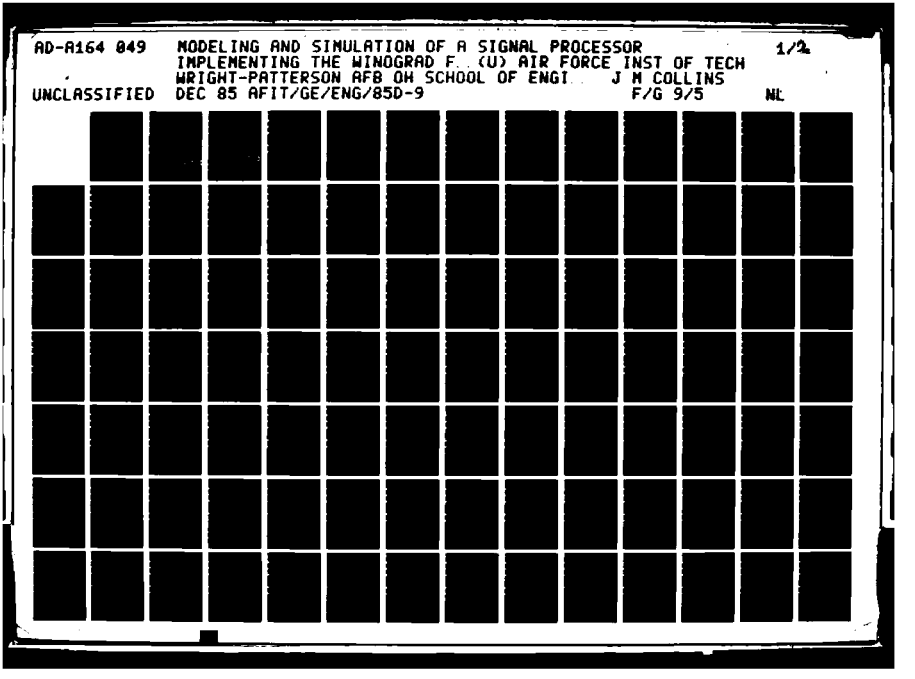
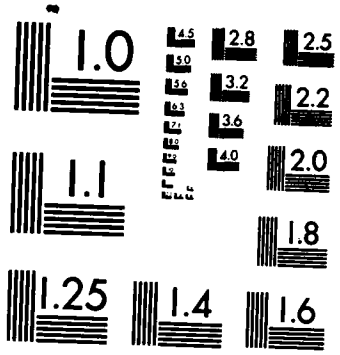

AD-A164 049 MODELING AND SIMULATION OF A SIGNAL PROCESSOR 1/2
IMPLEMENTING THE WINOGRAD F.. (U) AIR FORCE INST OF TECH
WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI J M COLLINS
UNCLASSIFIED DEC 85 AFIT/GE/ENG/85D-9 F/G 9/5 NL



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

MODELING AND SIMULATION OF A SIGNAL PROCESSOR IMPLEMENTING THE WINOGRAD FOURIER TRANSORM

James M. Collins, B.S.E.E, M.B.A.

Captain, USAF

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

Wright-Patterson Air Force Base, Ohio

AFIT/GE/ENG/85D-9

MODELING AND SIMULATION OF A SIGNAL PROCESSOR IMPLEMENTING THE WINOGRAD FOURIER TRANSORM

James M. Collins, B.S.E.E, M.B.A.

Captain, USAF

December 1985

Approved for public release; distribution unlimited

AFIT/GE/ENG/85D-9

MODELING AND SIMULATION OF A SIGNAL PROCESSOR IMPLEMENTING THE WINOGRAD FOURIER TRANSFORM

THESIS

Presented to the Faculty of the 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 M. Collins, B.S.E.E, M.B.A.

Captain, USAF

December 1985

Acknowledgement

I would like to thank my thesis advisor, Captain Richard Linderman, for the guidance and timely remotivation needed to ensure successful completion of this research. The members of the WFT research group, Captains Paul Coutee, Paul Rosssbach, and Kent Taylor provided a much needed source of answers to the many questions I had concerning the theory and operation of the hardware we were developing. I would especially like to thank Lieutenant Colonel Harold Carter and Paul Rossbach for the repeated explanations of the idiosyncrasies of the C programming language.

Most importantly, I would like to thank my wife, Ave Maria, whose introduction to married life came during the course of this effort. Without her support and understanding, this effort would not have been completed.

Jim Collins

TABLE OF CONTENTS

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF FIGURES (continued)

LIST OF TABLES

Abstract

The VHSIC Hardware Description Language was applied to the problem of modeling a VHSIC class circuit being designed by the VLSI design group at the Air Force Institute of Technology. A methodology was defined to decompose and model the circuit using the hierarchical facilities of the VHDL. The circuit embeds the Winograd Fourier Transform Algorithm into a pipelined serial architecture. This architecture was modeled using the VHDL and the C programming languages. A custom simulation tool was developed to verify the timing, control and hardware macrocells used to implement the WFTA processor. This simulation modeled the architecture at the bit level and validated the design.

MODELING AND SIMULATION OF A SIGNAL PROCESSOR IMPLEMENTING THE WINOGRAD FOURIER TRANSFORM

Chapter 1

Introduction

1.1. Overview

Continuing advances in the state-of-the-art of silicon fabrication technology have allowed a tremendous increase in both the functionality and performance that can be achieved by a single integrated circuit. The natural counterpart of this increased functionality is, of course, increased design complexity. Increased complexity limits the individual designer's ability to completely understand the circuit being designed. Thus, large ICs are now developed by design teams, leading to another problem, how to concisely and accurately communicate design information.

The formal language oriented approach, using hardware description languages (HDLs), is one method used to describe and model electronic circuits. Unfortunately, most HDLs were developed in a simpler time when IC functionality was limited to small and medium scale circuits. As we head into the very large scale, and very high speed integrated circuit (VLSI, VHSIC) era, there exists a need to develop tools that can both model and simulate these complex ICs in a concise and timely fashion.

Once military applications drove the state of the art in the electronics industry. Potential commercial spinoffs encouraged industry to pursue Department of Defense (DoD) business, as military Integrated Circuits (ICs) were sufficiently general purpose to be directly applicable to marketable products. As the industry grew, however, the DoD share of the total IC market fell to under 10% [11]. In addition, the continual need to

maintain technological superiority over potential adversaries required ever more complex special-purpose circuitry, driving the DoD into an increasingly specialized sector of the marketplace [11].

As military and civilian applications began to diverge, the military driving toward high speed signal processors, and the civilian market toward general purpose data processors, it became apparent to planners in the DoD that industry could no longer be expected to develop ICs directed towards military applications in a timely manner. Thus, in 1980, the DoD launched the Very High Speed Integrated Circuit (VHSIC) technology development program. Formulated as a seed program, it was designed to spur development of technology directed towards military needs. It was anticipated that once the technology was available industry would find civilian applications that would complement future military needs. Major goals of the VHSIC program are development of technology necessary to produce submicron devices, increased processing throughput, and the formulation of new circuit design methodologies and computer-aided design (CAD) tools required for maximum exploitation of the new technology [16].

Insertion of the new technology into existing weapons systems is considered a priority goal. The reduction of system size, weight, and power requirements using the new VHSIC class ICs over systems using current technology is expected to decrease the cost and increase the reliability/maintainability of the new systems. The VHSIC program office plans to demonstrate the replacement of over 50 ICs in current systems with one VHSIC chip. This implies that the VHSIC chip could have upwards of 250,000 logic gates, an extremely complicated part to design and validate. Modeling and simulation of a circuit of this complexity could easily be on the critical path towards a correct implementation of the intended function. However, current simulation languages are not capable of simulating large circuit designs in a timely manner.

After surveying existing Hardware Description Languages (HDLs), the VHSIC program office decided none would adequately meet its projected requirements and thus

funded the development of a VHSIC HDL (VHDL) to meet both present and anticipated applications. VHDL, based on Ada, the new DoD standard High Order Language, incorporates VHSIC specific requirements such as portability, maintainability, timing, and the ability to do hierarchical modeling and simulation. VHDL is now in its final design stage. A test version of the VHDL simulator is scheduled to be made available to the Air Force Institute of Technology (AFIT) for beta-site testing during the spring of 1986. Although VHDL has been designated as the DoD standard HDL for VHSIC circuitry, a significant amount of work remains to evaluate the language for its ease of use and clarity of syntax in the description of a VHSIC class chip.

