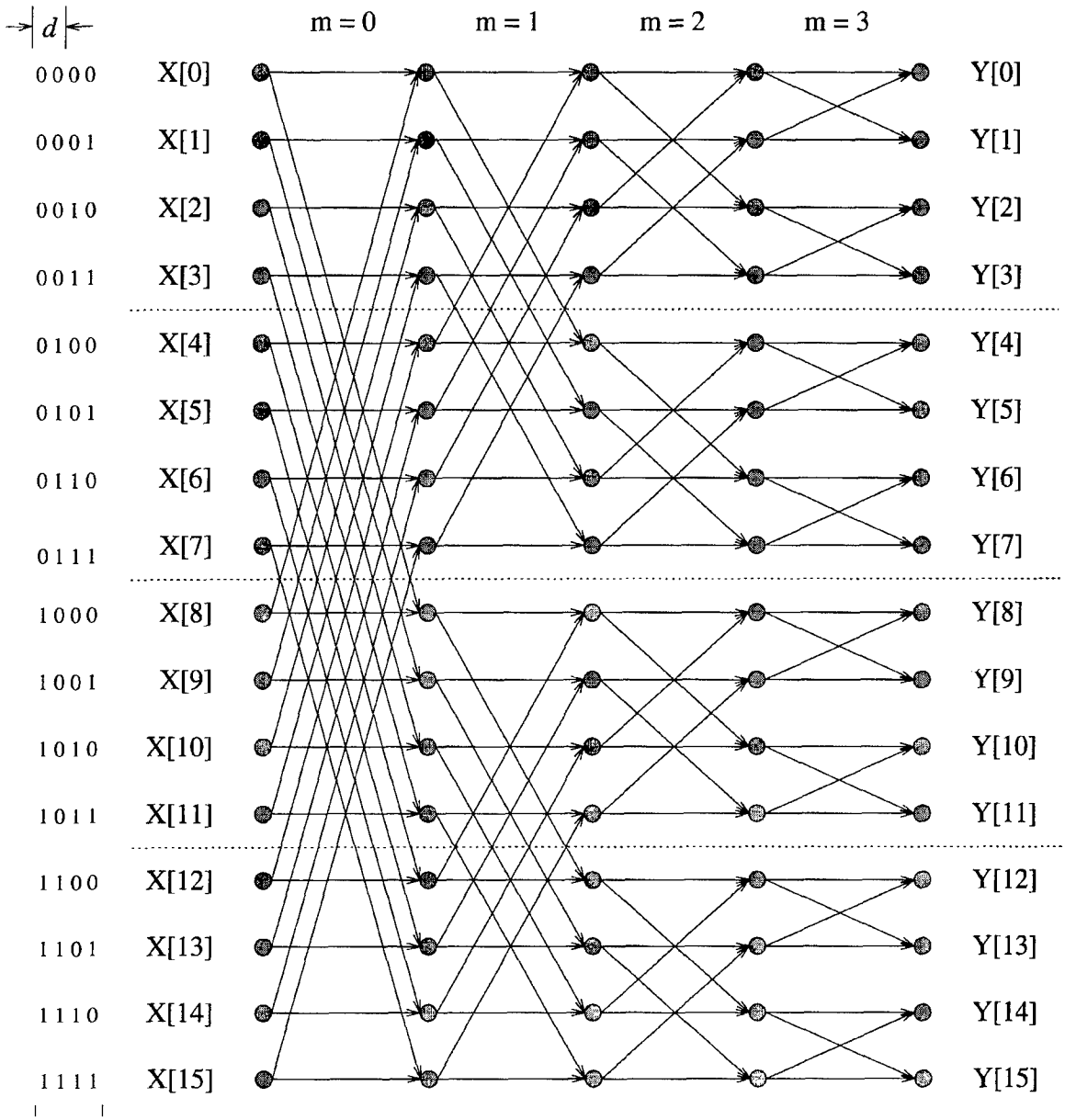

Figure 18 - 16-point FFT on four processors where every four rows is on a separate processor [7:384]

Kumar shows that for this approach the binary-exchange algorithm is reasonably scaleable if the problem size is increased at the rate of $\Theta(p \log p)$ and the communication bandwidth and processing speed of the processors are balanced [7:388]. On the other hand, the binary-exchange algorithm is not very scaleable on a mesh since the problem size must be increased exponentially with the number of processors to maintain constant efficiency [7:390].

Transpose Algorithm

The transpose algorithm, which uses matrix transposition, is useful when the ratio of communication bandwidth to processor speed is low and high efficiencies are required [7:393]. In the transpose algorithm, a $\sqrt{n}$ -point FFT is computed for each column of $\sqrt{n}$ x $\sqrt{n}$ array of points. After the array is transposed (the only communication required), $\sqrt{n}$ -point FFTs are computed for each of the columns for the transposed array [7:394]. Figure 19 shows the how the elements are combined to compute a 16 point 2D FFT using 4 x 4 square array.

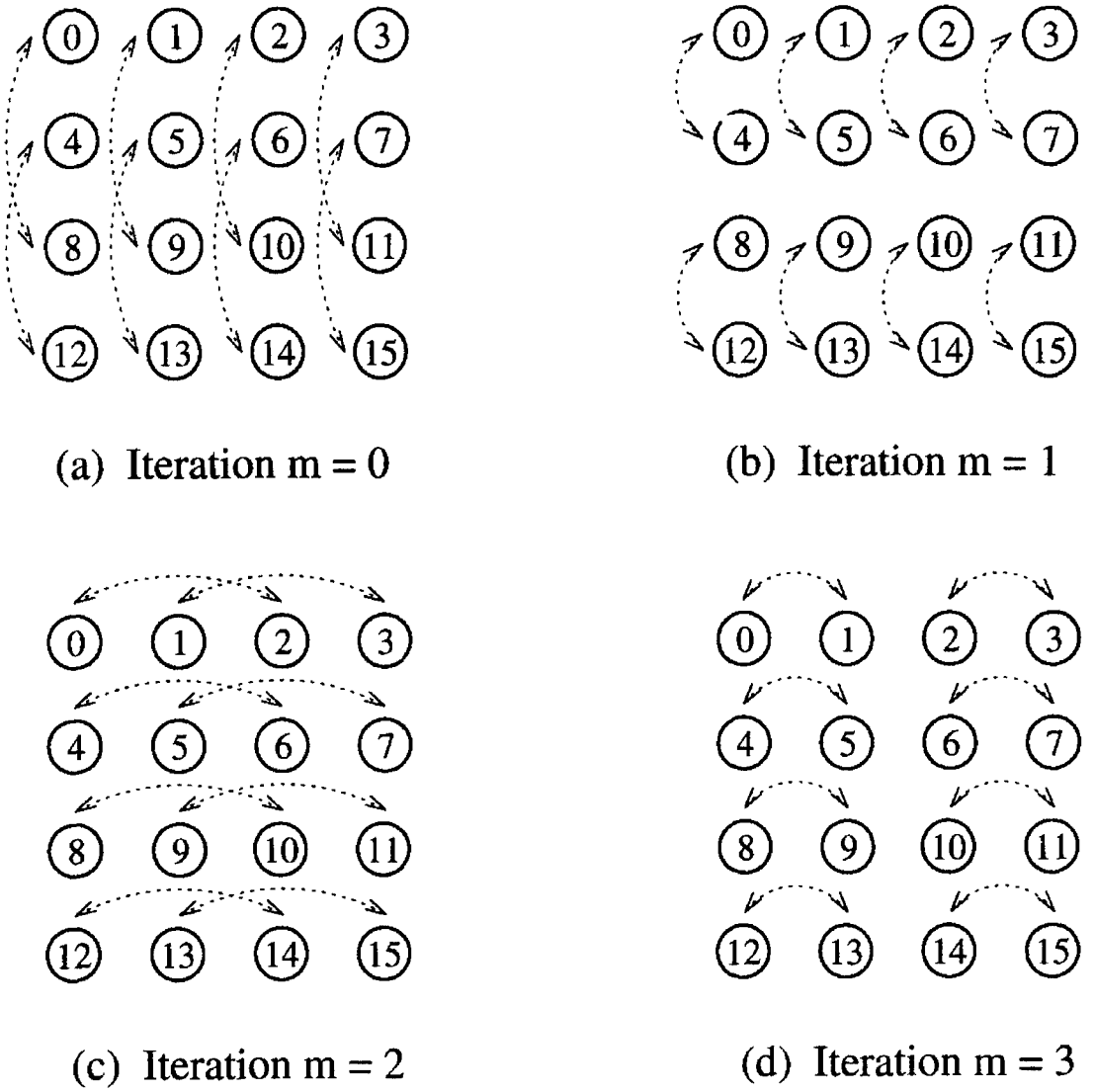

Figure 19 - Combination of elements in a (4x4)-point 2D FFT [7:394]

The choice of whether to use the binary-exchange or the transpose algorithm depends on the relative values of the communication time parameters with SIMD and shared-memory computers leaning toward the transpose algorithm and MIMD computers with the binary-exchange algorithm [7:396].

Parallel Vector-Radix Fast Fourier Transform

Since the vector-radix approach to the FFT is similar to the 1D FFT in that both are decomposed into smaller FFTs, a similar decomposition among multiple processors is possible. Two approaches are possible. The first approach involves pipelining where one or more of the columns of the signal flow diagram are handled by each processor. This approach would be more suitable for performing a series of FFT computations for higher throughput at the expense of latency. For example, if a single processor were assigned to handle the computations of each stage, $\log_2 N$ processors would be required for a N x N 2D FFT. This results in $\Theta(N^2)$ computations assuming the same execution times for both complex additions and multiplications. On the other hand, partitioning the graph horizontally would reduce latency times subject to the increased communication required between processors. If the number of processors, p, were equal to N for an N x N FFT, the computations would be $\Theta(N\log_2 N)$. The overhead required for communication would be $\Theta(N\log_2 N)$ and $\Theta(N^2)$ for the vertical partitioning ($p=N\log_2 N$) and horizontal partitioning ($p=N$) respectively. The pipelining method can benefit from fewer messages with less processors since messages may be combined between stages for better efficiency whereas horizontal partitioning requires communication between all processors at each stage.

Tensor Product Programming Language

Another approach to designing FFT algorithms is through the use of tensor products. The tensor product notation is a concise method for characterizing complex signal processing algorithms with mechanisms for specifying operation for serial, vector,

or multiprocessor computers [32:41]. The tensor product is a binary matrix operator used to combine two matrices to form a single, larger matrix, and tensor product factorizations can reveal underlying symmetries which may be used to design efficient algorithms [32:41]. The reader is referred to [32] for an explanation of the tensor product approach and the applicability to FFT algorithms.

Two-Dimensional Fast Fourier Transform Design in SPW

Algorithm Selection

The vector-radix 2D FFT was chosen as the algorithm to implement in SPW for its applicability to avionics and more specifically, radar applications. While a row-column approach would have sufficed, the vector-radix algorithm is well suited in a hierarchical design environment whereby FFTs capable of a larger number of input points are built using FFTs of fewer points. Thus, larger and larger systems may be constructed while increasing system complexity and exercising the software's capability to handle this complexity. Also, the regular structure of the vector-radix 2D FFT is well suited for partitioning for multiple processors.

One assumption must be stated before block diagram design. First, input sampling points are assumed to be equally spaced in each dimension or rectangular. Algorithms capable of handling different geometries may be designed, but there is no specific need for that capability in this research. Furthermore, in radar applications of the 2D FFT, source data is collected in traditional rectangular form. While other geometries can use more efficient sampling geometries, such an approach would require more effort on the sampling side of the problem.

Block Diagram Design Example

The block diagram design process begins with the Block Diagram Editor (BDE) of SPW. In the BDE, a signal flow diagram of a particular algorithm is constructed by connecting functional blocks together. SPW provides a wide variety of functional blocks (approximately 350) for this purpose, and SPW has the facility to allow a designer to create custom blocks by linking in FORTRAN or C programs to a new block. At the block diagram design level, features such as self-test and redundancy may be designed in to address reliability. Reproducing sections of a block diagram design and adding decision-making logic is possible within the BDE.

For the purposes of the vector radix 2D FFT, all the required blocks are provided. Levels of hierarchy are used to hide detail at higher levels of abstraction. Every block, whether it be SPW-provided or user-designed, contains a detail model and a symbol. The following explanation of the block diagram design process centers on the design of the basic vector radix-(2x2) butterfly.

Butterfly Detail Model

The detail model is the level of design which determines how a particular block functions. It consists of other blocks and the associated interconnections. The vector radix butterfly consists solely of complex additions as shown by the following equations [28:73]:

$$X(0,0) = x(0,0) + x(1,0) + x(0,1) + x(1,1) \tag{15}$$

$$X(0,0) = x(0,0) - x(1,0) + x(0,1) - x(1,1) \tag{16}$$

$$X(0,0) = x(0,0) + x(1,0) - x(0,1) - x(1,1) \tag{17}$$

$$X(0,0) = x(0,0) - x(1,0) - x(0,1) + x(1,1) \tag{18}$$

At first glance, it appears the calculation of the four points requires twelve additions/subtractions. However, by calculating intermediate values, the number of additions/subtractions may be reduced to eight:

$$a = x(0,0) + x(1,0) \tag{19}$$

$$b = x(0,0) - x(1,0) \tag{20}$$

$$c = x(0,1) + x(1,1) \tag{21}$$

$$d = x(0,1) - x(1,1) \tag{22}$$

$$X(0,0) = a + c \tag{23}$$

$$X(1,0) = b + d \tag{24}$$

$$X(0,1) = a - c \tag{25}$$

$$X(1,1) = b - d \tag{26}$$

All that remains is to choose the necessary blocks to represent the required calculations. Assuming the general case of complex input points, the complex addition and subtraction blocks can be used to represent the respective additions and subtractions. The following block diagram in Figure 20 shows the detail model for the butterfly with the appropriate interconnections.

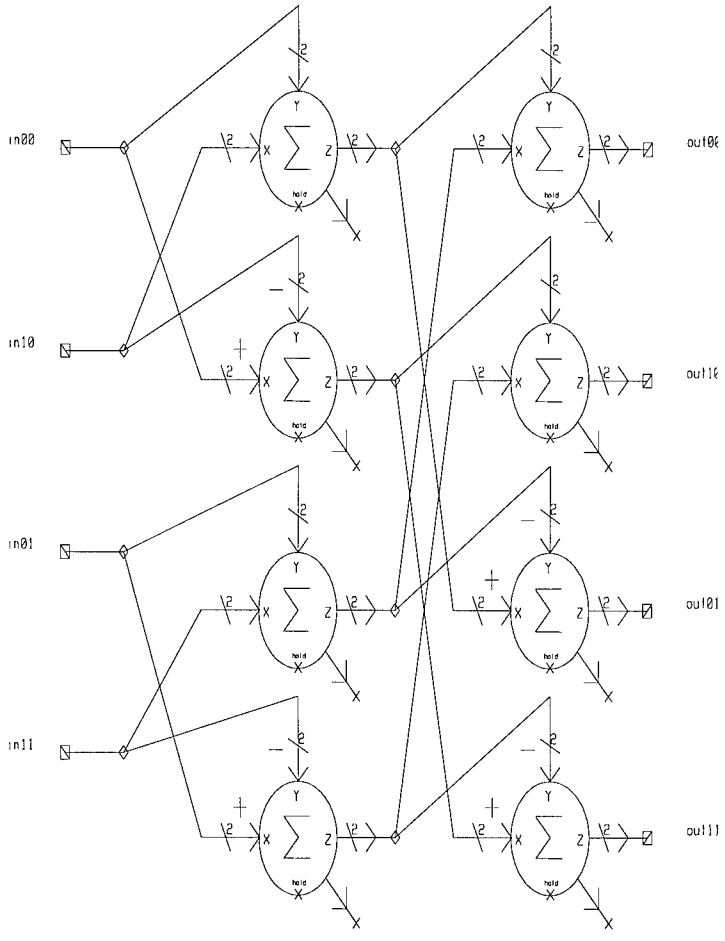

Figure 20 - Block diagram detail model of the (2x2)-point butterfly

The in00 and out00 ports of the diagram represent the complex input point $x(0,0)$ and output point $X(0,0)$ respectively. Note the '/2' on the signal lines which indicate complex numbers. The hold inputs of the blocks may be used for synchronization purposes.

Butterfly Symbol Design

The BDE does have the capability to automatically generate a symbol for a detail model based on the model's inputs and outputs to the external world. Figure 21 shows what BDE created automatically for the butterfly detail model:

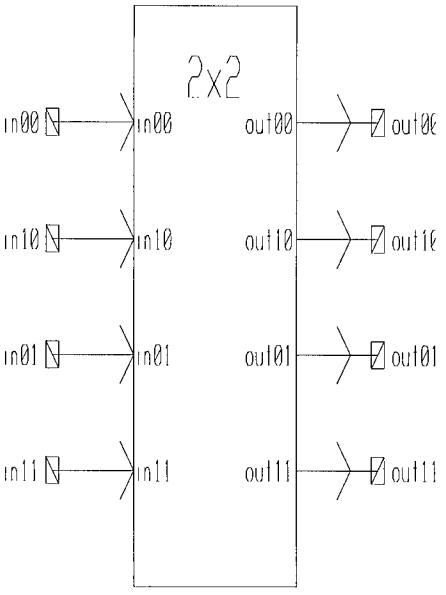

Figure 21 - BDE created symbol for the (2x2)-point butterfly

While suitable for most designs to hide detail and improve readability, the basic drawing tools of the BDE allows a designer to create a symbol which may be more representative of the block's function. With the basic drawing tools (box, circle, line, and text tools) the following familiar butterfly structure was created to make the block readily identifiable.

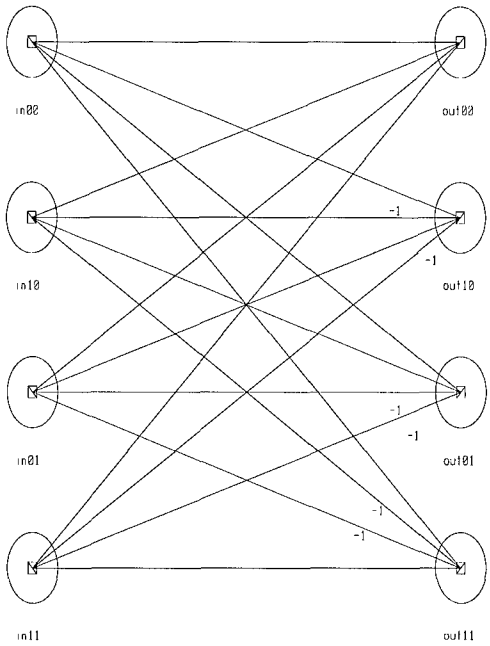

Figure 22 - Custom symbol for the (2x2)-point butterfly

SPW allows the designer to design multiple symbols to represent the same underlying detail model. This is convenient for the design of the vector radix algorithm because it allows the creation of multiple butterfly symbols of different sizes for different stages of the design. For an example of this see Figure 17 where both columns contain identical butterflies in function, but use different symbols for clarity.

Butterfly Block Diagram Testing

It is prudent in any design project to test the components at each level of design. The combination of the BDE, the Signal Calculator, and the Signal Flow Simulator allow such testing. Testing of the butterfly block diagram requires input points and storage/observation of the resulting output points. For this, a system consisting of the unit under test (UUT), signal source, and signal sink are required. SPW also has virtual

instruments in what is called the Interactive Simulation Library (ISL) to monitor signals during the simulation and provide interactive input.

Experimental Design

The UUT is simply added to a system model by adding the part through the use of the associated symbol. This way the detail of the model in hidden and only the inputs and outputs are accessible. It may be necessary to test at a lower level of detail so that internal signal are accessible. A signal source is used to provide the signal input. This block is the interface from a signal file stored on disk to the UUT. A signal sink then collects the UUT's outputs and stores them to an output signal file for viewing. Both the input and output signals may be viewed and modified in the Signal Calculator. Figure 23 shows the test configuration for the butterfly block diagram.

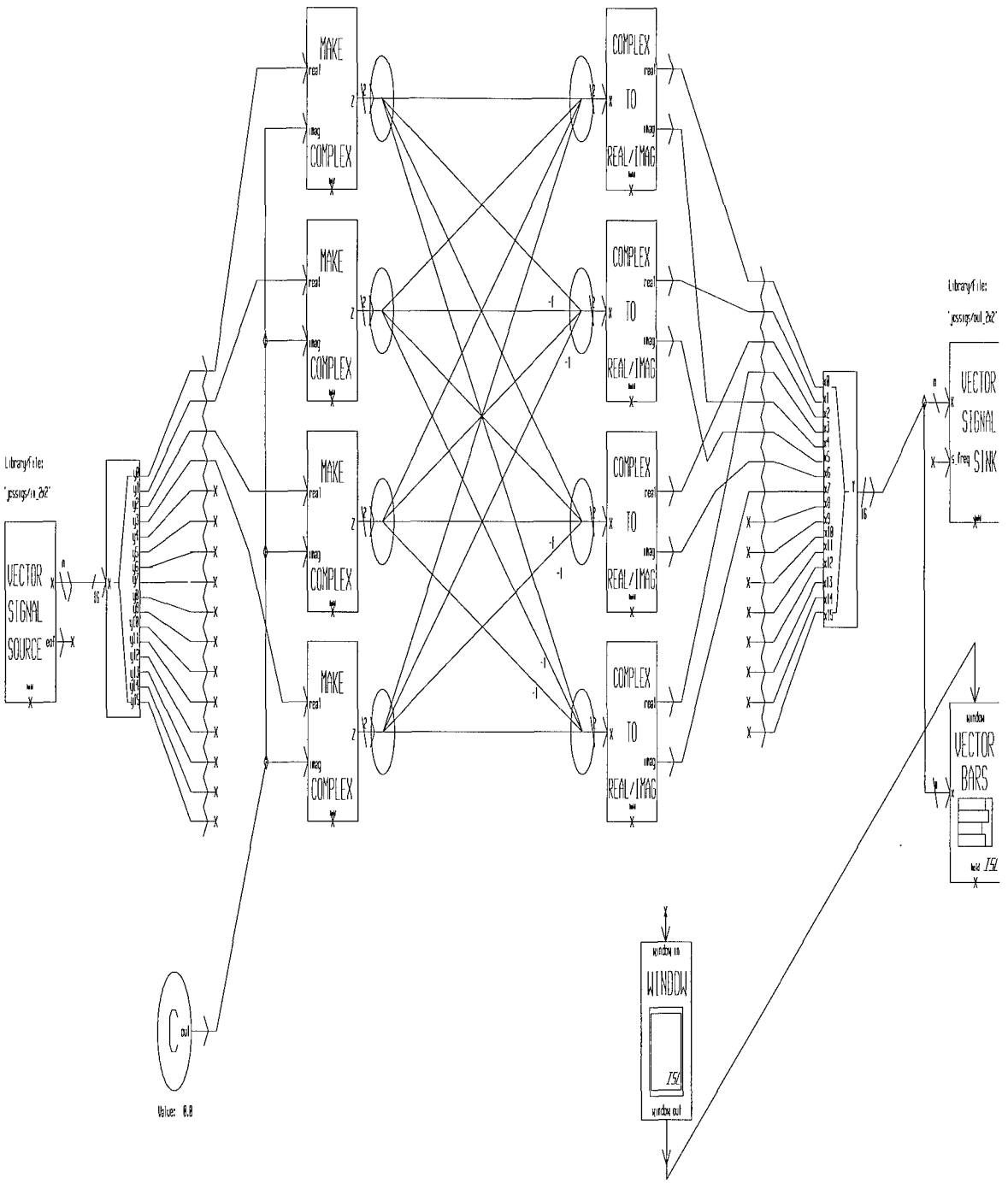

Figure 23 - Test system for the (2x2)-point butterfly

The test system makes use of vector input and output signals which are divided and recombined when necessary. Two ISL blocks are shown as well. The window block

displays the results of the vector bar block. The vector bar block displays a bar chart of an input vector to examine signals during the simulation. While useful for a cursory examination of data, no numbers are shown on the chart so only relative values are given.

Butterfly Testing

For testing the butterfly, a two-dimensional input and output format is desired. However, the Signal Calculator and the tools available in the ISL are designed to handle one-dimensional signals. This requires that the input signal and the output results be stored in a one-dimensional format. While this is of no consequence to the design, this does hinder testing by not allowing signal viewing in the more natural two-dimensional format. For one-dimensional signals, the tools of ISL are quite powerful ranging from virtual spectrum analyzers and bar graphs for display to buttons and sliding bars for interactive input.

One way of overcoming the data display limitations of SPW would be to use the MATLAB interface blocks. SPW contains MATLAB source and sink blocks to read in MATLAB-formatted input data and save MATLAB-formatted output data. While matrices are the standard data format in MATLAB, SPW is limited to reading one-dimensional data formats, and therefore, SPW reads matrices in column-major fashion. For this reason, while MATLAB could be convenient for creating and viewing input and output signals, care still must be taken in the manipulation of MATLAB matrix files in SPW designs.

The difficulty in handling the input and output data in the simulations begs the question, how should I/O be handled in a hardware implementation? The answer depends

largely on the format and means of data collection for the input and the memory capacity of the hardware. While the instantaneous availability of all input data is ideal, this is not always feasible given throughput and memory limitations. For example, a (1024 x 1024)-point 2D FFT requires 10E6 complex data points each of which may be represented by 32 or more bits depending on the chosen representation. This gives a 4 megabyte storage requirement for the data alone. Given the large number of data points, some means of decimation would be required. The throughput and memory requirements not only drive the design, but also the hardware implementation choices.

Adding Levels of Hierarchy

Adding levels of hierarchy and complexity simply requires the addition of multilevel blocks in other block detail designs. For example, a (4 x 4)-point radix-(2 x 2) FFT is constructed using eight of the basic butterflies along with intermediate complex multiplications. Figure 24 shows the detail model for this design. Note that the intermediate complex multiplications use custom symbols to represent the complex multiplication blocks. Also, the two different symbols for the butterfly detail design are shown.

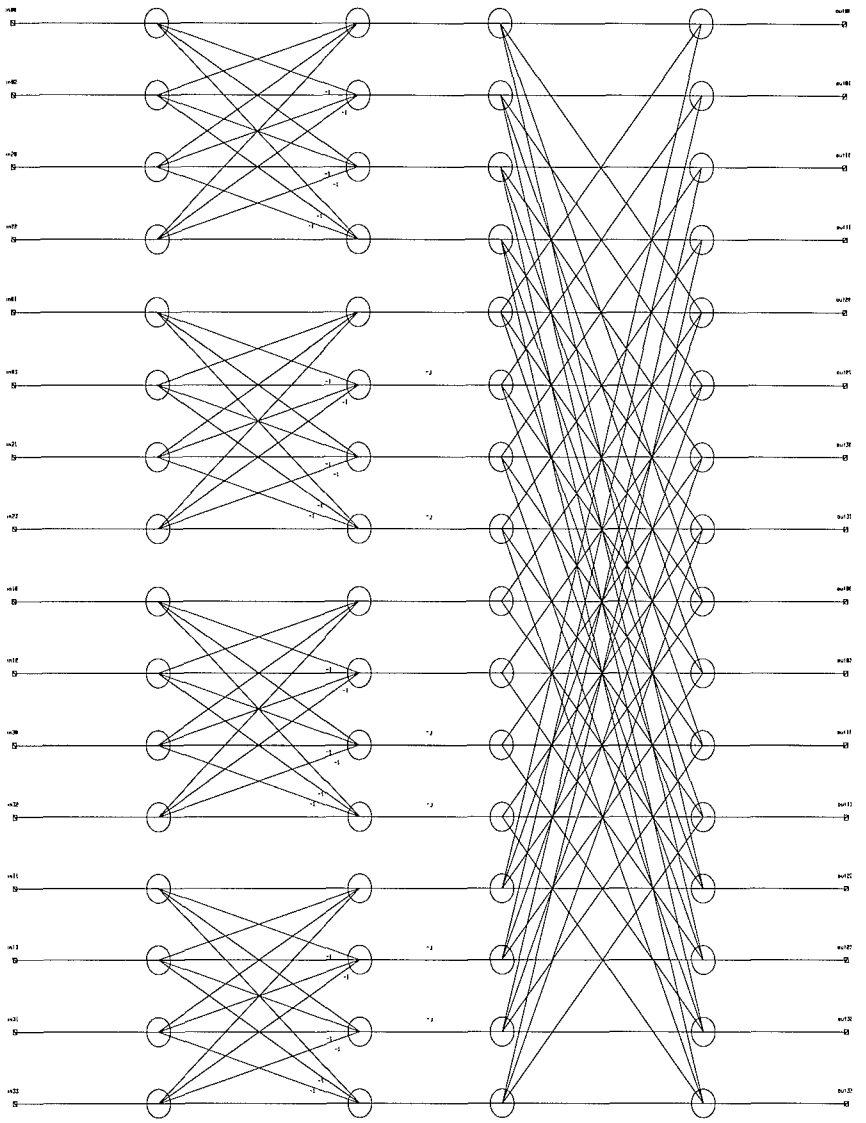

Figure 24 - Block diagram detail model for the (4x4)-point FFT

This detail design was tested in a similar manner to the basic butterfly design. The inability to easily handle the 2D data has a greater impact with the larger number of points. Using four (4 x 4)-point FFT blocks, a (16 x 16)-point FFT may be created. Four (16 x 16)-point FFT blocks may be used to construct a (64 x 64) point FFT and so on until the desired size is reached. Larger and larger transform construction can become

quite tedious as the number of interconnections increases. Also, testing quickly becomes difficult using a 1D representation for 2D signals.