1.2. Digital Signal Processing

Signal processing involves Fourier series analysis of continuous or discrete time-varying signals. With the advent of large-scale integration of digital systems, it became practical to implement complex signal processing functions on a single substrate. System designers began to foresee applications requiring Fourier analysis which were previously infeasible due to size and/or speed limitations of available analog or digital systems. Some current systems use Fourier analysis as the basis for pattern recognition systems. A time domain picture is taken and converted into the frequency domain by a fast Fourier transform (FFT) algorithm. Results are compared with a prestored spectrum to determine identity of the objects in the field of view. Fourier analysis of seismic feedback from explosions is a primary method of searching for petroleum deposits. Sonar detection of enemy submarines through processing of signal returns is another important defense application. Future applications include not only enhancements of current implementations but also many potential applications not currently feasible due to speed and size limitations of current technology. For example, a digital front-end for a phased-array radar, real-time computer resolution of satellite imagery, and medical needs, such as pictorial representation of internal body organs through low-level X-ray tomography, would benefit from more processing power than is available using today's

technology [1]. Advances in device technology must be matched with clever algorithmic design to reduce the computational burden in order to bring these applications into the realm of feasibility.

1.3. Winograd FFT

The Winograd Fourier Transform Algorithm (WFTA) is a method for implementing a Discrete Fourier Transform (DFT) for signal processing. It offers the potential for a tenfold increase in processing throughput over existing signal processing algorithms. A group of AFIT graduate students is designing a WFTA processor that will be implemented using 1.2μ CMOS technology similar to that developed in VHSIC Phase I.

1.4. Statement of the Problem

The problem addressed in this thesis is to analyze the effectiveness of the VHSIC Hardware Description Language (VHDL) for modeling large CMOS integrated circuits, and to verify the architecture, data flow, and control sequencing of the 16-point Winograd FFT signal processor.

The major portion of the research is directed toward analysis of VHDL as a tool useful in VLSI design. This analysis covered learning the language syntax, development of a methodology to be used for VHDL modeling, and modeling the primary CMOS circuits that make up the WFTA processor. In support of the WFTA verification effort, a model of the 16 point architecture was developed using the C programming language. This model completely describes the arithmetic and control functions of the processor at the bit level. It verified correct operation of the algorithmic implementation, and was exercised to generate test vectors for future VHDL simulations and hardware testing.

1.5. Problem Environment

The research reported in this thesis is one of four related efforts working toward the design and implementation of VLSI signal processors that implement the Winograd Fourier Transform. Captain Kent Taylor [17] developed the architecture of the WFTA

chip from the original concept developed by Linderman [8]. Taylor's thesis covers theoretical development, numerical performance, and control and timing details of the processors. He developed and validated programs that performed FFTs using the 15, 16, and 17 point Winograd algorithms. Captain Paul Rossbach [13] designed and implemented the control portion of the WFTA chip. An interim control sequencer test chip was designed, fabricated, and tested at clock rates exceeding 50Mhz. He also designed and implemented a X (shaped storage cell) Read Only Memory (XROM) to provide the data addresses to an off-chip Random Access Memories (RAM) in the order specified by the Chinese Remainder Theorem. The XROM has been optimized to minimize the number of transistors by a solution to both the graph partitioning and the traveling salesman problems using the approach of Kernighan and Lin [7]. Finally a silicon compiler was written to automatically place the address sequencing scheme into the XROM personalization mask. Captain Paul Coutee [4] developed and implemented the serial adders and multipliers used in the processor's arithmetic section. The multipliers are derived from Lyon's serial multiplier architecture, but redesigned to use fixed coefficients [9]. In addition, the horizontal and vertical pitch was minimized. The resulting dense cell structure is critical towards achieving the goal of an entire Winograd processor on a single silicon chip. In addition, cells were designed to check and generate parity and to perform arithmetic rounding of the results.

1.6. Summary of Current Knowledge

Hardware Description Languages are not a new item. As early as 1939, Shannon used a type of HDL in his work on switching circuits [8]. Nor are they rare. In a special IEEE issue on HDLs Liposki noted that whenever someone developed a circuit simulator they felt compelled to develop a HDL to drive it rather than learn and adapt an existing one to their application [8]. In other special issues on HDLs by the IEEE Computer Society, writers have called for a common HDL [3]. It was noted that although there were many languages that were adequate for a specific purpose, none were suitable for

application over the entire range of a large hardware design project. The IEEE has sponsored project CONLAN (CONsenus LANguage) to develop a group of languages linked by common syntax and design conventions. The new language would use desirable features and concepts from the myriad existing languages and incorporate these into its basic syntax Base CONLAN [12].

Around the same time the DoD, faced with an explosion in the number of software languages in its computer systems, launched an effort to slow the growth of the cost of software maintenance. After studying the problem the DoD concluded that computer languages had not kept pace with the advances in technology. Accordingly an effort was made to develop a language incorporating both features in current languages and modern concepts in software engineering such as structured programming, information hiding, data abstraction, real time control and data handling. The result was the Ada Programming Language which has been designated as the standard DoD High Order Language [2]. The VHSIC program office, looking at the problem of concisely communicating design information on integrated circuits containing up to 250,000 gates, recognized that the basic concepts and constructs used in Ada could be used in a new HDL. The relationships between VHDL and Ada are detailed in the VHDL Design Analysis and Justification [6]. In general VHDL constructs supported by Ada were required to use the Ada syntax [6]. The basic objectives of the VHDL are:

1. It be capable of documenting digital hardware over the range of entire systems to logical gates.
2. It be able to be used as a design and documentation tool
3. Its complexity be kept to a minimum.

A contractor team of Intermetrics, IBM, and Texas Instruments was selected to develop the VHDL. The contract was for a two-phase design effort followed by a testing phase. AFIT was selected as a test site to determine if VHDL meets the requirements set forth in the requirements documents, and if the VHDL is a practical tool for use in VLSI

design development.

1.7. Approach

As has been stated, the main thrust of this thesis effort has been to analyze the effectiveness of the VHSIC Hardware Description Language (VHDL) as a tool for modeling the detailed design of a VHSIC class chip. This analysis will be accomplished by structurally decomposing the 16-point processor into its constituent processing elements and modeling the primary CMOS circuits which make up those processing elements. In addition, the arithmetic and control sections of the architecture will be modeled at the bit level using the C programming language. This will serve to verify the correctness of the architecture and the circuit design.

The modeling of the architecture has been accomplished by a structural decomposition of the system into subsystems, components of those subsystems, macro cells and finally the microcells that make up the cells. Decomposing the architecture led to the definition of the hardware interfaces. This top-down interface definition imposed a signal flow structure on the system that was followed by the definition of the internal circuitry. Once the chip was decomposed into its smallest individual logic components, the micro and macrocells were modeled using VHDL library descriptions as well as user defined descriptions. In this fashion the system could be reconstructed following the previously defined interfaces.

Subgoals of the modeling process were to establish functional equivalency between the simulation program and the actual hardware, development of test cases to simulate various data sets, and development of test vectors for use in future VHDL simulations and hardware testing.

1.8. Sequence of Presentation

Chapter 2 reports on the development of the architecture of a signal processor based on the Winograd algorithm. Details on the Winograd Transform, the Good-

Thomas prime factor algorithm, the Chinese Remainder Theorem and their implications for the system architecture are included in this chapter.

Chapter 3 presents the VHDL constructs used to model hardware. A hardware *entity* is described as it is used in the VHDL. Examples will be used to illustrate a methodology to be followed when modeling circuits.

Chapter 4 details the modeling of the 16-point processor using VHDL. The 16 point processor is completely decomposed into the smallest independent circuits, inverters and transmission gates, which are then used to construct the primary cells. The VHDL descriptions and modeling of the major cells are presented.

Chapter 5 presents the C simulation used to verify the 16-point architecture. A discussion of the need for system simulation is presented, followed by a description of the general approach used in program development.

Chapter 6 is an analysis of the utility of the VHDL as a VLSI design tool. Recommendations for applications of the C simulations are presented. Finally the recommendations and conclusions based on the research performed while carrying out this thesis will be presented.

CHAPTER 2

Development of the WFTA Architecture

2.1. Overview

As stated in the first chapter, the Winograd DFT algorithm is a computationally efficient method of computing the discrete Fourier transform. It is of interest in VLSI because the matrix form of the algorithm maps very efficiently, in terms of space, and regularity of structure, into a signal processing architecture. In addition, by combining various Winograd modules into a pipelined architecture in the manner specified by the Good-Thomas Prime Factor algorithm, large data blocklengths may be computed. The development of an architecture based on these algorithms is discussed in this chapter. The approach will be to introduce the Fourier transform, how it is used signal processing applications, and then demonstrate how a more efficient implementation of the basic Fourier transform leads to the architecture modeled in this thesis. A 4080 point block length is initially assumed and later justified in section 2.2. Concepts which will be introduced in this section include the Good-Thomas Prime Factor algorithm, the Chinese Remainder theorem, the Winograd Fast Fourier Transform algorithm and cyclic convolution.

2.2. Fourier Series Representation

Most signals of interest in communications or signal processing applications can be described as a function of time by the equation:

$$f(t) = A \sin(\omega t + \phi) \tag{2-1}$$

where A is the signal Amplitude.

ϕ is the signal Phase.

ω is the frequency in radians/sec.

Signals which conform to the relaxed Dirchilet conditions [15]:

1. f(t) has only finite number of maxima and minima in the interval.
2. f(t) has only a finite number of discontinuities in the interval.
3. f(t) satisfies the inequality:

$$\int_0^T \left| f(t) \right|^2 dt < \infty \tag{2-2}$$

may be represented by the Fourier series which is defined as:

$$f(t) = \sum_{n=-\infty}^{n=\infty} F_n e^{(jn\omega t)} \tag{2-3}$$

where:

F_n is a complex coefficient representing the initial phase angle and magnitude

the exponential, $e^{(jn\omega t)}$, represents phasor rotation at angular frequency ω.

By summing the phasors, $e^{j\omega t}$, over the index the instantaneous amplitude and phase of the original signal can be determined. In addition, the Fourier coefficients, F_n, can be summed to find the average signal power. A plot of the Fourier coefficients versus frequency is known as the spectrum of the signal. Characteristics of the spectrum are (1) that its envelope is dependent on the pulse shape, and (2) there is an inverse relationship between pulse width and frequency spread.

The Fourier transform, used to calculate the Fourier coefficients, is defined by the equation:

$$F(\omega) = \int_{\frac{-T}{2}}^{\frac{T}{2}} f(t) e^{-(j\omega t)} dt \tag{2-4}$$

Using the Fourier transform we can describe any signal of the form (2-1) in terms of a spectral density function of the form (2-4).

2.3. Fast Fourier Transforms

Many of the problems in digital signal processing involve computation of the Discrete Fourier Transform (DFT) for finite input sequences of real or complex data points. The DFT of a complex data sequence v is given by:

$$V_k = \sum_{i=0}^{i=n-1} \omega^{ik} v_i \qquad (2\text{-}5)$$

where:

n is the blocklength of the data sequence

ω is the complex phasor $e^{-(j2\pi/N)}$

v is a vector of complex numbers.

The DFT can be computationally expensive. The number of complex additions and multiplications is $O(N^2)$. For example, a direct implementation of a 4080 point DFT will require 16,646,400 multiplications and 16,642,320 additions. The body of theory labeled Fast Fourier Transforms is concerned with manipulation of input and output data indices in order to achieve a more efficient means of performing this DFT operation. FFT algorithms generally use a variety of methods to shuffle elements around in the data matrices to reduce the number of multiplications required. Figures of merit for FFT algorithms revolve around the numbers of additions and multiplications, with replacement of multiplications by additions being the preferred approach to achieve a more efficient algorithmic implementation. Fast multipliers are costly in terms of silicon area and processing time. Reduction of multiplications in favor of additions reduces the space requirements of the multiplier section and decreases latency through the pipeline. Additional space freed up can then be used to allow a smaller die size, resulting in greater yield, or to implement desirable features such as error detection, correction, and other fault tolerance measures.

The Winograd Fourier Transform Algorithm (WFTA) architecture was developed using both the Winograd and Good-Thomas Prime Factor algorithms. The Good-Thomas algorithm is used to break the 4080 point blocklength into mutually prime

sequences of length 15, 16, and 17. These smaller blocklengths are computed using the Winograd FFT algorithms. Combining the Good-Thomas Prime Factor Algorithm (PFA) and the WFTA in this fashion will reduce the number of operations to 31,148 multiplications and 157,164 additions [17]. This represents a reduction in the number of multiplications by a factor of over 500. We now wish to examine the theory which allows us to decompose the 4080 point DFT in order to achieve these reductions.

2.3.1. Good-Thomas Prime Factor Algorithm. The Good-Thomas PFA allows the representation of a linear array of n data points as an m-dimensional array in such a manner as to allow calculation of a sequence of true m-dimensional Fourier Transforms. The CRT is used to map the sequential data addresses onto a unique location in a m-dimensional hypercube. In order to use the CRT the decomposition factors, $m1$, $m2$, and $m3$ must be relatively prime (sharing no common factors). Considerations for selection of a WFTA block length were computational efficiency of the pipeline and adaptability to existing signal processing systems.

Pipelined architectures achieve maximum efficiency when all processors require approximately the same time to compute each problem. Current radar systems use 4096 point scans for signal processing, but may be adapted for other block sizes. For these reasons the decomposition factors $m1 = 15$, $m2 = 16$, $m3 = 17$ [8] were chosen. This balances the processing delay through all stages in the pipeline. The product of the decomposition factors $m1 \times m2 \times m3$ equals 4080. This can be thought of as mapping the 4080 data points into a cubic data structure with sides of length 15, 16, and 17. The sides of the cube are the block lengths of the decomposed DFT. The entire DFT can then be computed by piping the output of one stage into the input of the next. Using the PFA we can rewrite the 4080 point DFT originally given as:

$$V_k = \sum_{i=0}^{4079} \omega^{ik} v^k \qquad (2\text{-}6)$$

into the following form by remapping the input and output data indices using the CRT.

$$V_k = \left[\sum_{i1=0}^{239} \left[\sum_{i2=0}^{271} \left[\sum_{i3=0}^{254} \omega^{i_1 k} v_k \right] \omega^{i_2 k} \right] \omega^{i_3 k} \right] \tag{2-7}$$

Now instead of doing one 4080 point transform, we are doing a 16 point DFT (15)(17) times, a 15 point DFT (16)(17) times and a 17 point DFT (15)(16) times. Taylor performed a numerical simulation of the 4080-point pipeline and the results showed that the best ordering of the DFT modules would be as shown above. The 16-point FFT has the best numerical performance, while the 17-point shows the worst. Ordering the pipeline in this fashion will minimize truncation and rounding noise [17].

The combined effect of the PFA and the CRT is shown figuratively in Figure 2-1. The CRT maps each element of the 4080 point data sequence into a unique address on a 15 x 16 x 17 cube. The 4080 point DFT is then computed as a sequence of three 2-D DFTs. For example, the 15 point DFT can be visualized as an array (16,17) of columns with 15 elements per column. This is represented by the XZ plane in Figure 2-1. Complete computation of the DFT will require computation of a DFT for each of the surface planes of the cube. The summation notation in equation (2-8) above reflects the DFT being computed and the number of iterations through the data set that are required.

Thus computation of the DFT is performed in a pipelined implementation as follows:

a). computation of all columns perpendicular to the XZ plane, map the outputs via the CRT into new location on the cube. (16 point DFT).

b). computation of all columns perpendicular to the YZ plane, map the outputs via the CRT into new location on the cube. (15 point DFT).

c). computation of all columns perpendicular to the XY plane, map the outputs via the CRT into new location on the cube. (17 point DFT).

This conceptualization leads directly into the pipelined architecture of the 1080 point DFT processor, shown in Figure 2-2. In hardware, the cube is a memory element of 1080 words with data addresses determined by the CRT and the array of columns represents

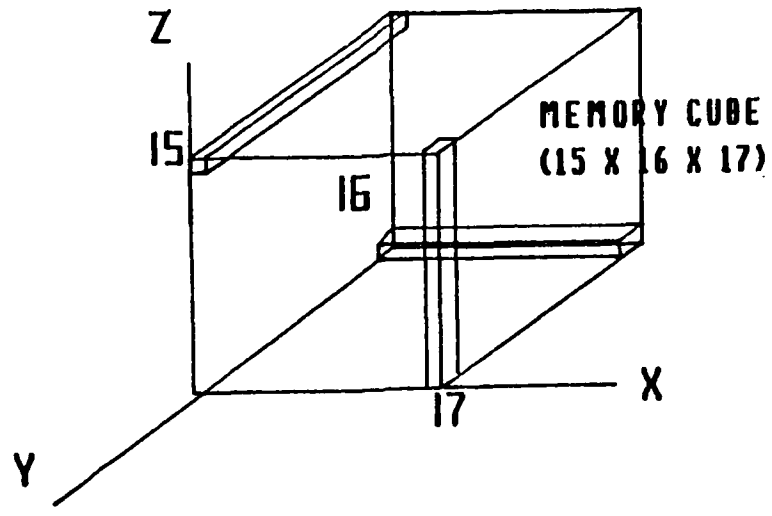


Figure 2-1. Cubic Data Structure of the 4080 Point PFA Implementation

a 15, 16, or 17 point WFTA processor element. Dual 4080 word memories are used between each WFTA element in order to allow each element exclusive access to a 4080 word data cube. After each element completes a scan through the data set the results are sent to the next memory element in the pipeline.