Parallel Partitioning in SPW

A designer may parallelize an SPW design through the use of the MultiProx option. Any design's block diagram may be partitioned by the designer for parallel operation, but evaluation of the design using parallel processing metrics is difficult due to nearly nonexistent parallel processing analysis support within SPW. The capacity to analyze load balancing is parallel designs is discussed in Chapter IV.

SPW MultiProx Option

The MultiProx option of SPW allows a designer to partition a block diagram for execution on one or more processors. After a block diagram has been successfully simulated for a single processor, three steps are required to map the design to two or more processors. First, the existing block diagram must be partitioned to identify which processors are responsible for which functional blocks. Partitioning a block diagram is accomplished from within the BDE. The block diagram is partitioned into subsections known as regions. Second, MultiProx must generate the code for each processor and generate the necessary interprocessor communication. MultiProx does so by automatically inserting inter-processor communication (IPC) blocks between partitions. Third, the partitioned, parallel design is simulated to determine correct operation and evaluate the partitioning scheme employed. During simulation, real-time bar graphs are available to monitor individual processor workloads to assess and re-partition if necessary in an effort to balance load distribution.

2D FFT Block Diagram Partitioning

Since partitioning in MultiProx occurs at the top-most level of hierarchy in a block diagram, the choices made when designing building blocks affect the partitioning options. For example, if a large portion of a design, in terms of computational complexity, were combined to form one functional block to be represented in the top-level block diagram, partitioning would allow only a single processor to handle the calculations of that block. This situation would affect the designer's ability and flexibility to partition a design with suitable load balancing. This may require re-examining the hierarchical structure entirely.

As discussed previously, two possibilities for partitioning a vector radix block diagram implementation of the 2D FFT are through either a horizontal or vertical partitioning of the diagram. To demonstrate the ease of MultiProx partitioning, the (4 x 4)-point FFT was partitioned vertically by columns using two processors. The partitioning process is simple and straightforward requiring drawing simple rectangles to enclose regions. In a similar manner, horizontal partitioning is possible. Each of tnese partitioning strategies suggests a mesh topology of processors such as the OSCAR I discussed in Chapter II. As of this writing, the AFIT SPW installation is not properly configured to allow simulation among multiple workstations to test the multiprocessor partitioning.

Summary

This chapter introduces the 2D FFT and describes how a particular 2D FFT algorithm, the vector radix FFT, may be designed with the use of SPW. The

parallelization possibilities of the 2D FFT are also discussed along with how SPW and the MultiProx enable a designer to quickly partition a block diagram for parallel operation. Analysis is limited to checking a design's functionality. The lack of support for analysis using parallel processing metrics makes comparison to theoretical analysis difficult. Other DSP development environments mentioned in Chapter II, Pegasus and RIPPEN for example, offer similar parallel processing support as SPW. These tools require test results from an implementation before useful comparisons may be made with theoretical predictions. The next chapter discusses how the algorithm may be implemented using the block diagram and the parallel processing analysis support SPW provides.

IV. Detailed Design and Implementation

Once an algorithm has been designed in block diagram form and successfully simulated and analyzed, there are two paths supported by SPW to implement the design. Code generation through CGS or VHDL generation through HDS are the two paths. This chapter covers the CGS and HDS options and why a designer would choose one or the other. SPW is also evaluated for whether or not it offers sufficient tools to allow the designer to evaluate trade-offs. The CGS and HDS options are evaluated in the context of implementing the 2D FFT block diagram design.

Code Generation using CGS

SPW's CGS option converts a block diagram design using the BDE into a C code implementation for execution on any platform which supports a C compiler [33:1-1]. This includes not only workstations, but also nodes of a multiprocessor machine and DSP microprocessors. The following sections describe C code generation for workstations and DSP microprocessors.

Standard C Code Generation System

The Standard C Code Generation System [33] produces generic C code which may be compiled and executed on any platform with a C compiler. There are several possible uses for generating C code from a block diagram for execution on a workstation or network of workstations in the case of MultiProx partitioned designs. First, executable code for workstations speeds simulation times. For the purposes of running multiple simulations on different data, the extra overhead of SPW's graphical interface can be

eliminated by compiling C code for execution on workstations independent of the SPW environment. However, during initial development, the Signal Flow Simulator may be the preferred method of simulation since design changes using CGS require recompilation and the simulator block libraries have built-in error handling. Second, C code is portable among all ANSI C compliant machines, and portability may be advantageous. Third, executable C code for a workstation may be all that is required for a particular application. A hardware solution is not always necessary. Executable C code or even simulation results alone may be sufficient for algorithm validation purposes.

All the steps leading to executable C code for a block diagram are performed automatically. A designer can create, compile, and even execute the C program for a block diagram from within SPW. Each block in a block diagram design has an associated "expression file" which is converted into instance-specific source code during code generation [33]. Figure 25 illustrates the different components of SPW which are involved.

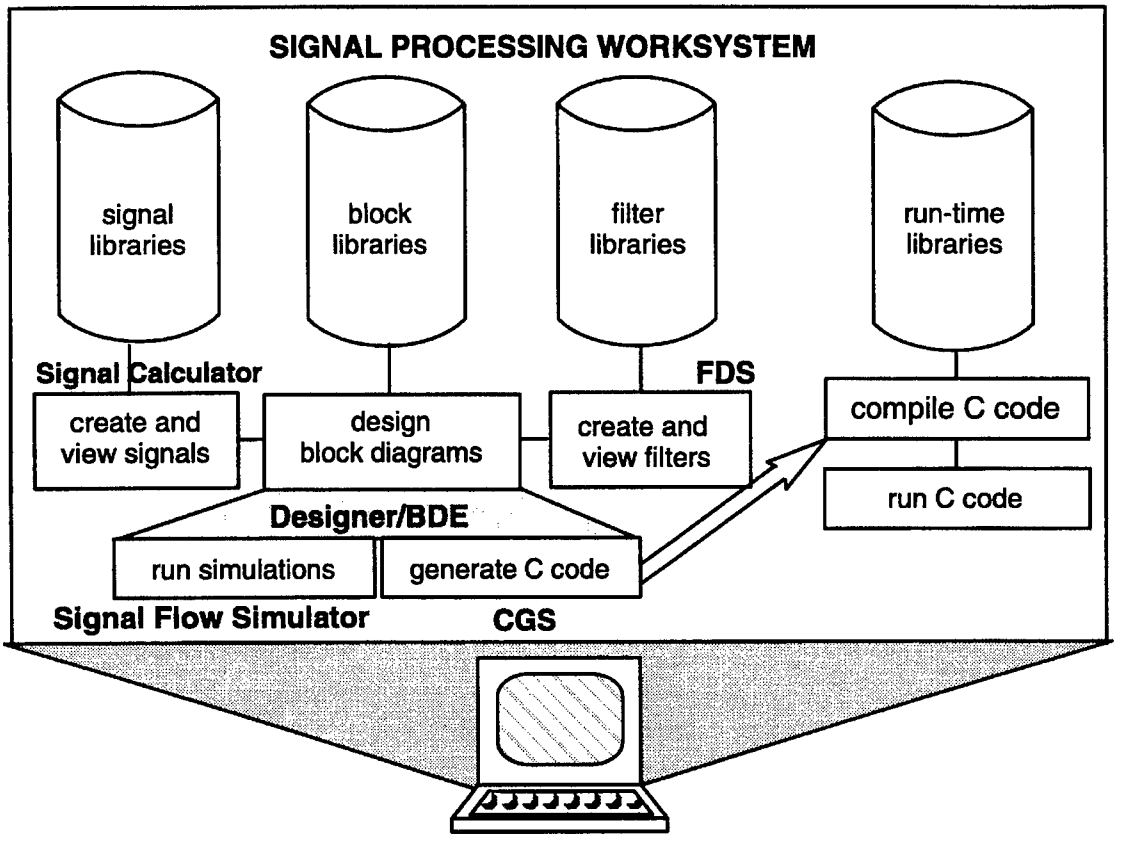

Figure 25 - Standard C CGS on Local Platform [33:1-3.]

In Figure 25, the Signal Calculator is used to create and view signals from either SPW or user signal libraries. Block diagram designs are built using blocks from the block libraries. If used, the Filter Design System (FDS) uses blocks from the filter libraries. The BDE brings these components together to run simulations using the Signal Flow Simulator and generate C code through CGS. A single SPW workstation is used to generate, compile, and run the C code. A remote workstation may be used for compilation and execution of the C code. This may be beneficial to perform compilation and execution in the background or to use a faster machine, one which cannot host SPW.

2D FFT Code Generation

Code generation for the (4 x 4)-point 2D FFT performed without error. The first step requires specification of the target platform as illustrated in Figure 26.

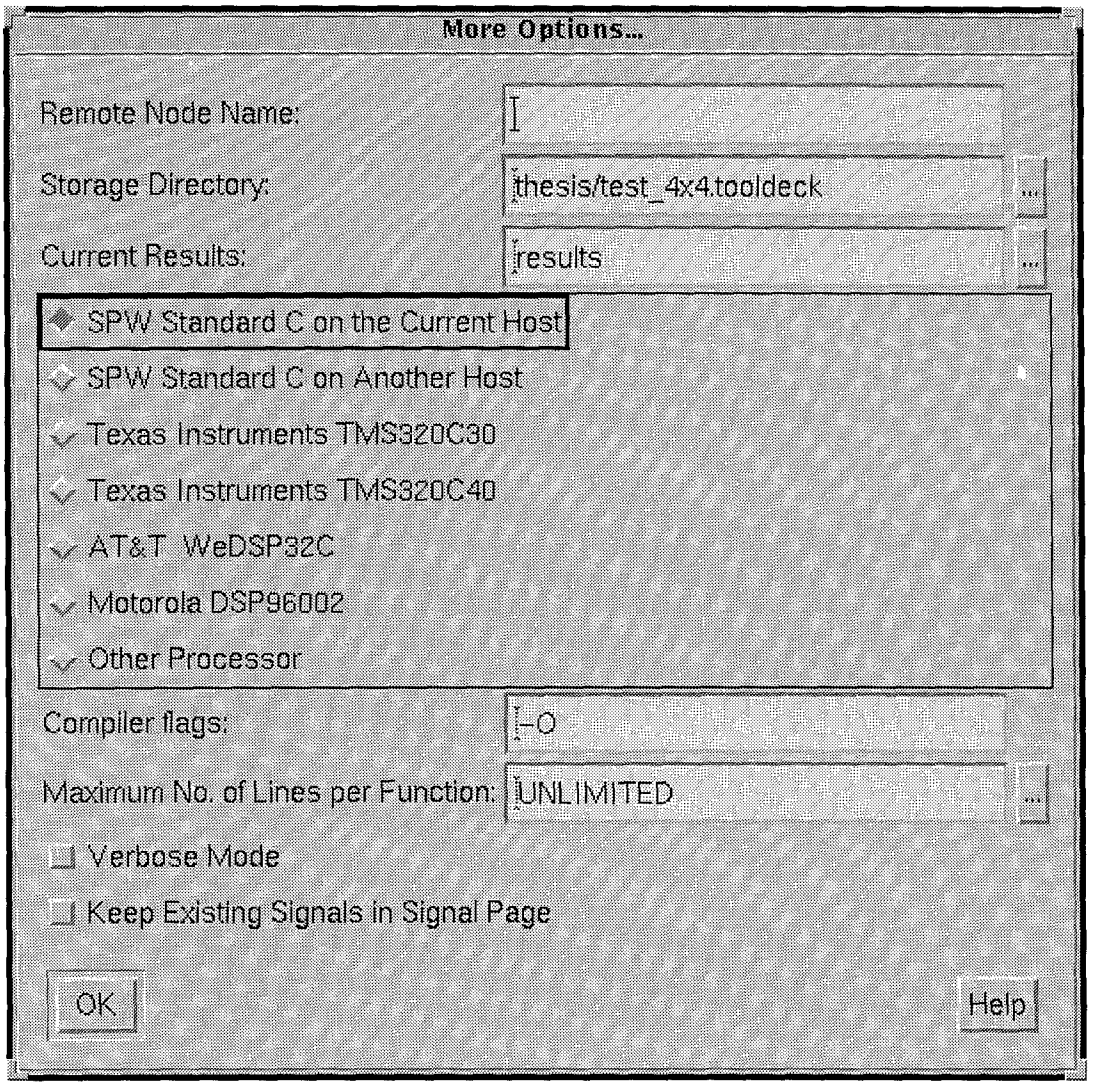

Figure 26 - Platform Selection [2]

Compilation and execution were also performed successfully from within SPW. The following figure shows the window from which these tasks are accomplished:

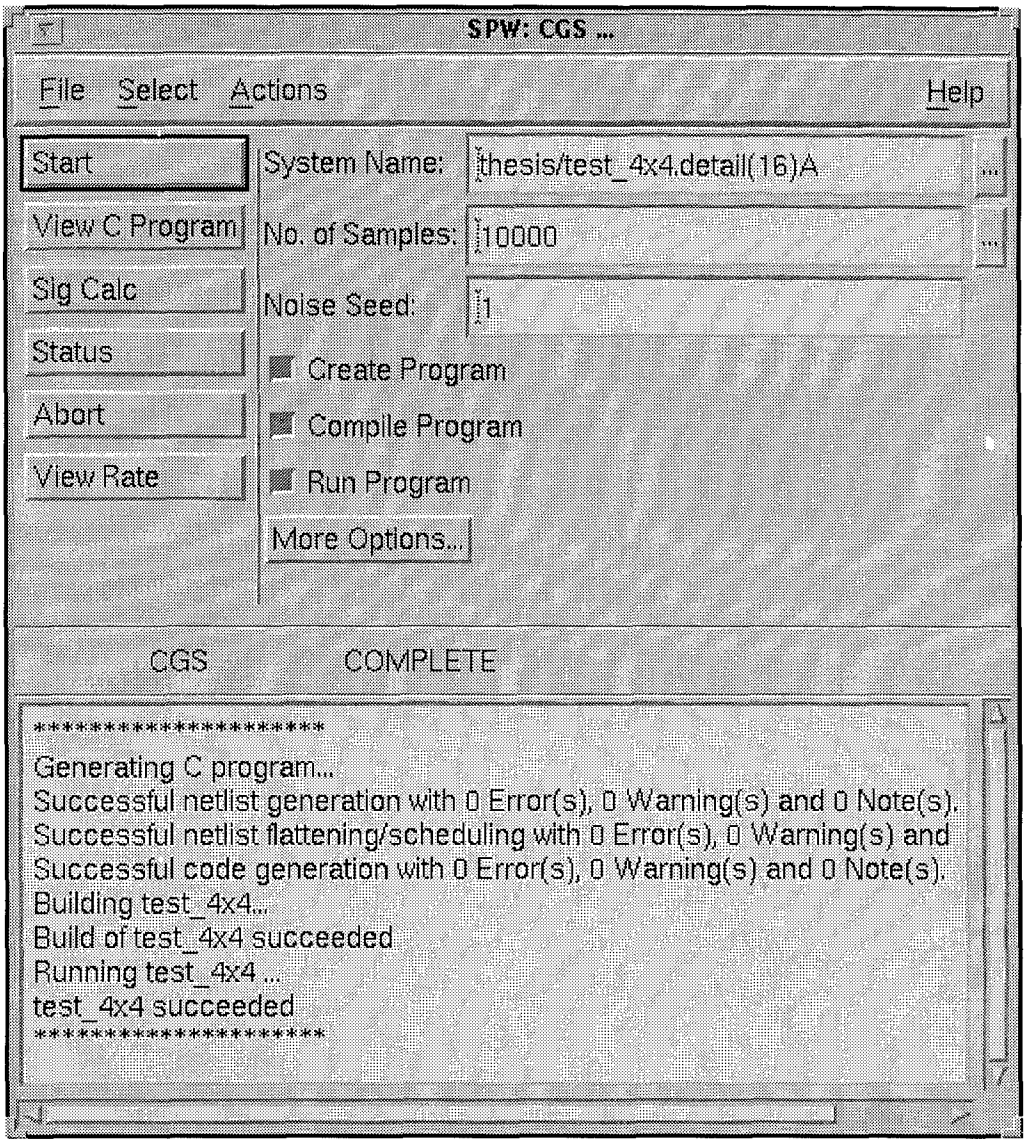

Figure 27 - CGS control window [2]

Status information regarding the creation, compilation, and execution are provided in the lower portion of this window. An interesting capability of CGS is that ISL block functionality is maintained even in generated C programs. This means that executable C programs can generate interactive windows if so desired.

SPW documentation suggests C code be generated to perform testing in lieu of slower simulations within SPW itself [33]. This is probably the only utility to code generation since execution times and code size do not compare with handwritten C implementations. For a (4 x 4)-point 2D FFT, SPW-generated C code execution times averaged 12.2 seconds over five runs. This is several orders of magnitude slower than the 2.33e-3 seconds reported for a (64 x 64)-point 2D FFT on a single node of an IBM SP2 [34]. The SPW-generated code execution times are however, faster than simulations within SPW which take on the order of minutes to complete. This said, SPW-generated code is best used for making multiple runs for validating block diagram design instead of using the slower Signal Flow Simulator. The structure of SPW-generated C code is provided in Appendix D.

The value of having portable C code is also suspect. Execution of C programs generated by SPW require SPW software licenses to be checked out. This means SPW must be running on the workstation. Remote compilation and execution is reportedly possible, however two entire libraries of .c and .h files must be copied to the remote machine in order to compile. The requirements of this process are briefly covered in the documentation [33].

SPW provides no means of evaluating algorithm design in terms of parallel processing metrics for C code generated for workstations. Any execution time analysis has to be performed by the designer through the use of UNIX system utilities outside of SPW. There is no facility to determine the amount of overhead required due to

interprocessor communication. The only support for parallel design analysis is provided with code generation for DSP microprocessors.

Code Generation System for DSP Microprocessors

The primary use for generating code for execution on a DSP or multiple DSPs is real-time testing on actual hardware representative of the final product. CGS supports the following DSPs:

- AT&T WE DSP32C
- Motorola M96002
- Texas Instruments TMS320C30
- Texas Instruments TMS320C40

The procedure for generating code for execution on DSPs is similar to the procedure for generating standard C code. The primary difference is the specification of the target DSP and the additional hardware required. A complete setup consists of a workstation and a PC hosting a development board containing one or more DSPs. Through an Ethernet connection to the PC host, the workstation running SPW transfers the C source files for compilation on the PC. The PC downloads the executable files to the DSP(s) on the development board. The results of the execution on the DSP are analyzed using the SPW Signal Calculator. The requirements of the host PC are

- IBM-PC AT or compatible computer with 640K bytes of memory
- DOS 3.3 or higher operating system
- PC-NFS file transfer software (version 4.0a or greater)
- PC-NFS compatible Ethernet interface between the PC and SPW workstation
- a compatible DSP microprocessor development board

Figure 28 illustrates the setup required.

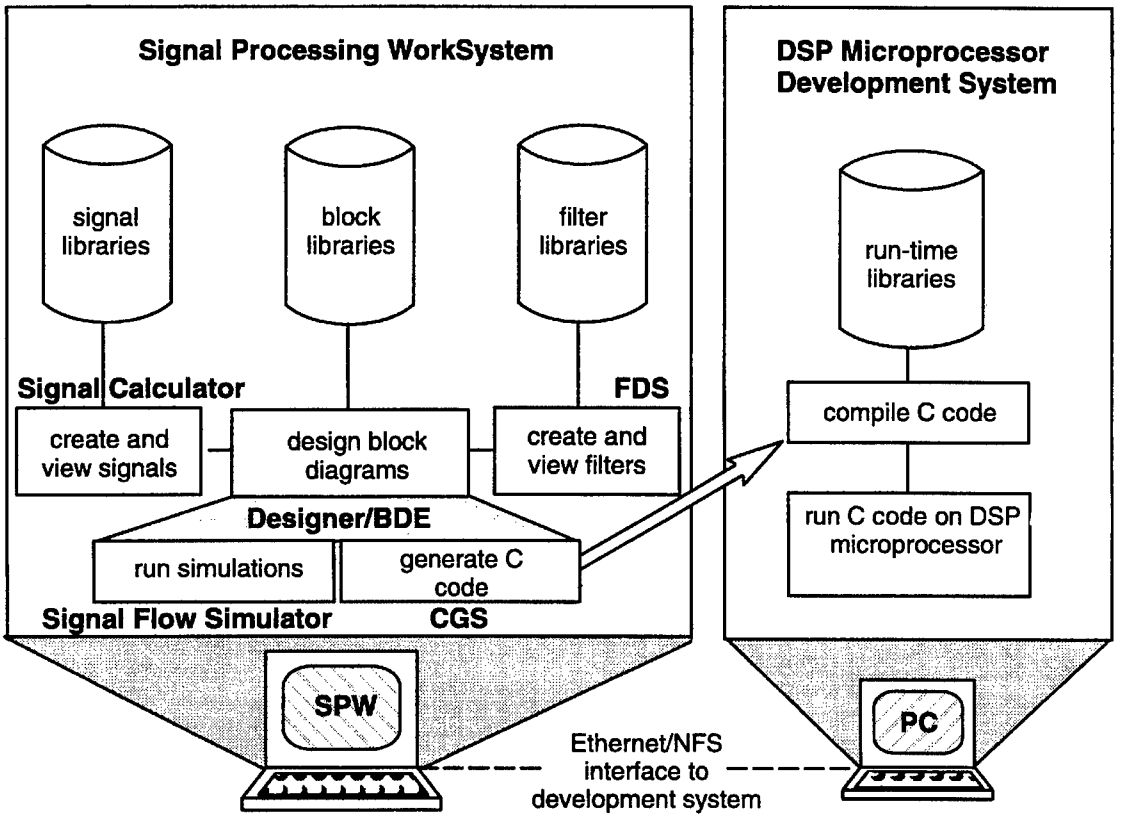

Figure 28 - CGS Using PC Development Board [33:1-4]

Figure 28 is very similar to Figure 25 with the exception of the added PC with development board. Since the program and data for a particular design must be stored in the memory and the PC. Complexity of the design may be limited to either's memory capacity. Both the PC and the development board may be controlled from the SPW workstation using CGS commands in the CGS Run Control window.

2D FFT Code Generation

The development board owned by WL is not supported by SPW for CGS control. The only TMS320C40 development board supported by SPW/CGS is the Loughborough Sound Images Ltd. Dual TMS320C40 Development Board with AM/D16SA Analog Daughter Module [33]. The current 386-based PC which hosts the development board currently does not have an Ethernet interface for connection to an SPW workstation. While some software is provided with the PC/development board, no utilities for communication to the board are provided. Therefore, while C code may be generated for the DSP microprocessors on the development board, without PC to development board communication utilities, programs may not be uploaded to the processors.

Use of a supported development board and control through SPW/CGS would allow a designer to evaluate the load balancing for a parallel 2D FFT design. In this setup, SPW, through the CGS Run Control window, can monitor processor workloads so that load balancing may be observed. Only relative processor workloads are shown without particular units of measurement. While this support is minimal, it would allow a designer to return to the block diagram level, repartition the design, and observe the effects of the changes on relative load balancing among the processors.

VHDL Generation Through HDS

The HDS option of SPW allows a designer to perform fixed-point simulations and generate Hardware Description Language (HDL) to permit hardware synthesis. Figure 29 illustrates the different components of the HDS and their relationships.

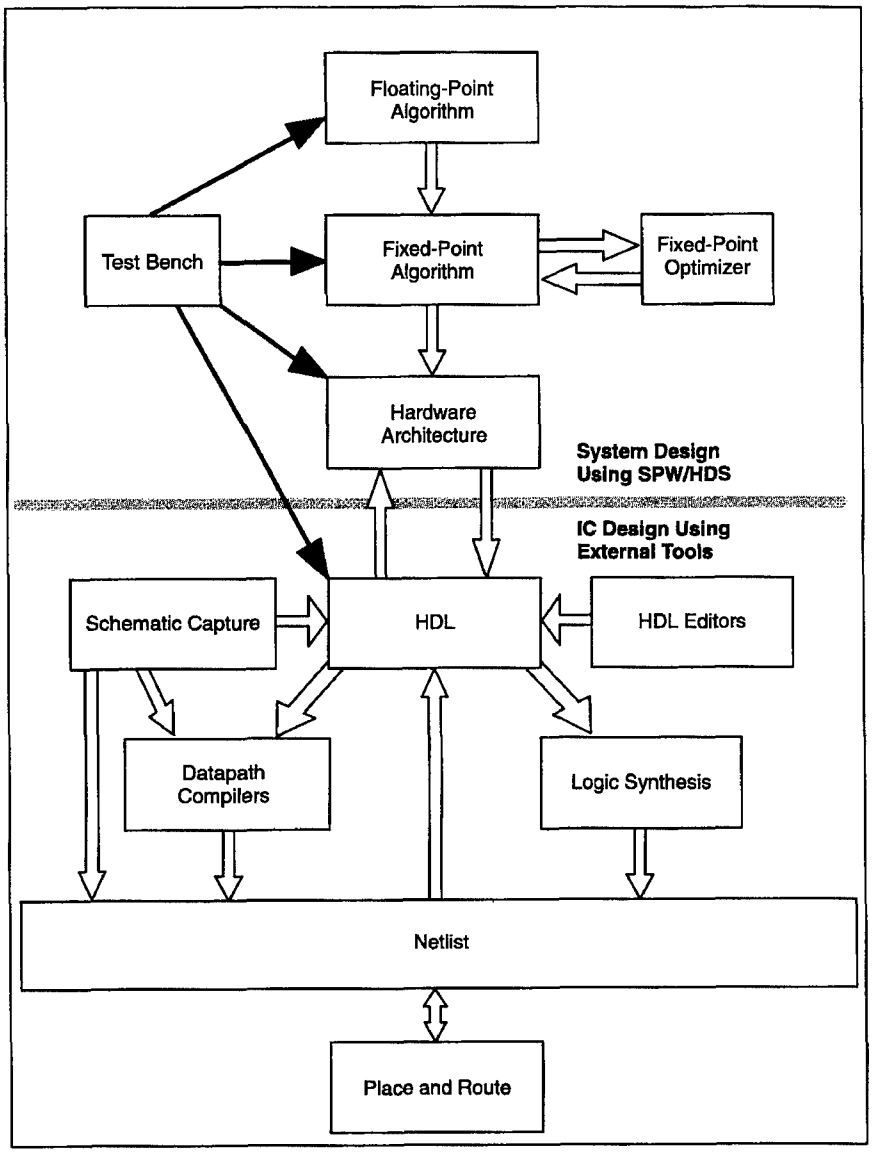

Figure 29 - Hardware Design Flow [35:1-11]

Figure 29 shows the paths to design fixed-point hardware systems. The upper half shows how SPW and HDS are used for high-level system design. HDS provides the links to the lower half where a HDL is used to specify the hardware implementation [33:1-10]. Since a hardware implementation is limited by a specified number of bits in a fixed-point design, a floating-point algorithm must be converted to a fixed-point algorithm. A fixed-

point design is less complex than a floating-point design in hardware, but care must be taken to choose the proper word length for anticipated data to be represented. Once HDL code is generated in HDS, external VLSI editors, compilers, and synthesis tools are used to generate a netlist which leads to silicon during place and route.

Before fabricating, a designer must make additional choices with regard to process technology and packaging. None these decisions are possible using SPW tools. Instead, VLSI experience is essential to assess which process technology and packaging is suitable for the design based on size, cost, and performance factors such as speed and power. While SPW allows for design and test of high-level algorithms, testing and eventual synthesis of SPW-generated VHDL code is left to any external VHDL tools and the designer's VLSI experience and knowledge.