2.3.2. Winograd Fast Fourier Transform. Dr. Shmuel Winograd first introduced the Winograd Fast Fourier Transforms in 1975 [18]. Some of the characteristics of these algorithms are that the number of multiplications is nearly O(N) while the number of additions remain in the neighborhood of those required for other FFT algorithms. Winograd's algorithms are used to compute each 15, 16, and 17 point DFT. The small algorithms treat three cases of block size:

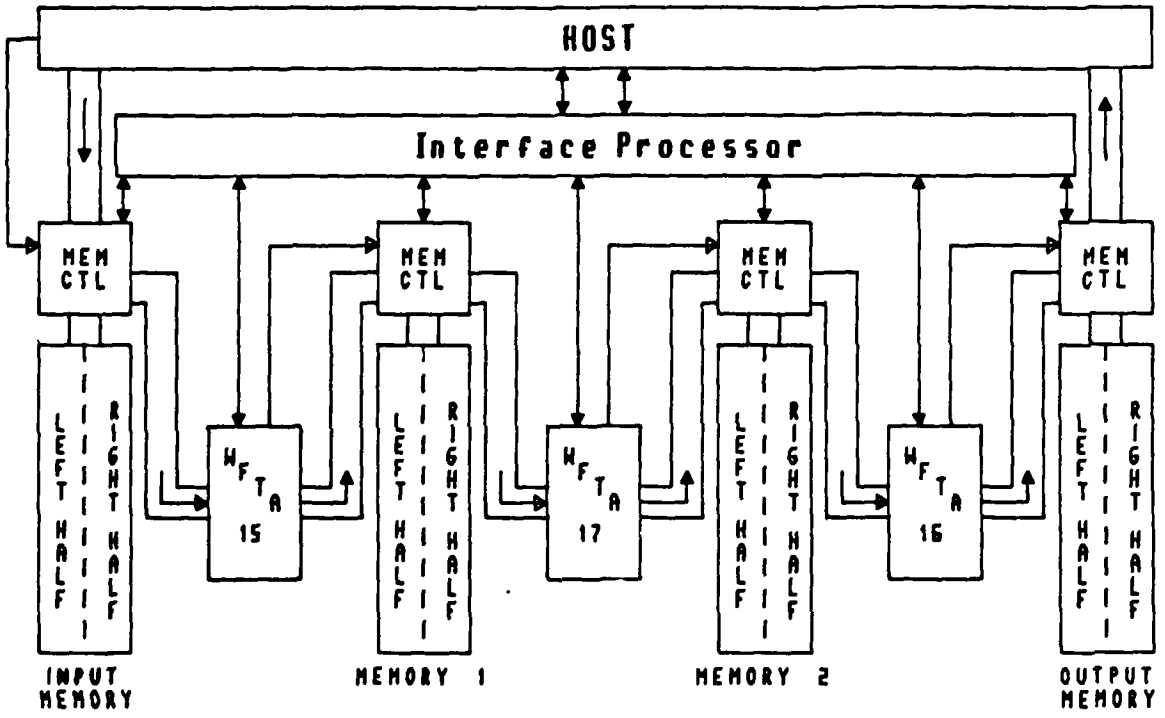


Figure 2-2. 4080-Point WFTA Pipeline Implementation

1. Blocklength a prime.
2. Blocklength a power of a prime.
3. Blocklength a power of two.

Cases one and t' ee, respectively, will be used to compute the 17 and 16 point DFT. The 15 point DFT does not fall under any of the cases listed above. In order to compute this DFT, and other blocklengths which are not one of the cases listed above, Winograd's large algorithm must be used. The large algorithm combines smaller blocklengths, which can be computed using the small algorithm, into a larger DFT module. In the case of the 15-point module it may be computed using blocklengths of sizes three and five, which are both case two.

2.4. WFTA Processor Architecture

The 16-point DFT, shown in (2-6), is computed using case three, blocklength a power of two.

$$V = \sum_{i=0}^{i=15} \omega^{ik} v_i \tag{2-8}$$

Convolution theory allows the DFT of a data sequence to be written as a cyclic convolution. Using the procedure for the second case, the 16 data points are partitioned into sets of even and odd indices. The eight odd indices are arranged into a set of four cyclic convolutions, while the eight even indices form an eight-point DFT, again a power of two. This partitioning process continues until the resulting DFT is composed of only two points which may be then directly converted into a cyclic convolution. The theoretical aspects of this process are covered in more detail in [1], [17]. The basic principle involved is that the DFT may be converted to a series of cyclic convolutions using the Winograd Algorithm. The rationale behind the conversion to a convolution is that the convolution may be calculated more efficiently using a fast convolution algorithm such as the Winograd Fast Convolution Algorithm.

The form of a cyclic convolution :

$$s(x) = g(x)d(x) \mod [\, m(x) \,] \tag{2-9}$$

where d(x) is the data sequence.
g(x) is the coefficient sequence.
m(x) is a fixed polynomial arising out of the partioning process.

Through an application of the Chineses Remainder Theorem for polynomials and some manipulations shown in detail in [1], [17], (2-9) may be converted into the form:

$$X = \mathbf{C}\,\mathbf{D}\,\mathbf{A}\,x \tag{2-10}$$

C is an incidence matrix of preadditions.
D is a diagonal matrix of coefficients.
A is an incidence matrix of postadditions.

The coefficient sequence is a diagonal matrix with constant either real or imaginary terms, the dimension of which is equal to the number of multiplications to be performed. The architecture that implements this equation in hardware is shown in Figure 2-3. The structure exploits the fact that the data is not complex until the postaddition matrix, where the paths merge in the final postaddition operation. This allows the architecture which implements the preaddition and multiplication operations to be separate real and imaginary parts. The arithmetic operations are performed using serial hardware to reduce routing space and complexity. However the I/O paths are word parallel in order to lessen the memory access time constraint. Additional structures needed are a control sequencer to generate control signals, and a ROM to store the data addresses in the order specified by the Chinese Remainder Theorem.

Winograd's large algorithm could have been used to compute the entire 4080 point DFT by nesting the 15, 16, and 17 point using the Winograd Large Algorithm, as in the case of the 15-point DFT. However, the size of the multiplication matrix limits the ability to embed an entire processor on a single silicon chip. For example, the 4080 point DFT would require a multiplication matrix over 23,000 serial multipliers tall [17]. A more modular implementation which is more suitable for VLSI implementation using state-of-the-art fabrication technology uses the Winograd modules to compute the 15-, 16-, and 17-point DFTs and the Good-Thomas Prime Factor Algorithm (PFA) to compute the entire transform. This implementation requires more operations but is more area efficient and lends itself to a pipelined implementation yielding greater computational throughput.