HDS Main Library

The HDS Main Library contains about 90 functional blocks. A complete list and descriptions of the blocks is found in [35]. The following list of block categories provides examples of block types:

- Bit manipulation - bit and word merge/split
- Clocking
- Signal flow control - compare, counters, multiplexors
- Logic functions - Boolean operations, flip-flops, latches
- Mathematical functions - absolute value, add, subtract, increment/decrement, multiply, divide
- Simulation I/O - format conversion, sinks, sources
- Vector processing - vector constant, extract/join operations, vector sink/source

Not all of the blocks have the associated HDL code to perform HDL generation however. Two useful blocks which do support hardware synthesis are the Mealy state machine and combinational logic blocks [35]. The Mealy state machine block allows one to model and synthesize a state machine by specifying the state equations to establish inputs, outputs, states, and state transitions. The combinational logic block may be configured to represent any set of Boolean logic equations.

HDS Micro Library

The HDS Micro Library contains hardware architecture blocks. These include such items as arithmetic logic units (ALUs), encoders, decoders, registers, stacks, FIFO queues, dual-port RAMs, and shifters. A complete list along with descriptions of these blocks is provided in [35]. The presence of these blocks allows a designer to integrate the digital control with the signal processing portion of a system so the entire system may be simulated and eventually synthesized.

Floating to Fixed-Point Conversion Utility

This utility converts a design from a floating-point design to a fixed-point design by replacing all instances of floating-point blocks with their fixed-point counterparts [35].

Fixed-Point Optimizer

The Fixed-Point Optimizer performs multiple simulations repeatedly using different fixed-point attributes. From the simulation results, the Optimizer may determine the sign format and the minimum number of integer bits needed to prevent any overflow errors [35]. Fixed-point signal values are described by several attributes.

- total number of bits
- number of bits to represent the integer part of the value (excluding the sign bit)
- sign representation (unsigned 'u' or two's complement 't')

For example, in SPW's syntax, <8,2,t> indicates a total of eight bits with two integer bits, and the 't' stands for two's complement representation [35:2-1]. The Fixed-Point Optimizer serves to determine the fixed-point attributes based on simulation results. The Fixed-Point Optimizer is only beneficial if the designer has access to sample input signals which are representative of all anticipated signals.

HDL Link

The HDL Link is the component of HDS which translates an architectural design built with HDS blocks into an HDL description for synthesis.

HDL Simulator Interface

The HDL Simulator Interface enables a designer to compile and run HDL simulations from within SPW. The following HDL simulators are supported:

- Cadence: Verilog-XL
- Cadence: Leapfrog
- Ikos: Voyager
- Synopsys: VSS
- Vantage: Spreadsheet
- Model Technology: V-System

Co-Simulation

Co-Simulation enable simulations in the SPW environment and supported VHDL simulators to communicate during a single simulation run and therefore, provide a single

set of results. This may be useful if portions of the system are already described in VHDL or if writing a component's description in VHDL may be an easier task than creating a block diagram representation in SPW. This co-simulation scenario has obvious advantages in the flexibility it allows designers to combine different components into a unified system. It does, however, require a computing environment with both SPW/HDS and compatible VHDL tools. Wright Laboratory only maintains the basic SPW configuration and does not have the HDS option. During the course of this research, the Synopsys VLSI tools have been installed on the Zoo network where SPW currently resides to facilitate co-simulation. The configuration steps required to link the tools is not trivial and requires not only system administrator privileges but also in-depth knowledge of the installations of both the Synopsys and SPW packages. Currently, VHDL code generation is possible at AFIT, and while Synopsys Design Analyzer recognizes SPW VHDL libraries, user designs are not recognized. Attempts to compile the VHDL code also results in errors suggesting misplaced design files. This prohibits design synthesis and VHDL simulation.

2D FFT VHDL Generation

There are several steps which must be taken to allow VHDL code generation. Once VHDL code is generated, proper tool configuration is required to use that code. The first step is to convert a floating-point block diagram design into a fixed-point block diagram design. For this purpose, the aforementioned floating to fixed point conversion utility may be used. This is only successful if the functional blocks used in the floating-point design have equivalent fixed-point blocks. If this is not the case, portions of a

design must be redesigned in terms of lower-level fixed-point blocks. In the case of the 2D FFT Butterfly diagram, fixed-point blocks are not available for the floating-point addition block, and complex additions were constructed of simple addition blocks. A design completely composed of fixed-point blocks may be used to generate VHDL code through HDS. The generated code for the 2D FFT Butterfly is provided in Appendix E. This VHDL code can only be useful if a suite of VHDL tools is available. At a minimum, this consists of a VHDL compiler and a simulator. In order to compile the VHDL code, all of the VHDL code representing each of the blocks in a design must be available to the compiler. For this reason, it is useful to have both the SPW software and the VHDL tools hosted on the same network.

Summary

This chapter has describes the implementation options available once a design's block diagram has been simulated. C code generation through SPW's CGS enables execution on workstations or development boards consisting of DSP processors. C programs generated for workstation execution were found to be fast only when compared to performing simulations within SPW. Evaluation of a parallel design using parallel processing metrics is only available through a SPW-supported DSP development board or by analysis of C program execution external to SPW. With an SPW-supported development board, relative load balancing may be observed. Any other performance metrics, such as run-time, for a development board or C program implementation must be obtained externally to SPW. The lack of support to gather performance data prevents a designer from making comparisons between theoretical parallel metric values and

experimental results. SPW VHDL generation through the HDS component of SPW has the potential to allow VLSI synthesis tools to create custom hardware. In order for this to be feasible, the VHDL tools (compiler, simulator), SPW and the associated HDS VHDL libraries, and the VLSI synthesis tools should be hosted on the same network along with the links established to facilitate co-simulation.

V. Conclusions and Recommendations

Based upon the literature review and experience gained through the use of SPW and its various components, an evaluation of SPW to support rapid prototyping of parallel DSP algorithms is presented. Through designing an algorithm, the 2D FFT, representative of high-speed avionics applications and exploring the various implementation options, an evaluation of SPW is possible. This chapter begins with an overall evaluation of SPW. Since neither AFIT nor WL are currently configured to take advantage of all of SPW's functionality, recommendations on steps to rectify this situation are also presented.

SPW Review

As described in the Literature Review (Chapter II), properly designed software balances ease of learning, ease of use, and functionality. These factors were under continuous evaluation during the design and implementation of the 2D FFT. A review is provided in this chapter to summarize how SPW satisfies the criteria of good software design.

Ease of Learning

The amount of time necessary for a user to gain proficiency with a software package relies on many factors. The experience of the user is probably the greatest factor, but assuming a general knowledge in the application of interest, the software and accompanying documentation can ease the learning process. SPW, with the included tutorials and well laid out manuals, provides an excellent learning environment.

Completing tutorials while learning the features of a software program is an excellent hands-on approach as opposed to simply reading manuals. Each of the major components of SPW includes tutorials which lead the user through sample designs. For example, the tutorial for the CGS begins with a block diagram and step-by-step walks the user through the process to generate, compile, and execute C code. These tutorials frequently point out alternate methods of performing tasks as well. After completing the tutorials, the user is familiar with most of the functions needed to effectively use a particular component of SPW.

If the tutorials do not provide enough information on a particular task, the user's guides and the on-line help are available to fill in the gaps. Fully indexed user's guides may be consulted to learn about the details of any function. At any time while working within SPW, on-line help is available. The on-line help includes both a searchable database and context-sensitive help. For example, upon activating the context-sensitive help and selecting a particular functional block, SPW offers a window providing the basic description of that block. This help option also provides information on any window that is active. The combination of the tutorials, the user's guides, and the on-line help ensure a question may be answered by at least one source.

Ease of Use

SPW's user interface contains a number of features contributing to its ease of use. Among these features are toolbars which allow one click access to commonly used functions. Keyboard shortcuts are also available which shortcut access to functions within menus. User-defined macros are also available for sequences of commands which

are repeated often. The greatest contributor to the ease of use is the consistent interface and the ability to move seamlessly from one component of SPW to another. SPW includes a number of individual applications, but the communication and transition among them is transparent to the user.

File management, while adequate, can cause problems. Careful attention must be paid to the file structure SPW uses to organize user libraries. In other words, the user should never attempt to manipulate library files other than through SPW's own file manager utility. This includes copying directory structures for archiving. Such action attempted outside of SPW will not cause loss of data, but SPW may no longer recognize libraries. Another problem which may arise cause block symbols to lose their links to the detail models. This is solved by re-linking the model to the detail model from the menu. These two characteristics of SPW's file management can cause headaches for the uninitiated.

During simulation, two characteristics regarding error checking and reporting may cause problems. Simulation error messages do not always inform the designer of unconnected signals. Netlist checking and simulation may proceed without warning and produce erroneous results. Connections should be visually inspected throughout the design. Also, since individual functional blocks are not given specific identifiers, error messages which do appear refer to blocks only by type and SPW-assigned reference numbers. If a design requires many instantiations of the same type of block, debugging becomes difficult since the error messages do not allow the designer to readily distinguish among blocks of the same type. The impact of these error checking and reporting

problems may be lessened by using a modular design, validating modules at each step of the design.

Functionality

SPW for Block Diagram Design

The BDE component of SPW is essentially the hub of all design activity. For block diagram design, it should be intuitive for any designer with computer-aided design experience. If the many libraries don't provide a needed function, the BDE allows importing functions from other sources such as MATLAB, C source code, or VHDL code. Symbol creation is also very useful allowing the customization of a block for aesthetic and functional gain.

Design simulation is seamlessly integrated with the BDE and the Signal Calculator. The limited library of ISL blocks also could be helpful for real time analysis for a given design.

SPW for Code Generation

The CGS component may be executed from within the BDE. All functions execute from within the CGS control window. Like the other tools, CGS functions seamlessly executed from with the BDE. The efficiency of code generation is questionable however. For example, the C code generated for the (4 x 4)-point 2D FFT amounted to a staggering 102 pages. Examination of the code shows a portion of the code is used to emulate the ISL tools. Portability of the C code is suspect since executables require access to the original SPW directory structure for input and output

signal files. To execute the C programs on remote workstations, network access to the these files must be provided. The bloated code produced, along with restrictions placed on portability, severely limit the C programs from doing more than simply speeding up simulations.

If a DSP processor is the target platform, CGS only supports certain processors and development boards. This limitation requires a thorough evaluation of the computing requirements envisioned for the application, and subsequent acquisition of a supported development board hosting DSP processor to meet those requirements. Refer to Appendix D for a sample listing of DSP processor performance capabilities.

SPW for HDL Generation

The HDS component of SPW provides a means to design logic for the purpose of controlling signal processing systems. SPW was originally developed for signal processing algorithms, and HDS and the HDS libraries are not as mature. A good, basic set of blocks is provided, however many of the blocks only support fixed-point simulation and do not include accompanying VHDL code for VHDL generation purposes. Since HDS does support user-defined blocks from VHDL, the limitation of the library may be overcome. The capability for co-simulation in cooperation with VHDL simulation tools shows the most promise. The combination of SPW's signal flow simulator and a quality VHDL simulator allows greater flexibility in system design.

The designer must be cognizant of the fact that HDL Generation does not determine the area a design requires when synthesized for silicon layout. For example, if single set of VHDL code is generated for a given block diagram design, the resultant

synthesized layout may not conform to a die size which is cost effective. The design would have to be either partitioned at the block diagram level to generate subsections of VHDL or partitioned at the VHDL code level. This would require VHDL knowledge to create the interfaces required for these subsections of VHDL code. Only experience with the HDL generation capabilities of SPW and the synthesis capabilities of the VHDL tools will allow the designer to make the correct decisions.

Like the C code generated for a design, the generated VHDL code benefits from good organization attributed to the use of underlying VHDL building blocks. VHDL code generated for the basic (2x2)-point 2D FFT amounted to five pages of code (Appendix E) with each entity clearly specified and user-defined signal names used for VHDL signal names. Good coding practice calls for logically organized components, and VHDL hand-coding requires close attention to organization details throughout the design process. The HDS VHDL code generation not only provides rapid results, but it also enforces a consistent, logical framework. In the absence of development time and/or VHDL knowledge, HDS can provide well organized, rapid VHDL code results.

Summary

In summary, SPW and the complete set of optional components offers an excellent environment to allow a designer to proceed from block diagram design of an algorithm through hardware implementation. While the libraries may not include blocks needed for a particular application, the flexibility to add custom-coded blocks does not limit the designer to the included libraries. The interfaces between MATLAB and VHDL tools provides even greater flexibility by allowing the use of pre-existing MATLAB

functions or VHDL entities. Lastly, the excellent forms of documentation allow a designer to quickly gain familiarity with the wide range of tools available. The following table summarizes the criteria for well designed software [22].

Table 4- How SPW satisfies criteria of well designed software

CRITERIA	YES/NO	COMMENTS
Tutorials	YES	Clear, practical tutorials expose the new user to basic functions
Help facilities	YES	On-line help and user's guides provide quick guidance
Shortcuts to perform tasks quickly	YES	Keyboard shortcuts, macros, and toolbars
Necessary functions provided	YES	Extensive libraries available with ability to incorporate user-designed blocks and interface with MATLAB and VLSI tools
Good use of color	YES	Colors are used to distinguish functional blocks, signal lines, and text
Good use of icons	YES	Toolbars use icons for instant recognition of commonly used functions
Helpful error messages	NO	Error messages are somewhat cryptic in problem specification

For evaluating designs in terms of parallel processing metrics, SPW offers limited functionality to the user. During simulation, whether it be performed on a single workstation or a network of workstations, no simulation time information is provided in order to evaluate relative run-times. Generating C code programs does allow the designer to use UNIX system utilities to gather run-time information to evaluate speedup however. In either case, evaluating the effect of communication overhead is not directly possible for determining efficiency. The only useful means of evaluating a parallel design is through monitoring a DSP processor implementation on a supported development board. In this setup, SPW can monitor processor workloads so that load balancing may be

observed. While SPW has the potential to allow a designer to evaluate parallel DSP designs in terms of parallel processing metrics, this is accomplished only through SPW-supported DSP development boards. Refer to Table 1 in Chapter II for a list of other software packages with similar capabilities to SPW.

Recommendations

The optional components of SPW, MultiProx/CGS and HDS, are not fully utilized given the current configuration of the software on the AFIT Zoo network. A timely concept to hardware process is always desired, but this is required for AFIT's fixed-time degree programs. A proper software and hardware configuration in an AFIT laboratory has the potential to offer a student the opportunity realize an algorithm in testable hardware during the time allowed for thesis research. There are three scenarios envisioned each supporting different objectives.

1. Configuring SPW currently installed to support parallel DSP algorithm implementation on a network of workstations.

2. Acquisition of a compatible development board for hardware testing.

3. Complete configuration of SPW and Synopsys VHDL tools on the AFIT Zoo to support co-simulation and hardware synthesis.

SPW installed on the AFIT Zoo network of workstations currently includes all of the necessary options, CGS and MultiProx, to support design, simulation, and standard C code generation and execution. However, in order to simulate designs on multiple workstations, MultiProx must be properly configured to identify and communicate with

other workstations on the network. This is a minimal effort requiring the cooperation of network system administration. As it stands now, designs may be partitioned, but simulation among multiple workstations is not possible. Of the three scenarios, this does not require additional software or hardware. While code generation and execution for multiple workstations would be possible, the process would end there without an accompanying development board or an integrated VLSI software environment.

The second scenario requires the acquisition of a development board (~$2K) compatible with SPW, a PC to host that board, and an Ethernet connection between the PC and a workstation hosting SPW. To allow code generation for multiple DSPs, configuration changes similar to those already mentioned are required. A PC and associated development board allows a designer to design and simulate a block diagram on the SPW workstation and download CGS-generated code to the PC/development board for compilation and execution. PC and development board control and monitoring is possible through the SPW workstation. This interaction provides the best means of monitoring program status and allows analysis of input and output signals using SPW's Signal Calculator.

The third scenario requires completing the configuration of the SPW software and VHDL simulation and synthesis tools on the AFIT Zoo network. This allows for co-simulation whereby different components of a system may be described in both SPW block diagrams and VHDL code and simulated together. This setup also allows a designer to generate VHDL code in SPW/HDS and immediately simulate and/or synthesize hardware using the SPW-generated VHDL. While this scenario does not

require the purchase of additional software, it requires an understanding of the versions necessary and configuration required to support co-simulation and the link between tools for hardware synthesis. During the course of this research, Synopsys has been installed on the Zoo network, and steps have been taken to link the two software packages. However, configuration problems still exist. Resolution of these problems would be possible through an independent study with a student versed in the tools in question and direct contact with the Alta Group for troubleshooting.

As for the Texas Instruments Quad C40 DSP320 Development Board owned by WL, this board's use requires additional development software. The software allows compilation, assembly source code generation, and linking. There is also an archiving utility and a hex translator for an EPROM programmer. However, there is no debugger, emulator, or simulator. Also, there are no utilities to allow communication with the development board. Moreover, this particular development board is not supported by SPW for direct communication with CGS. Since this board's DSP processors are no longer the state-of-the-art and integration with SPW is limited, additional investment in this board is not recommended. If the capability to simulate designs in hardware is desired, purchase of a SPW compatible development board containing desirable processors is required. If any of the SPW-supported development boards do not use DSP processors which meet processing, cost, or power requirements, an unsupported board could still be used in conjunction with SPW at the expense of not having control of the board through SPW. The features and capabilities of DSP processors shown in Appendix A would have to be taken into account when selecting a suitable target processor.

Summary

This research shows the advantages that a software tool such as SPW can provide the DSP designer. The integrated design, simulation, and implementation features of SPW allows a designer to focus on higher-level design trade-offs rather than implementation details. Designing the parallel 2D FFT shows how the logical, consistent interface and functionality of SPW simplifies the design process. If commercially purchased, the current AFIT educational installation of SPW would cost over $100K, but the extended potential is not currently usable [36]. By investing only the time to make the modifications to current software configurations at AFIT or WL or purchasing a development board, the full potential of SPW to provide a complete rapid prototyping environment may be realized. A designer has the flexibility to design an algorithm at the block diagram level and implement the design in either a C code program for workstations and DSP processors or a VHDL code implementation for later synthesis and layout. The initial investment has already been made, and with additional time invested to configure and learn the various SPW components, significant dividends in long-term time savings may be achieved in the design and implementation of parallel DSP algorithms.

Appendix A - Guide to DSP Processors and Cores

The following page contains the Pocket Guide to DSP Processors and Cores. This guide provides a summary of current DSP processors and their capabilities and costs which represent general metrics to use when evaluating a DSP processor solution.

POCKET GUIDE TO DSP PROCESSORS AND CORES

Revised 10/18/95

Courtesy of BERKELEY DESIGN TECHNOLOGY, INC.

39355 California Street, Suite 206, Fremont CA 94538 Tel: (510) 791-9100; Fax: (510) 791-9127

E-mail: info@bdti.com

									On-chip Memory (in words) [2]						
							Address Space								
Manufacturer	Family	Core or Chip?	Arithmetic	Data Width	Program Width	Native MIPS [1]	Program	Data	Program ROM	Program RAM	Data ROM	Data RAM	Voltage(s)	Unit Price (Qty. 1,000) or Silicon Area [3]	Notes
Adaptive Solutions	CNAPS	Chip Set	Fixed-point	16 bits	64 bits	1280x1 MIPS	64 K	4 G	0	0	0	256 K	5.0	$435-$975 (2-chip set)	2-chip SIMD multiproc.; up to 1280 processors/chip.
Analog Devices	ADSP-21xx	Chip	Fixed-point	16 bits	24 bits	33 MIPS	16 K	16 K	12K	16 K	0	16 K	3.3, 5.0	$10-$81	Many family members w/ assorted peripherals
	ADSP-21csp01	Chip	Fixed-point	16 bits	24 bits	50 MIPS	16 M	16 M	0	4 K	0	4 K	5.0	$49	DMA controller, 2 serial ports,12 bit I/O pins
	ADSP-21020	Chip	Floating-pt.	32 bits	48 bits	33 MIPS	16 M	4 G	0	0	0	0	5.0	$106	Two off-chip memory buses
	ADSP-2106x	Chip	Floating-pt.	32 bits	48 bits	40 MIPS	4 G	4 G	0	170 K	0	256 K	3.3, 5.0	$196-$444	Strong support for multiprocessor designs
AT&T Microelectronics	DSP16xx	Chip	Fixed-point	16 bits	16 bits	70 MIPS	64 K	64 K	36K	12 K	36 K	12 K	3.0, 3.3, 5.0	$11-$61	Flash memory versions available for prototyping
	DSP32C	Chip	Floating-pt.	32 bits	32 bits	20 MIPS	4 M	4 M	0	1536	0	1536	5.0	$60-$101	Intended for PC multimedia applications
	DSP32xx	Chip	Floating-pt.	32 bits	32 bits	20 MIPS	1 G	1 G	0	2 K	0	2 K	3.3, 5.0	$37-$60	
Clarkspur Design	CD2400	Core	Fixed-point	16 bits	16 bits	25 MIPS	64 K	512	64 K	64 K	512	512	5.0	2.75 mm^2 (0.8 u CMOS)	Simple, very compact architecture
	CD2450	Core	Fixed-point	16-24 bits	16 bits	25 MIPS	64 K	128 K	64 K	64 K	128 K	128 K	5.0	3.86 mm^2 (0.8 u CMOS)	Adjustable data word width
DSP Group	PineDSPCore	Core	Fixed-point	16 bits	16 bits	40 MIPS	64 K	64 K	64 K	64 K	64 K	64 K	3.3, 5.0	8.2 mm^2 (0.8 u CMOS)	Provided as synthesizable HDL and layout
	OakDSPCore	Core	Fixed-point	16 bits	16 bits	40 MIPS	64 K	64 K	64 K	64 K	64 K	64 K	3.3, 5.0	8.4 mm^2 (0.6 u CMOS)	Provided as synthesizable HDL and layout
IBM Microelectronics	MDSPxxxx	Chip	Fixed-point	16 bits	24 bits	25 MIPS	32 K	32 K	0	0	0	0	3.3	$51	Intended for PC multimedia applications
Infinite Solutions	Green Core [4]	Core	Fixed-point	16 bits	16 bits	N/A	64 K	64 K	64 K	64 K	64 K	64 K	3.3, 5.0	5 mm^2 (est.) 0.5 u CMOS	RISC-like architecture
Motorola	DSP5600x	Chip	Fixed-point	24 bits	24 bits	40 MIPS	64 K	128 K	6400	4608	1 K	2688	3.3, 5.0	$25-$57	24-bit data word
	DSP561xx	Chip	Fixed-point	16 bits	16 bits	30 MIPS	64 K	64 K	12 K	2 K	4 K	4 K	5.0	$36-$41	Integrated 16-bit A/D and D/A
	DSP56301	Chip	Fixed-point	24 bits	24 bits	80 MIPS	16 M	32 M	0	4 K	0	4K	3.3	$80	PCI bus/host port, 2 timers, DMA, 3 serial ports
	DSP9600x	Chip	Floating-pt.	32 bits	32 bits	20 MIPS	4 G	8 G	0	1 K	1 K	1 K	5.0	$143-$157	Dual external memory buses
NEC	uPD7701x	Chip	Fixed-point	16 bits	32 bits	33 MIPS	64 K	96 K	24 K	1.5 K	24 K	6 K	3.0, 5.0	$17-$51	2 serial ports, one parallel port, 4 bit I/O lines
SGS-Thomson	D950-CORE	Core	Fixed-point	16 bits	16 bits	40 MIPS	64 K	128 K	64 K	64 K	128 K	128 K	3.3	10 mm^2 (0.5 u CMOS)	Coprocessor interface provided
TCSI	Lode Core [4]	Core	Fixed-point	16 bits	16 bits	N/A	64 K	64 K	64 K	64 K	64 K	64 K	3.3		Two MAC units
Tensleep Design	A/DSC321	Core	Fixed-point	16 bits	16 bits	12.5 MIPS	8 K	512	8 K	8 K	512	512	5.0	14.4 mm^2 (1.25 u CMOS)	Similar to TMS320C25
	A/DSC421	Core	Fixed-point	16 bits	16 bits	25 MIPS	16 K	2 K	16 K	16 K	2 K	2 K	5.0	10.2 mm^2 (1.0 u CMOS)	Similar to TMS320C25
	A/DSC521	Core	Fixed-point	16 bits	16 bits	30 MIPS	64 K	2 K	64 K	64 K	2 K	2 K	5.0	10.2 mm^2 (0.8 u CMOS)	Similar to TMS320C25
Texas Instruments	TMS320C1x	Both	Fixed-point	16 bits	16 bits	8.8 MIPS	64 K	256	8 K	0	0	256	3.3, 5.0	$4-$18	First commercially successful DSP
	TMS320C2x	Both	Fixed-point	16 bits	16 bits	12.5 MIPS	64 K	64 K	4 K	1.5 K	0	1.5 K	5.0	$11-$55	TI's second generation fixed-point DSP
	TMS320C209	Chip	Fixed-point	16 bits	16 bits	28.57 MIPS	64 K	64 K	4 K	4 K	0	4.5 K	5.0	$13-$14	
	TMS320C3x	Both	Floating-pt.	32 bits	32 bits	30 MIPS	16 M	16 M	4 K	2 K	4 K	2 K	3.3, 5.0	$26-$173	Low-cost versions compete with fixed-point DSPs
	TMS320C4x	Chip	Floating-pt.	32 bits	32 bits	30 MIPS	2 G	2 G	0	2 K	0	2 K	5.0	$130-$276	Intended for multiprocessor applications
	TMS320C5x	Both	Fixed-point	16 bits	16 bits	50 MIPS	64 K	64 K	32 K	9.5 K	0	10 K	3.3, 5.0	$14-$54	
	TMS320C54x	Chip	Fixed-point	16 bits	16 bits	50 MIPS	64 K	64 K	28 K	10 K	8 K	10 K	3.0, 5.0	$43-$47	TI's latest fixed-point family
	TMS320C80	Chip	Both	8/16 bits	32/64 bits	5 x 50 MIPS	4 G	4 G	0	12 K x 8	0	38 K x 8	3.3	$450	Contains 4 fixed-point DSPs plus a RISC CPU
3Soft Corporation	M320C25	Core	Fixed-point	16 bits	16 bits	15 MIPS	64 K	64 K	4 K	4.25K	64 K	544	3.3, 5.0	approx. 16,600 gates	Provided as synthesizable HDL
	M320C50 [4]	Core	Fixed-point	16 bits	16 bits	N/A	64K	64 K	64 K	64 K	64 K	64 K	5.0	approx. 40,000 gates	Provided as synthesizable HDL
Zilog	Z893xx	Chip	Fixed-point	16 bits	16 bits	20 MIPS	64 K	512	4 K	0	0	512	5.0	$4-$11	Very low cost
	Z894xx [4]	Chip	Fixed-point	16 bits	16 bits	N/A	64 K	64 K	0	1 K	0	1 K	3.3, 5.0		
Zoran	ZR3800x	Chip	Fixed-point	20 bits	32 bits	33 MIPS	1 M	1 M	12 K	1 K	12 K	4 K	5.0	$39-$42	20-bit data word is unique, intended for audio

NOTES:

[1] Native MIPS for fastest member of family.

[2] Maximum on-chip ROM and RAM for all family members. Memory sizes in native-sized words.

[3] Unit prices and silicon area data as quoted by manufacturers as of March 1995. Core silicon area excludes memory.

[4] Not yet fabricated.

Appendix B - Multidimensional FFT Vector Notation

By representing a multidimensional discrete Fourier transform (DFT) using matrices, the periodicities due to both the sampling lattice and the signal to be transformed [3030:45]. The transform pair for the multidimensional DFT is expressed as

$$X(m) = \sum_{n \in \chi_N} x(n) e^{[-jn^T (2\pi N^{-T}) m]} \tag{27}$$

and

$$x(n) = \frac{1}{J(N)} \sum_{m \in \chi_{N^T}} X(m) e^{[jn^T (2\pi N^{-T}) m]} \tag{28}$$

where

$x(n)$ is a multidimensional sequence periodic with period N, the periodicity matrix

$X(m)$ is a multidimensional sequence periodic with period N^T

χ_N is one period of x(n)

$J(N) = |\det N|$ or the density of the periodicity matrix [30:45-48]

These expressions offer the flexibility of mathematically representing multidimensional DFTs using sampling lattice structures other than rectangular matrices such as hexagonal or quincunx.