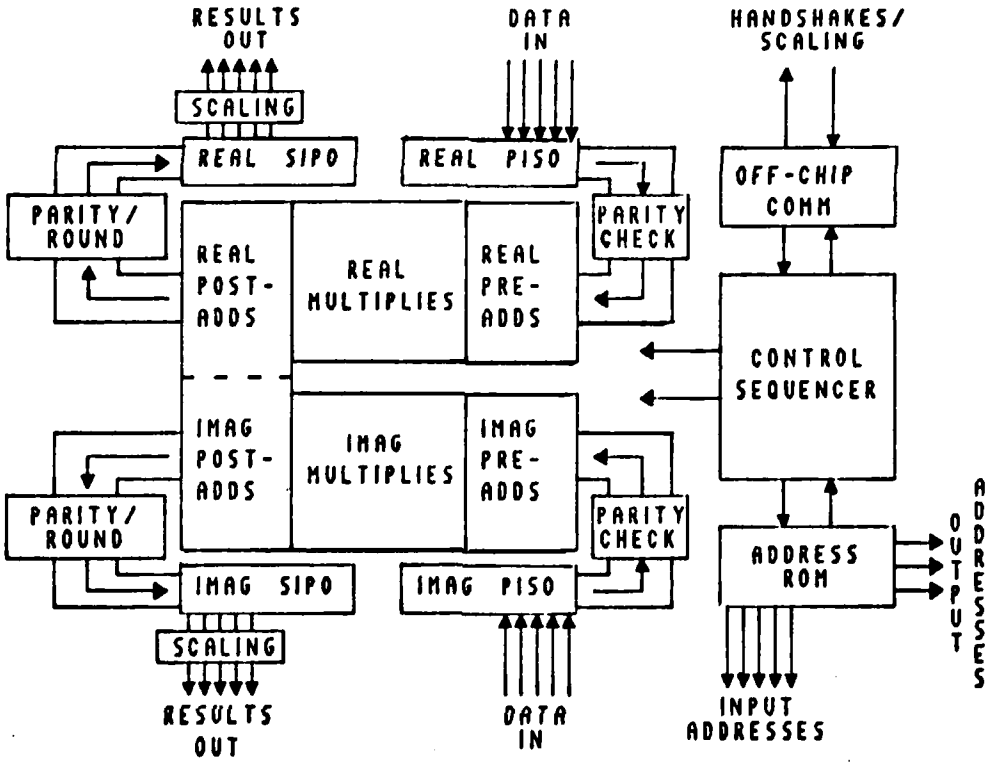


Figure 2-3. Winograd Processor Architecture

CHAPTER 3

VHSIC Hardware Description Language

3.1. Overview

The Winograd Fourier Transform (WFT16) processor presented in the last chapter is a complex circuit consisting of over 100,000 transistors. Complete comprehension of the function of every transistor would be impossible if the circuit was considered as a monolithic entity. An alternative approach might be to try and understand the function of the major components, and how they interact with the rest of the circuit. Detailed understanding would come by continually repeating this process, each time at a lower level, until the entire processor can be visualized as a grouping of simple circuits interacting in a complex manner. This is what VHDL modeling is all about. "VHDL is a language which can be used to describe hardware, ranging from simple logic gates to complex digital systems" [5]. The VHDL allows a circuit behavior to be described at a convenient level of understanding (or abstraction); more detail may be observed by stepping down one level in the hierarchy and describing the behavior of the components and the interactions which together create the larger behavior.

This chapter will present VHDL in the following context. A VLSI circuit of almost 29,000 transistors will be described, and its behavior modeled using the hierarchical procedure described above. Along the way, the syntax of VHDL will be presented as a tool useful in describing circuit behavior. The VHDL structures used for representation of a physical device will also be addressed. These include *entities* and *bodies*. This description will be followed with a representation of the constructs used to model data transforms such as sequential and concurrent signal assignment statements and bus resolution functions. Finally, a complete example of VHDL modeling of a CMOS latch will

be given.

3.2. VHDL Modeling of a Large Circuit

The output structure of the 16-point WFTA (WFT16) processor is a serial in, parallel out (SIPO) shift register. Every clock cycle one bit from each of 32 serial input vectors enters the register, and every other clock cycle a forty-eight bit vector is output in parallel to the data bus. This process is controlled by three signals. Thus, at the highest level of abstraction, the SIPO may be viewed as a black box which receives inputs and produces an output. By itself, this description does not impart very much information about how the SIPO operates. Referring to the system block diagram reveals that the thirty-two bit input is actually made up of two-sixteen bit vectors, and the output is two twenty-four bit words. This allows a second, lower level of behavior to be visualized: The SIPO is really composed of two smaller identical shift registers. Continuing in this fashion each register is found to consist of sixteen identical rows, each row made up of twenty-four identical cells. If the behavior of this one cell can be understood, it is easy to visualize the operation of the entire 29,000 transistor register. The decomposition of the SIPO behavior from one register into a cell is shown in Figure 3-1. Although not all circuits are an array of identical cells, most behaviors may be decomposed into a lower level of abstraction, which can then be more easily understood and modeled.

3.3. VHDL Modeling Structures

There are three independent units in VHDL: packages, entities, and bodies. A VHDL description of a piece of hardware consists of the interface (which is called an entity in the VHDL syntax) and an architectural description of how the device transforms inputs to outputs (a body in the syntax). Related type declarations, functions, and procedures can be grouped into a package and made available to the interface. VHDL defines two different types of information channels. Ports are the wires used to interconnect entities, while signals are used to carry information internal to a design

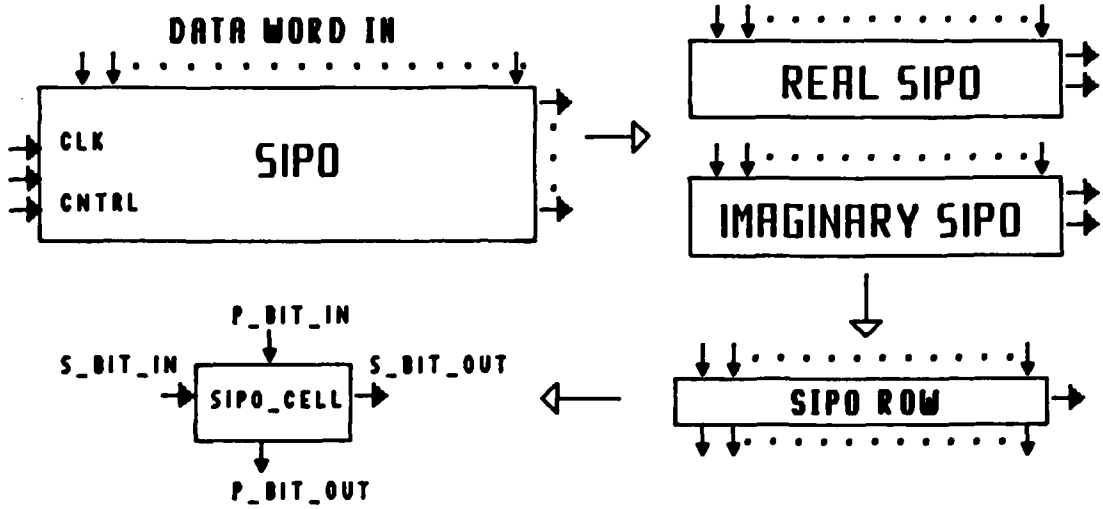


Figure 3-1. SIPO Decomposition

entity. Furthermore, ports are declared in interface entities, while signals are declared in bodies.

3.3.1. Packages. A package is an Ada-derived structure used to group logically related items so they can be referenced by a group of related design entities. Items which may be inside a package are type declarations, attribute declarations and specifications, constants, alias declarations, functions, and procedures. The contents of a package are made visible to interface declarations with a context clause:

for SOME_PACKAGE; use SOME_PACKAGE;

at the header of the entity declaration.

There are two kinds of types in VHDL, scalar and composite. Scalar types are single-valued such as:

a). Integer types
b). Floating point types
c). Enumeration Types
d). Physical types.

Enumeration types are declared by listing the values which objects of that type may have. A bit, which may take on the values '0' or '1', and boolean arguments which have the values of either 'true' or 'false', are examples of predefined enumeration types. Physical types represent physical parameters such as time, voltage, current, and so forth. Mathematical operations are defined on physical types. Composite types represent an array of values. Composites may be only of one type (such as an array of bit), or different types, such as a record listing the voltage and current requirements of a circuit.

VHDL also permits user-defined types. One example would define OPCODE as an array of eight bits. Then CPU instructions could be declared as being of type OPCODE. As another example, an enumeration type, TRI_STATE could be defined with a set of values ('0', '1', 'Z'). However, functions and operations on objects of user-defined types would have to be defined.

The SIPO would have various data types associated with the decomposition shown in Figure 3-1. The input and output at each level exhibit a certain word length which may be declared as a bit vector, a tristate data type is also needed, and the control and clock signals may form additional data types. Thus, a SIPO_PACKAGE would contain the following declarations:

```
Package SIPO_PACKAGE is

    type 16_bit_vector is bit_vector (15 downto 0);
    type 24_bit_vector is bit_vector (23 downto 0);
    type clk_signal is bit;
    type z_bit is ('0', '1', 'Z');
    type control is bit;

end SIPO_PACKAGE;
```

VHDL is a strongly typed language. Although control and clk_signal are both of type

bit, the compiler will flag assignments of clk_signals to signals declared of type bit, and vice versa.

3.3.2. Interface Declarations. The interface declaration defines the data paths (ports) over which information flows to and from a device. It consists of a list of port declarations and the direction and type of information which may flow through each port. To describe each level of decomposition of the SIPO shown in Figure 3-1, an interface description may be written. This description would contain a listing of all the inputs and outputs of the circuits. There is generally only one interface declared per design entity, but different implementations may reference the same interface. Items common to all bodies of that interface may also be included in the entity.

There are five port modes which are used to describe information flow across the interface boundary. Mode *in* is used for data entering a device from an external source. *In* ports may only be referenced, not changed, within that entity. However, *in* ports may be given a tie-off value in the interface declaration for use if that port is not connected to an external driver during a simulation. Mode *out* is used for data originating within a device for use in some external circuit. Its value, representing the result of an internal data transform, may not be used within the originating device. Mode *inout* is a bidirectional port which allows the port to be externally or internally driven (as in a system bus). Mode *buffer* allows the port to be referenced (read) by components both inside and outside the device boundary. However, it must be driven by a source within the entity defined by the interface. An example of *buffer* mode is the feedback inverter of a static latch. The mode *linkage* is used for ports whose direction of travel is unknown. This port mode is used only to pass information down to lower levels of the hierarchy. It cannot be either referenced or altered. Table 3-1 summarizes the port modes and allowed operations.

Port Mode		in	out	inout	buffer	linkage
Reading						
	inside	A	D	A	A	D
	outside	A	A	A	A	D
Writing						
	inside	D	R	R	R	D
	outside	A	A	A	D	D

(A) Allowed (D) Disallowed (R) Required

Table 3-1. Port modes [5].

An example will serve to illustrate several of these points. One cell of the output serial to parallel converter used in the WFT16 processor is shown in Figure 3-2.

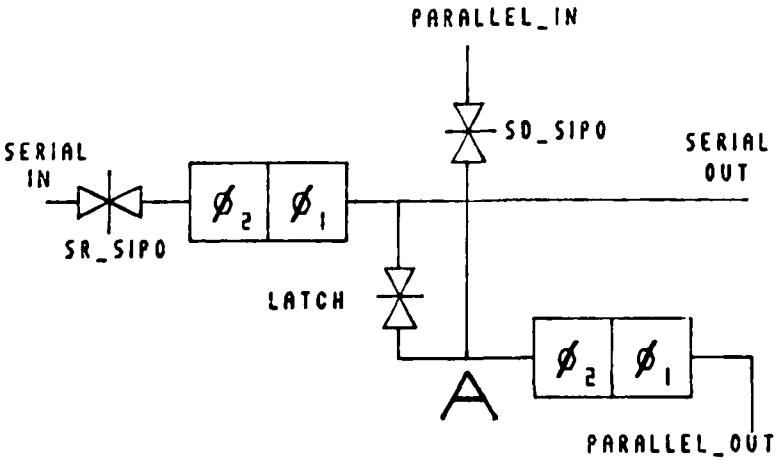


Figure 3-2. Serial to Parallel Output Cell

SERIAL_IN and PARALLEL_IN are the input data bits, SERIAL_OUT and PARALLEL_OUT are the output bits. SR_SIPO, SD_SIPO, and LATCH_SIPO are control signals. Circuit operation is under the control of a two-phase clock. The interface declaration for this cell is shown in Figure 3-3.

This example illustrates the use of port modes and user defined types. Types control and clk_signal are defined in SIPO_PACKAGE. The *in* ports are given tie-off values which will be used if the port is not connected in some simulation model. It is important to note what is happening at node A in Figure 3-2. First, the node is being driven by both the input port PARALLEL_IN, and the output port, SERIAL_OUT. Any node driven by more than one independent source is termed a "bus" in VHDL syntax. Busses must be declared by type and have an associated bus resolution function, which will determine how the several source values will be resolved to arrive at a signal value. Bus resolution functions will be discussed in section 3.5.3. SERIAL_OUT is also used to drive internal and external nodes, this must be reflected in the port mode. Both *inout* and *buffer* modes could be used. In this case, mode *buffer* was chosen to reflect that node A should not be driven from an external source. The assertion construct reflects the design intent that

```
with SIPO_PACKAGE; use SIPO_PACKAGE;
entity SIPO_CELL

(SERIAL_IN, PARALLEL_IN: in bit := '0';
 SR_SIPO, SD_SIPO, LATCH_SIPO: in control;
 CLK2, CLK2_NOT, CLK1, CLK1_NOT: in clk_signal := '0';
 SERIAL_OUT: buffer bit;
 PARALLEL_OUT: out bit) is

 assert (not(LATCH_SIPO and SD_SIPO))
  report "LATCH_SIPO AND SD_SIPO are both set"
 severity fatal;

end SIPO_CELL;
```

Figure 3-3. Interface Declaration

only one of the two drivers of node A should be active during any simulation cycle. If this assertion is violated the simulator will report the error message shown. The severity level is referenced at simulation time, along with a user specified error threshold, to determine which errors encountered should cause termination of the run.

3.3.3. Bodies. Many different approaches may be used to implement a given function. In general, however, the inputs and outputs to the function are fairly well defined early in the design cycle. VHDL supports design flexibility by allowing multiple architectural bodies to be written for a given interface. Each architectural body may be tested by linking the interface description to the body using a configuration block statement or configuration body.

The body defines how the device actually operates. There are two types of bodies, architectural and configuration. An architectural body describes the behavioral and/or structural characteristics of a particular implementation. A configuration body defines how a particular instantiation of an design entity is to be implemented. The configuration body is the linkage between the entity declared in the component declaration, the specific architectural body the designer wishes to use, and the component instantiation.

3.3.3.1. Architectural Bodies. The architectural body may consist of different levels which describe the operation of a device. A purely structural description exclusively uses component declarations and instantiations to describe its operation. On the other hand, a purely behavioral description contains no component instantiations. All data transforms are completely described using concurrent signal assignment or process statements inside the architecture. VHDL allows any combination of these two extremes to be used to model a device. The basic structure of an architectural body is shown in Figure 3-4.

```
architecture NAME_OF_BODY of NAME_OF_INTERFACE_DECLARATION is

 block_name: block (boolean guard statement)

   declarative section:
    component declaration;
    component configuration;
    local signal declaration;

 begin
  concurrent statements
    processes
    nested block_name;

end name_of_body;
```

Figure 3-4. Architectural Body Template

Note that these statements and declarations could be in any order within their respective regions.

The nature of digital systems is that their operation is made up of multitudes of circuits, all operating in parallel. The WFT16 processor is a pipelined, bit-serial processor which is designed to operate at high speeds, as such, there are few logic stages between clocked elements. Thus, the circuit could be roughly partitioned into just two sets of parallel operations: those which occur on the $\phi 2$ pulse and those which occur on the $\phi 1$ pulse. The block structure which makes up the architectural body shown in Figure 3-4 is the structure that VHDL uses to represent parallel events. All the statements contained within the block execute concurrently and may be controlled by the boolean guard. The guard expression specifies a condition which must be true before statements within the block which reference the guard can execute. This will be discussed further in the sections on signal assignment statements.

Before a lower level VHDL description can be referenced in a higher level description it must be declared. This is done by listing the interface entity name, followed by a listing of its port names, data types, and modes. A component declaration makes available a local copy of an interface to a body. To describe the behavior of the SIPO_CELL in terms of the behavior of its components, those components would first have to be declared as shown in Figure 3-5.

```
component MSFF
      port(A: in Z_bit;
      CLK2, CLK2_NOT, CLK1, CLK1_NOT: in clk_signal;
      B: buffer bit);

component T_GATE:
      port (X: in Z_bit;
      CLK: in clk_signal;
      Y: out bit);
```

Figure 3-5. Component Declaration Statement

Note the type clash between the input and outputs of the MSFF and T_GATE. The MSFF produces an output of type bit, and the T-GATE expects its inputs to be of type Z_bit. Since these are connected as per Figure 3-2, a type conversion function must be used to convert the output of the MSFF into the type that the T_GATE expects.

A component instantiation statement fits a declared component into the framework of the design. This is done by an interconnection of the ports of the instantiated component with ports declared within the interface and locally defined signals. A component instantiation is a concurrent construct and will be further discussed in section 3.5.2.