Appendix C - Vector-Radix FFT

The fundamental concept behind the vector-radix FFT algorithm is the decimation in time which occurs to express a (N_1 x N_2)-point DFT in terms of four $N_1/2$ x $N_2/2$ DFTs [28:77]. This assumes both N_1 and N_2 are divisible by 2. The following equation represents the direct calculation of the 2-D DFT as a double sum [28:75].

$$X(k_1 k_2) = \sum_{n_1=0}^{N_1-1} \sum_{n_2=0}^{N_2-1} x(n_1, n_2) W_{N_1}^{n_1 k_1} W_{N_2}^{n_2 k_2} \tag{29}$$

for

$$0 \le k_1 \le N_1 - 1 \text{ and } 0 \le k_2 \le N_2 - 1$$

and

$$W_N \equiv \exp(-j\frac{2\pi}{N})$$

Decomposing this summation into four separate summations yields [28:77]:

$$S_{00}(k_1, k_2) = \sum_{m_1=0}^{N/2-1} \sum_{m_2=0}^{N/2-1} x(2m_1, 2m_2) W_N^{2m_1 k_1 + 2m_2 k_2} \tag{30}$$

$$S_{01}(k_1, k_2) = \sum_{m_1=0}^{N/2-1} \sum_{m_2=0}^{N/2-1} x(2m_1, 2m_2 + 1) W_N^{2m_1 k_1 + 2m_2 k_2} \tag{31}$$

$$S_{10}(k_1, k_2) = \sum_{m_1=0}^{N/2-1} \sum_{m_2=0}^{N/2-1} x(2m_1 + 1, 2m_2) W_N^{2m_1 k_1 + 2m_2 k_2} \tag{32}$$

$$S_{11}(k_1, k_2) = \sum_{m_1=0}^{N/2-1} \sum_{m_2=0}^{N/2-1} x(2m_1 + 1, 2m_2 + 1) W_N^{2m_1 k_1 + 2m_2 k_2} \tag{33}$$

where S_{00} is for samples of x with even n_1 and n_2, S_{01} is for n_1 even and n_2 odd, S_{10} is for n_1 odd and n_2 even, and S_{11} is when both n_1 and n_2 are odd. Since each of these arrays is periodic in (k_1,k_2) with horizontal and vertical periods $N/2$ and $W_N^{N/2} = -1$, the following equations are derived [28:77]:

$$X(k_1,k_2) = S_{00}(k_1,k_2) + W_N^{k_2} S_{01}(k_1,k_2) + W_N^{k_1} S_{10}(k_1,k_2) + W_N^{(k_1+k_2)} S_{11}(k_1,k_2) \quad (34)$$

$$X(k_1 + \frac{N}{2},k_2) = S_{00}(k_1,k_2) + W_N^{k_2} S_{01}(k_1,k_2) - W_N^{k_1} S_{10}(k_1,k_2) - W_N^{(k_1+k_2)} S_{11}(k_1,k_2) \quad (35)$$

$$X(k_1,k_2 + \frac{N}{2}) = S_{00}(k_1,k_2) - W_N^{k_2} S_{01}(k_1,k_2) + W_N^{k_1} S_{10}(k_1,k_2) - W_N^{(k_1+k_2)} S_{11}(k_1,k_2) \quad (36)$$

$$X(k_1 + \frac{N}{2},k_2 + \frac{N}{2}) = S_{00}(k_1,k_2) - W_N^{k_2} S_{01}(k_1,k_2) - W_N^{k_1} S_{10}(k_1,k_2) + W_N^{(k_1+k_2)} S_{11}(k_1,k_2) \quad (37)$$

These four equations are used to compute four DFT points for a particular value of (k_1,k_2) from the four points $S_{00}(k_1,k_2)$, $S_{01}(k_1,k_2)$, $S_{10}(k_1,k_2)$, and $S_{11}(k_1,k_2)$ [28:77]. These equations may be illustrated as a radix-(2x2) butterfly shown in Figure 30.

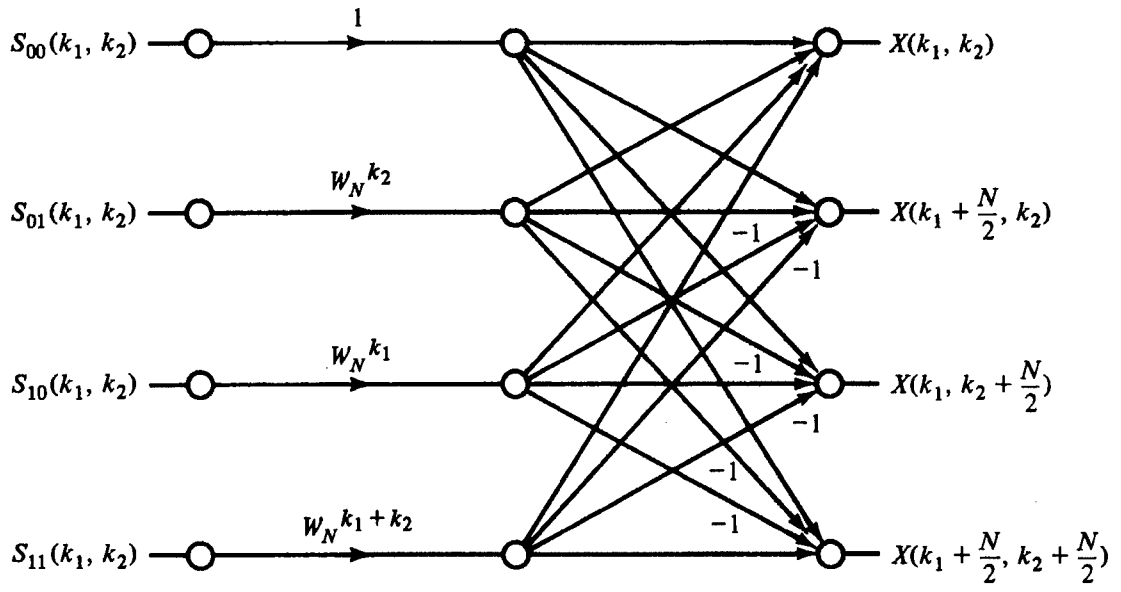

Figure 30 - Isolated radix-(2x2) butterfly[28:78]

For larger systems of points, decimation can occur $\log_2 N$ times (if N is a power of 2), and each stage has $N^2/4$ butterflies with three complex multiplications and eight complex additions in each butterfly [28:78].

Appendix D- SPW/CGS Generated C Code Structure

The C code generated by the CGS option of SPW is divided into seven different sections as described in [33]. The contents of the different sections are described in this appendix. While the generated C code is well structured, portability is limited due to strong ties to the SPW environment itself.

1. Preamble: The first part of the program, the preamble contains the function name, date of creation, and any needed *include* or *define* statements.

2. Variable Declaration: The parameter, output, and state variables for the system are declared in this section as static globals.

3. System Initialization: In system initialization, vectors are initialized and disconnected input/output buffers are zero-filled.

4. Parameter Initialization: Block parameters are initialized when variable are declared.

5. Local Variable Declaration: Variable which were declared locally for a block within the block declarations are declared once per block as *variablename_blockname*.

6. Initialize and Terminate Actions: Each of the blocks in the design is listed and identified by a comment in the form /* *library__ function__ model__ instance __* */

7. Run Code: This is the main iteration loop of the system with either a predetermined number of loops or an infinite loop which is executed until terminated.

8. Iteration Count Variable: This variable, a long int, holds the current iteration count.

Appendix E - SPW/HDS Generated VHDL Code for the (2x2)-point 2D FFT

```
--      ********************************************************************************
--      *                                                                              *
--      *         This confidential and proprietary software may be used only          *
--      *  as authorized by a licensing agreement from the Alta Group of               *
--      *  Cadence Design Systems, Inc.  In the event of publication, the              *
--      *  following notice is applicable:                                             *
--      *                                                                              *
--      *    (c) COPYRIGHT 1994 ALTA GROUP OF CADENCE DESIGN SYSTEMS, INC              *
--      *                          ALL RIGHTS RESERVED                                 *
--      *                                                                              *
--      *         The entire notice above must be reproduced on all authorized         *
--      *  copies.                                                                     *
--      *                                                                              *
--      ******************************************************************************** *

library IEEE;
library alta_synopsys;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use alta_synopsys.hds_fxp_alta.all;
use alta_synopsys.hds_proc_alta.all;
use alta_synopsys.hds_comp_alta.all;

entity hds_fft is
        port(   r_in10 : in std_logic_vector(8-1 downto 0);
                r_in00 : in std_logic_vector(8-1 downto 0);
                i_in10 : in std_logic_vector(8-1 downto 0);
                i_in00 : in std_logic_vector(8-1 downto 0);
                r_in11 : in std_logic_vector(8-1 downto 0);
                r_in01 : in std_logic_vector(8-1 downto 0);
                i_in11 : in std_logic_vector(8-1 downto 0);
                i_in01 : in std_logic_vector(8-1 downto 0);
                r_out00 : out std_logic_vector(8-1 downto 0);
                r_out10 : out std_logic_vector(8-1 downto 0);
                r_out01 : out std_logic_vector(8-1 downto 0);
                r_out11 : out std_logic_vector(8-1 downto 0);
                i_out00 : out std_logic_vector(8-1 downto 0);
                i_out10 : out std_logic_vector(8-1 downto 0);
                i_out01 : out std_logic_vector(8-1 downto 0);
                i_out11 : out std_logic_vector(8-1 downto 0));
end hds_fft ;


architecture hds_body of hds_fft is
```

```
        constant offset : integer := 0-2*fxpMinValue(0, -7 );

        signal sig_net_r_out00_0 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_r_out10_1 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_r_out01_2 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_r_out11_3 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_out00_4 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_out10_5 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_out01_6 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_r_in10_7 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_r_in00_8 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_in10_9 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_in00_10 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_r_in11_11 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_r_in01_12 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_in11_13 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_in01_14 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_15 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_16 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_17 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_18 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_19 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_20 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_21 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_22 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_net_i_out11_23 : std_logic_vector(0-1+1+offset downto 0-8+1+offset) ;
        signal sig_sys_clk : std_logic := '0';

begin -- architecture hds_body

        sig_net_r_in10_7 <= r_in10 ;
        sig_net_r_in00_8 <= r_in00 ;
        sig_net_i_in10_9 <= i_in10 ;
        sig_net_i_in00_10 <= i_in00 ;
        sig_net_r_in11_11 <= r_in11 ;
        sig_net_r_in01_12 <= r_in01 ;
        sig_net_i_in11_13 <= i_in11 ;
        sig_net_i_in01_14 <= i_in01 ;

        hds_fft_async_process : process
        ( sig_net_r_in10_7 , sig_net_r_in00_8 , sig_net_i_in10_9 , sig_net_i_in00_10 ,
sig_net_r_in11_11 , sig_net_r_in01_12 , sig_net_i_in11_13 , sig_net_i_in01_14 )

                variable var_net_r_out00_0 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_r_out10_1 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_r_out01_2 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_r_out11_3 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_i_out00_4 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_i_out10_5 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_i_out01_6 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_r_in10_7 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_r_in00_8 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_i_in10_9 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_i_in00_10 : signed(0-1+1+offset downto 0-8+1+offset) ;
                variable var_net_r_in11_11 : signed(0-1+1+offset downto 0-8+1+offset) ;
```

```
        variable var_net_r_in01_12 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_i_in11_13 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_i_in01_14 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_15 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_16 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_17 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_18 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_19 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_20 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_21 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_22 : signed(0-1+1+offset downto 0-8+1+offset) ;
        variable var_net_i_out11_23 : signed(0-1+1+offset downto 0-8+1+offset) ;

    begin -- process hds_fft_async_process

        var_net_r_out00_0 := signed(sig_net_r_out00_0) ;
        var_net_r_out10_1 := signed(sig_net_r_out10_1) ;
        var_net_r_out01_2 := signed(sig_net_r_out01_2) ;
        var_net_r_out11_3 := signed(sig_net_r_out11_3) ;
        var_net_i_out00_4 := signed(sig_net_i_out00_4) ;
        var_net_i_out10_5 := signed(sig_net_i_out10_5) ;
        var_net_i_out01_6 := signed(sig_net_i_out01_6) ;
        var_net_r_in10_7 := signed(sig_net_r_in10_7) ;
        var_net_r_in00_8 := signed(sig_net_r_in00_8) ;
        var_net_i_in10_9 := signed(sig_net_i_in10_9) ;
        var_net_i_in00_10 := signed(sig_net_i_in00_10) ;
        var_net_r_in11_11 := signed(sig_net_r_in11_11) ;
        var_net_r_in01_12 := signed(sig_net_r_in01_12) ;
        var_net_i_in11_13 := signed(sig_net_i_in11_13) ;
        var_net_i_in01_14 := signed(sig_net_i_in01_14) ;
        var_net_15 := signed(sig_net_15) ;
        var_net_16 := signed(sig_net_16) ;
        var_net_17 := signed(sig_net_17) ;
        var_net_18 := signed(sig_net_18) ;
        var_net_19 := signed(sig_net_19) ;
        var_net_20 := signed(sig_net_20) ;
        var_net_21 := signed(sig_net_21) ;
        var_net_22 := signed(sig_net_22) ;
        var_net_i_out11_23 := signed(sig_net_i_out11_23) ;

        hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_r_in11_11 ,
        in2 => var_net_r_in01_12 , outp => var_net_15 );
        -- Local Block Id: 5, Flattened Block Id: 624

        hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_r_in10_7 ,
        in2 => var_net_r_in00_8 , outp => var_net_16 );
        -- Local Block Id: 1, Flattened Block Id: 560

        hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_15 ,
        in2 => var_net_16 , outp => var_net_r_out00_0 );
        -- Local Block Id: 16, Flattened Block Id: 432
```

```
                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_r_in01_12 ,
                in2 => var_net_r_in11_11 , outp => var_net_17  );
                -- Local Block Id: 8, Flattened Block Id: 608

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_r_in00_8 ,
                in2 => var_net_r_in10_7 , outp => var_net_18  );
                -- Local Block Id: 3, Flattened Block Id: 544

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_17 ,
                in2 => var_net_18 , outp => var_net_r_out10_1  );
                -- Local Block Id: 14, Flattened Block Id: 448

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_16 ,
                in2 => var_net_15 , outp => var_net_r_out01_2  );
                -- Local Block Id: 12, Flattened Block Id: 464

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_18 ,
                in2 => var_net_17 , outp => var_net_r_out11_3  );
                -- Local Block Id: 9, Flattened Block Id: 480

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_i_in11_13 ,
                in2 => var_net_i_in01_14 , outp => var_net_19  );
                -- Local Block Id: 6, Flattened Block Id: 656

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_i_in10_9 ,
                in2 => var_net_i_in00_10 , outp => var_net_20  );
                -- Local Block Id: 2, Flattened Block Id: 592

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_19 ,
                in2 => var_net_20 , outp => var_net_i_out00_4  );
                -- Local Block Id: 15, Flattened Block Id: 496

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_i_in01_14 ,
                in2 => var_net_i_in11_13 , outp => var_net_21  );
                -- Local Block Id: 7, Flattened Block Id: 640

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 =>
var_net_i_in00_10 ,
                in2 => var_net_i_in10_9 , outp => var_net_22  );
                -- Local Block Id: 4, Flattened Block Id: 576

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_21 ,
                in2 => var_net_22 , outp => var_net_i_out10_5  );
                -- Local Block Id: 13, Flattened Block Id: 512

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_20 ,
                in2 => var_net_19 , outp => var_net_i_out01_6  );
                -- Local Block Id: 11, Flattened Block Id: 528

                hds_main_sum2 ( overflow_mode => clip , loss_mode => truncate , in1 => var_net_22 ,
```

```
                in2 => var_net_21 , outp => var_net_i_out11_23  );
                -- Local Block Id: 10, Flattened Block Id: 672

                sig_net_r_out00_0 <= std_logic_vector(var_net_r_out00_0) ;
                sig_net_r_out10_1 <= std_logic_vector(var_net_r_out10_1) ;
                sig_net_r_out01_2 <= std_logic_vector(var_net_r_out01_2) ;
                sig_net_r_out11_3 <= std_logic_vector(var_net_r_out11_3) ;
                sig_net_i_out00_4 <= std_logic_vector(var_net_i_out00_4) ;
                sig_net_i_out10_5 <= std_logic_vector(var_net_i_out10_5) ;
                sig_net_i_out01_6 <= std_logic_vector(var_net_i_out01_6) ;
                sig_net_15 <= std_logic_vector(var_net_15) ;
                sig_net_16 <= std_logic_vector(var_net_16) ;
                sig_net_17 <= std_logic_vector(var_net_17) ;
                sig_net_18 <= std_logic_vector(var_net_18) ;
                sig_net_19 <= std_logic_vector(var_net_19) ;
                sig_net_20 <= std_logic_vector(var_net_20) ;
                sig_net_21 <= std_logic_vector(var_net_21) ;
                sig_net_22 <= std_logic_vector(var_net_22) ;
                sig_net_i_out11_23 <= std_logic_vector(var_net_i_out11_23) ;
        end process hds_fft_async_process ;

        r_out00 <= sig_net_r_out00_0 ;
        r_out10 <= sig_net_r_out10_1 ;
        r_out01 <= sig_net_r_out01_2 ;
        r_out11 <= sig_net_r_out11_3 ;
        i_out00 <= sig_net_i_out00_4 ;
        i_out10 <= sig_net_i_out10_5 ;
        i_out01 <= sig_net_i_out01_6 ;
        i_out11 <= sig_net_i_out11_23 ;

end hds_body ;

--------------------------------------------------------------------------
--
--      Configuration declaration
--
--------------------------------------------------------------------------
--

-- synopsys synthesis_off

--
configuration hds_fft_hds_body_cfg of hds_fft is
  for hds_body
  end for;
end hds_fft_hds_body_cfg;
--


-- synopsys synthesis_on
```