Information transfer within an architecture takes place using signals and ports. Ports, which are listed in the interface declaration for the register, are connected to the port with that same name in the component instantiation statement or signal

assignment statements. Signals are declared by name, by type, and by the reserved word *atomic* that indicates that multiple drivers are defined for that signal. As stated earlier, signals with multiple drivers are called busses. Signals are declared *atomic* followed by the name of a bus resolution function. *Atomic* is a flag indicating that multiple drivers are associated with a signal, and the name of the function to be used to resolve the drivers into an output value.

3.3.3.2. Configuration Blocks and Bodies. Since interface declarations can be associated with more than one implementation (body), it is necessary to identify which body is being declared. Identification of a component with a specific body can be done with a configuration specification within the body or by a separate configuration body. The disadvantage of placing the configuration within an architectural body is that the architecture is now specifically associated with one design. Flexibility to instantiate different components is lost unless the code is edited and recompiled. The more flexible approach would define a separate configuration body for each design. This would allow different configuration bodies to be written for different component instantiations within the same architectural framework.

A configuration specification assigns a specific body to be used with the interface in the component declaration. It may also specify port maps, additional ports, and generic declarations. The label used in a component instantiation statement identifies which instance of the component is being configured. Figure 3-6 shows the configuration of the MSFF and T_GATE used in the SIPO_CELL declared in Figure 3-5.

```
for M1, M2: MSFF
use
 entity (MSFF)
 port map (bit_in => A,
     CLK2 => CLK2, CLK2_NOT => CLK2_NOT,
     CLK1 => CLK1, CLK1_NOT => CLK1_NOT,
     bit_out =>B)
 body (MIXED_BODY)
end for;

for all: T_GATE
use
 entity (T_GATE)
 port map (bit_in => convb-z(X),
     clk => clk,
     bit_out => Y)
 body (BEHAVIOR)
end for;
```

Figure 3-6. Configuration Specification for SIPO_CELL

Note the use of the type conversion function, convb_z(X), in the port map for the T_GATE. Also note that since both instantiations use the same configuration, the instantiations labels, M1 and M2, could be replaced with the reserved word *all* as in the T_GATE configuration. If multiple configurations of the same entity are involved, all but one the same, the different one could be configured first, as above, and the rest identified with:

-- for others: MSFF use --

and configured in one block.

Entity (MSFF) identifies which interface entity is used. The entity entry links the component declaration to an entity which is stored in the VHDL design library. The port map associates the formal names listed in the entity with those used in the component declaration; association is left to right, formal => actual. The statement body

(MIXED_BODY) directs the use of the architecture (MIXED_BODY) for use in this architectural body.

The configuration body describes the implementation of any entities in an architectural body which are referenced by component instantiation statements. Design units referred to by the configuration body must reside in the design library. The configuration body is the top level of the hierarchy for the entity listed in the body header. The configuration specification is the main item which constitutes the body.

Configuration of component instantiations two levels deep is allowed in a configuration body. For example, if the SIPO_CELL is instantiated to build a SIPO_REGISTER_ROW a configuration body could configure the SIPO_CELL hierarchy down one additional level, allowing different MSFF implementations to be simulated within the framework of the SIPO_REGISTER.

3.4. Signals

VHDL uses signals to represent wire interconnections within a design entity, and ports to represent data channels to external devices. Input-to-output transforms in VHDL are represented by a future signal value and a time when it will become valid. This time/value pair is called a transaction. The time aspect could be represented by a delta delay or simulation time value. A delta delay is an infinitely small time unit, the sum of any number of which will never add up to any finite amount of physical time (in terms of circuit delays). Delta delay is used to represent events which must occur in response to other events without considering the nuances of their timing interaction. Simulation time represents real time, and is used to simulate timing dependencies between component and events.

The form of a simple signal assignment statement:

A <= B after Tns;

reads "the value of signal B is assigned to the driver of signal A and will possibly become the value of signal A after T nanoseconds". It is not possible to affect the current value of a signal only its future values. A simple example repeated from the VHDL tutorial will serve to illustrate this point.

"Consider the following pairs of statements:

A := B ;	X <= Y ;
B := A ;	Y <= X ;
Variable Assignment	Signal Assignment

In the case of variables A and B, after the two variable statements are executed, the values of A and B are identical after the two statements are executed. More interesting, however, is the fact that the values of X and Y will be swapped as soon as simulation time advances, because the current value of each signal has been scheduled to become the next value of the other signal's driver (after delta delay)." [5]

The value of a signal depends on the value of all of its drivers. (Some devices, such as node A of the SIPO_CELL, have multiple inputs to a single node. These multiply driven nodes are known as busses). When a signal assignment statement is executed it inserts a transaction into a signal driver. The signal driver can be thought of as a stack ordered with respect to time, time being the stack pointer. As time advances the value of the pointer will become simulation time. If the signal has only one driver, that driver's value will become the signal value at the time indicated by the stack pointer. Signals with multiple drivers have their values arbitrated via a bus resolution function which is usually written by the VHDL programmer. The bus resolution function is automatically invoked by the simulator.

A signal assignment statement creates a "projected output waveform" for a signal. Once the projected output waveform is put on the stack, but before it becomes a current driver value, it may be affected by signal assignment statements which execute at some point in the future. In other words, assignment to a node with only one signal driver does not automatically guarantee that, at some future point in time, the value of the assignment will become the signal value. The reserved word *transport* may appear in an

assignment such as:

A <= *transport* B after T ns;

Transport acts to delete transactions scheduled for times later than the first scheduled transaction in the new waveform. Inertial delay, the default case, simply deletes all transactions from the stack which are scheduled to occur before the first event in the new projected waveform. Figure 3-7 illustrates transactions and drivers. The stack structure represents a driver for the output of the combinational logic network. A projected signal waveform is shown which assumes that this is the only driver for the signal S. In the absence of any future signal assignments taking place between current time and the last time on the transaction stack, this will be the future signal waveform.

3.5. Signal Assignment Statements

VHDL supports two types of signal assignment statements, those which execute sequentially (sequential signal assignment) and those which execute simultaneously (concurrent signal assignment). Sequential statements, which must be nested inside a concurrent process statement, are an abstract means of describing I/O transforms, while concurrent statements lean more toward a specific hardware implementation.

3.5.1. Sequential Assignment Statements. An algorithmic approach to hardware modeling would use a sequence of calculations to map inputs into outputs. Sequential statements in VHDL are used for this purpose. They must be nested within a region known as a process. The process statement is itself a concurrent statement which may execute once per simulation cycle. When the process executes, however, each sequential statement will execute in turn. Each process executes in response to changes in signals currently enabled in its "sensitivity list". A sensitivity list identifies all the signals which can trigger a change in an output signal value. Every time a signal in the sensitivity list changes state, the process is activated and computes a new projected output waveform.

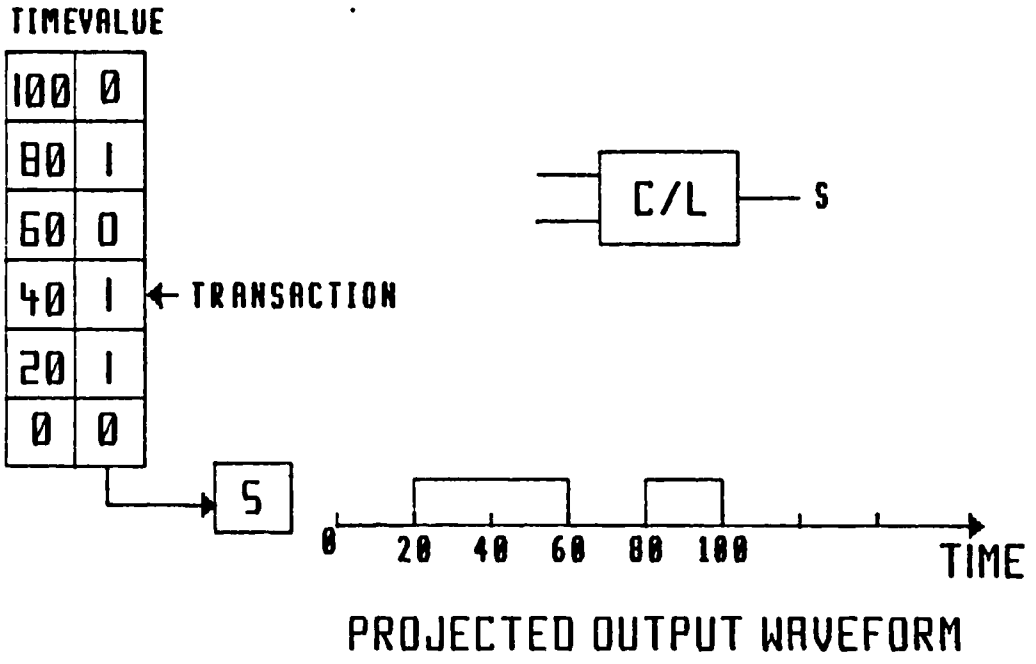


Figure 3-7. Signals and Drivers

The sensitivity list provides a means to improve the execution time of a simulation.