Vita

Captain James C. Savage [redacted].

Captain Savage received his Bachelor of Science degree in Electrical Engineering from Rose-Hulman Institute of Technology in 1992. He earned his commission through the Air Force Reserve Officer Training Corps (AFROTC). Upon entering active duty in 1993, he was assigned to Air Force Materiel Command (AFMC), Aeronautical Systems Center (ASC), Subsystems Developmental Systems Office (SM). While there, he served as a test engineer for the Embedded Global Positioning System/Inertial Navigation System (EGI) program office. He left in 1995 to attend the Air Force Institute of Technology to pursue a Master of Science degree in Electrical Engineering. Captain Savage's follow-on assignment is the Armament Directorate of Wright Laboratories, Eglin AFB, Florida.

[redacted]

Bibliography

1 Haddad, Richard A. and Thomas W. Parsons. *Digital Signal Processing: Theory, Applications, and Hardware*. New York: Computer Science Press, 1991.

2 Alta Group of Cadence Design Systems, Inc., Signal Processing WorkSystem, version 3.0, 1995.

3 Higgins, Richard J. *Digital Signal Processing in VLSI*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1990.

4 Caracciolo, G. and J. Pridmore. "Architecture for Rapid Prototyping of Embedded Signal Processors," *The RASSP Digest*, Volume 2, Number 1, 1st Quarter, 1995.

5 IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987. New York NY: Institute of Electrical and Electronics Engineers, 1988.

6 Free On-Line Dictionary of Computing (FOLDOC). http://wombat.doc.ic.ac.uk/ (July 1996).

7 Kumar, Vipin and others. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Redwood City CA: The Benjamin/Cummings Publishing Company, Inc., 1994.

8 *DSP Design Tools & Methodologies*. Fremont CA: Berkeley Design Technology, Inc., 1995.

9 DEMACO, Inc., Xpatch (available through WL/AACT), 1993.

10 Kadrovach, B. A. *Hardware/Software Codesign Model for Xpatchf Optimization*. MS thesis, AFIT/GE/ENG/95D-08. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1995.

11 Suhr, Scott. *High Frequency Scattering code in a Distributed Processing Environment*, MS thesis, AFIT/GE/ENG/91J-04. School of Engineering, Air Force Institute of Technology (AU),Wright-Patterson AFB OH, June 1991 (AAI-4106).

12 Work, Paul. R. *Parallelizing Serial Code in a Distributed Processing Environment with an Application in High Frequency Electromagnetic Scattering*. MS thesis, AFIT/GE/ENG/91D-26. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.

13 Ambardar, Ashok. *Analog and Digital Signal Processing*. Boston MA: PWS Publishing Company, 1995.

14 Pocket Guide to DSP Processors and Cores.. Fremont CA: Berkeley Design Technology, Inc., 1995.

15 Lapsley, Phil. "Low Power Programmable DSP Chips: Features and System Design Strategies." in Proceedings of the International Conference on Signal Processing Applications and Technology, 1994.

16 Madisetti, Vijay K. *VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis*. Boston MA: IEEE Press, 1995.

17 Bier, Jeffrey C. and others. "A Class of Multiprocessor Architectures for Real-Time DSP," in *VLSI Signal Processing, IV*. Ed. Howard S. Moscovitz and others. New York: IEEE Press, 1991.

18 Baraniecki, M. R. and A. Z. Baraniecki. "VLSI Multiprocessor Architecture for Signal Processing," in *VLSI Signal Processing,* Ed. Sun-Yuan Kung and others. New York: IEEE Press, 1984.

19 Kurugollu, F. and others. "Advanced Educational Parallel DSP System Based on TMS320C25 Processors," Microprocessors and Microsystems, Volume 19, Number 3, April 1995.

20 Raja, Paruvachi V. R. and Suramaniam Ganesan. "An SIMD Multiple DSP Microprocessor System for Image Processing," Microprocessors and

Microsystems, Volume 15, Number 9, November 1991.

21 Weller, Franz. "Choosing Block-Diagram Tools for DSP Design," DSP & Multimedia Technology, April 1995.

22 Brown, C. Marlin. *Human-Computer Interface Design Guidelines*. Norwood NJ: Ablex Publishing Corporation, 1988.

23 Dumas, Joseph S. *Designing User Interfaces for Software*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1988.

24 Klaus, Ferdinand, Ute Blewonska, and Bernhard Bundschuh. "Flexible, Runtime Efficient Vector-Radix Algorithms for Multidimensional Fast Fourier Transform," Proc. SPIE International Society for Optical Engineering, Vol. 2247, June 1994.

25 Brigham, E. Oran. *The Fast Fourier Transform and its Applications*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1988.

26 Shannon, C. E. "A Mathematical Theory of Communication," The Bell System Technical Journal, Vol. 27, pp. 379-423, 1948.

27 Smith, Winthrop W. and Joanne M. Smith. *Handbook of Real-Time Fast Fourier Transforms*. New York NY: IEEE Press, 1995.

28 Dudgeon, Dan E. and Russell M. Mersereau. *Multidimensional Digital Signal Processing*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1984.

29 Akl, Selim G. *The Design and Analysis of Parallel Algorithms*. Englewood Cliffs NJ: Prentice-Hall, Inc., 1989.

30 Suter, Bruce W. *Multirate and Wavelet Signal Processing*. Charles Chui's Wavelet Series. Academic Press, to appear.

31 CLASSPACK Signal Processing Library/C. Champaign IL: Kuck & Associates, Inc., 1993.

32 Granata, J., M. Conner and R Tolimieri. "The Tensor Product: A Mathematical Programming Language for FFTs and other Fast DSP Operations," IEEE Signal Processing Magazine, pp. 40-48, Jan. 1992.

33 *Code Generation System User's Guide.* Sunnyvale CA: Alta Group of Cadence Design Systems, Inc., 1996

34 Alsing, Paul M. "Timings for the FFT in MPI." 7 Dec. 1995. http://www.arc.unm.edu. (20 Oct. 1996).

35 Signal Processing WorkSystem HDS Library Reference. Sunnyvale CA: Alta Group of Cadence Design, Systems, Inc., 1994.

36 *DSP Design Tools and Methodologies.* Fremont CA: Berkeley Design Technology, Inc., 1995.

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 1996	Master's Thesis

4. TITLE AND SUBTITLE
EVALUATION OF DESIGN TOOLS FOR RAPID PROTOTYPING OF PARALLEL SIGNAL PROCESSING ALGORITHMS

5. FUNDING NUMBERS

6. AUTHOR(S)
James C. Savage, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
2750 P Street
WPAFB, OH 45433-7126

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GE/ENG/96D-18

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
WL/AASH
ATTN: Dr. Robert Ewing
2241 Avionics Circle Ste 17
Wright-Patterson AFB OH 45433-7319

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A DSP application of interest to the Air Force is high-speed avionics processing. The real-time computing requirements of avionics processing exceed the capabilities of current single-chip DSP processors, and parallelization of multiple DSP processors is a solution to handle such requirements. Designing and implementing a parallel DSP algorithm has been a lengthy process often requiring different design tools and extensive programming experience. Through the use of integrated software development tools, rapid prototyping becomes possible by simulating algorithms, generating code for workstations or DSP microprocessors, and generating hardware description language code for hardware synthesis. This research examines the use of one such tool, the Signal Processing WorkSystem (SPW) by the Alta Group of Cadence Design Systems, Inc., and how SPW supports the rapid prototyping process from an avionics algorithm design through simulation and hardware implementation. Throughout this process, SPW is evaluated as an aid to the avionics designer to meet design objectives and evaluate trade-offs to find the best blend of efficiency and effectiveness. SPW is shown to be a viable rapid prototyping solution allowing an avionics designer to focus on design trade-offs instead of implementation details while using parallelization to meet real-time application requirements.

14. SUBJECT TERMS
Digital Signal Processing, Parallel Processing,
Electronic Design Automation, Multidimensional Fast Fourier Transform

15. NUMBER OF PAGES
120

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

GENERAL INSTRUCTIONS FOR COMPLETING SF 298

The Report Documentation Page (RDP) is used in announcing and cataloging reports. It is important that this information be consistent with the rest of the report, particularly the cover and title page. Instructions for filling in each block of the form follow. It is important to ***stay within the lines*** to meet ***optical scanning requirements***.

Block 1. Agency Use Only *(Leave blank)*.

Block 2. Report Date. Full publication date including day, month, and year, if available (e.g. 1 Jan 88). Must cite at least the year.

Block 3. Type of Report and Dates Covered. State whether report is interim, final, etc. If applicable, enter inclusive report dates (e.g. 10 Jun 87 - 30 Jun 88).

Block 4. Title and Subtitle. A title is taken from the part of the report that provides the most meaningful and complete information. When a report is prepared in more than one volume, repeat the primary title, add volume number, and include subtitle for the specific volume. On classified documents enter the title classification in parentheses.

Block 5. Funding Numbers. To include contract and grant numbers; may include program element number(s), project number(s), task number(s), and work unit number(s). Use the following labels:

C	- Contract	**PR**	- Project
G	- Grant	**TA**	- Task
PE	- Program Element	**WU**	- Work Unit Accession No.

Block 6. Author(s). Name(s) of person(s) responsible for writing the report, performing the research, or credited with the content of the report. If editor or compiler, this should follow the name(s).

Block 7. Performing Organization Name(s) and Address(es). Self-explanatory.

Block 8. Performing Organization Report Number. Enter the unique alphanumeric report number(s) assigned by the organization performing the report.

Block 9. Sponsoring/Monitoring Agency Name(s) and Address(es). Self-explanatory.

Block 10. Sponsoring/Monitoring Agency Report Number. *(If known)*

Block 11. Supplementary Notes. Enter information not included elsewhere such as: Prepared in cooperation with...; Trans. of...; To be published in.... When a report is revised, include a statement whether the new report supersedes or supplements the older report.

Block 12a. Distribution/Availability Statement. Denotes public availability or limitations. Cite any availability to the public. Enter additional limitations or special markings in all capitals (e.g. NOFORN, REL, ITAR).

- **DOD** - See DoDD 5230.24, "Distribution Statements on Technical Documents."
- **DOE** - See authorities.
- **NASA** - See Handbook NHB 2200.2.
- **NTIS** - Leave blank.

Block 12b. Distribution Code.

- **DOD** - Leave blank.
- **DOE** - Enter DOE distribution categories from the Standard Distribution for Unclassified Scientific and Technical Reports.
- **NASA** - Leave blank.
- **NTIS** - Leave blank.

Block 13. Abstract. Include a brief *(Maximum 200 words)* factual summary of the most significant information contained in the report.

Block 14. Subject Terms. Keywords or phrases identifying major subjects in the report.

Block 15. Number of Pages. Enter the total number of pages.

Block 16. Price Code. Enter appropriate price code *(NTIS only)*.

Blocks 17. - 19. Security Classifications. Self-explanatory. Enter U.S. Security Classification in accordance with U.S. Security Regulations (i.e., UNCLASSIFIED). If form contains classified information, stamp classification on the top and bottom of the page.

Block 20. Limitation of Abstract. This block must be completed to assign a limitation to the abstract. Enter either UL (unlimited) or SAR (same as report). An entry in this block is necessary if the abstract is to be limited. If blank, the abstract is assumed to be unlimited.