Consider simulation of a D-latch, the primary cell structure for most components within the WFT16 processor. The latch has one data bit input and two clock senses for input signals. Since any of these three signals can affect the output, they all will be listed in the sensitivity list. However, the output changes only in response to changes in the input signal. Needless event scheduling can be avoided by activating the process only when the input has changed, not just because the clock "ticked." Enable and disable statements are used to achieve this purpose. All three signals must be listed in the sensitivity list, but the *enable* statement may be used to enable sensitivity to clock transitions only if the input has just changed state. While the input remains stable, the

disable statement will make the latch process insensitive to clock transitions.

Another feature of processes is that they may use variables and constants to compute a value. Since variable assignment occurs immediately, (instead of at some point in the future, as in the case of signals), an arbitrarily complex algorithm may be used to compute a value and assign it to a signal node. This feature can be used to model propagation through layers of combinational logic within one simulation cycle. Using delta delay for signal assignment statements will cause the signal to take one simulation cycle propagation delay per logic stage. If variable assignment statements are used, propagation delays through gates will not be a factor and delta delay simulation can still be used to simulate clocked stages. For computing variables, VHDL supports most of the control statements used in programming languages such as *loops, case statements, if .. then .. else, and, for.* These control constructs may also be used to assign a signal value to a target based on a the value of a variable.

3.5.2. Concurrent Signal Assignment Statements. "Concurrent statements allow the user to specify the structural characteristics of a design, and to describe its behavioral characteristics in terms of concurrently executing, sequential processes" [5]. Concurrent statements represent hardware components which operate in parallel upon receipt of some control signal or clock pulse.

The *block* statement defines a region of text and a guard statement which can affect execution of processes within that block. Blocks are delineated by:

block (optional guard) .... end block;

statements. The *guard* is a boolean expression which is referenced by concurrent statements using the reserved word *memoried.* Memoried statements fire only when the guard expression is true, if the guard is false, changes to the signals will not cause output transitions. Block statements group together statements which execute in parallel (on the same clock pulse for instance). Processes can import the guard value by inserting the guard into their sensitivity lists. Signals can be enabled or disabled depending

on the guard value. If the guard is omitted, its value defaults to true.

The scope of the guard is only within the nearest enclosing block statements. They can be nested but this must be done explicitly. Blocked statements consist of declarative and executable parts. The architectural body is an example of a blocked structure. Component and signal declarations follow the block label. *Begin* signals the start of the executable part. Component instantiations, conditional signal assignments and processes fall within this section. The *end block;* signifies the end of the scope of the guard statement, but it may be imported to nested blocks by declaring a port for the guard and assigning the value of the port to that guard.

Component instantiation statements make use of a unit defined with a component declaration by listing the signals which are to be connected to the ports named in the declarations. Ports can be assigned by name association, by positional association, or a combination of both. Name association is an explicit linkage of the port and the name declared in the component declaration. Positional association is implicit, local signals are identified by their position in the instantiation list with respect to the ports listed in the component declaration. If a combination of the two methods are used, all named associations must occur first.

A purely structural description of the SIPO_CELL could be written by instantiating the MSFFs and T_GATEs and connecting them through their port lists:

```
T1: T_GATE port(SERIAL_IN, SR_SIPO, S_IN);
M1: MSFF port(S_IN, CLK2, CLK2_NOT, CLK1, CLK1_NOT, SERIAL_OUT);
T2: T_GATE port(PARALLEL_IN, SD_SIPO, P_IN);
T3: T_GATE port(SERIAL_OUT, LATCH_SIPO, P_IN);
M2: MSFF port(P_IN, CLK2, CLK2_NOT, CLK1, CLK1_NOT, PARALLEL_OUT);
```

These statements will execute whenever one of the signals listed in the port list changes. This method of modeling provides a great deal of information about the device interconnections, but not much on its' operation. There are other concurrent statements which can be used to impart a little more information about the behavior of the device. Since

the transmission gates are mainly used to control the inputs to the MSFFs, the clarity of the description may be improved by using a conditional signal assignment statement. The three transmission gate instantiations will be replaced as shown below.

```
S_IN <= SERIAL_IN when SR_SIPO = '1'
   else 'Z';

P-IN <= PARALLEL_IN when SD_SIPO = '1'
   else
        SERIAL_OUT when LATCH_SIPO = '1'
   else 'Z';
```

A SIPO_REGISTER_ROW could be constructed using twenty-four instantiations of the SIPO_CELL. This would be very cumbersome method to model a regular array of cells. VHDL provides a more efficient way through the generate statement. The VHDL model for a SIPO_REGISTER_ROW is shown in Figure 3-8 below.

```
for i in (23 downto 0) generate
 if i = 23 generate
  SIPO(23): SIPO_CELL
   port(SERIAL_IN, PARALLEL_IN(i), CLK2, CLK2_NOT, CLK1,
     CLK1_NOT, SERIAL_OUT(i), PARALLEL_OUT(i));
 end generate;

 if i < 23 generate
  SIPO(i): SIPO_CELL
   port(SERIAL_OUT(i+1), PARALLEL_IN(i), CLK2, CLK2_NOT, CLK1,
     CLK1_NOT, SERIAL_OUT(i), PARALLEL_OUT(i));
 end generate;

end generate;
```

Figure 3-8. SIPO_CELL_ROW Model

3.5.3. Bus Resolution Functions. Each concurrent statement that assigns to a node creates a separate driver for that node. The signal cannot update its current value without considering the values of all the drivers, these signals are said to be *atomic*. No changes may be made to the value of a signal without considering the values of all the drivers. Bus type signals are declared using the reserved word *atomic* followed by the name of the bus resolution function, and the data type:

atomic **BUS_RESOLUTION_FUNCTION_NAME data type;**

Other data types may also be *atomic*, this simply means that the elements of an object, such as a record or bit vector type, are inseparable. Assignment cannot be made to any one element individually, all elements must be updated in parallel. If the programmer tries to update a single element an error will be flagged.

Bus resolution is the means by which multiple drivers are resolved into a single value. The function is defined by the user and invoked by the compiler each time a new driver value rises to the top of the stack. One nice feature is that there is no defined number of nodes per *atomic* signal. Additional components may be hung on the bus simply by assigning to that signal name. The function is implicitly called during simulation, its argument list is an unconstrained array of that signal type. An example of a bus resolution function for tristate signals is shown in section 3.4.

An explicit function call can also be used to perform bus resolution type behavior. The function call would contain a listing of the signals, both control and data, which could affect a node, and return the value of the future signal driver. Bus resolution via a function call is used in the LATCH example in the next section.

3.6. CMOS Latch Example

The latch is the building block for all clocked elements within the processor. A clocked CMOS latch, shown in Figure 3-9, will demonstrate how VHDL is used to model hardware. The box surrounding the latch represent the distinction between entities and

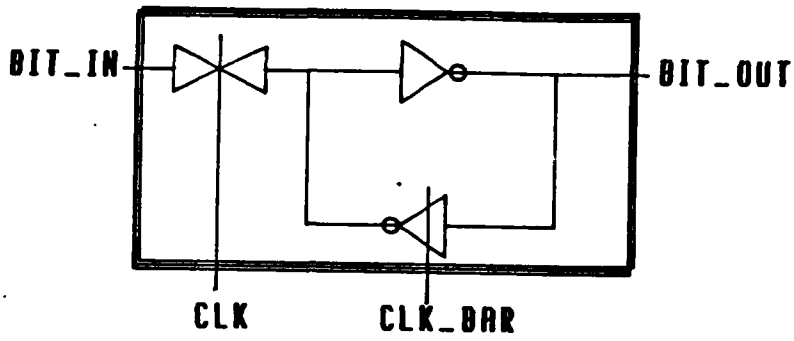


Figure 3-9. Clocked CMOS Latch

bodies. Entities describe the interface of the latch to external circuitry, and bodies describe how the internal hardware performs. The Latch interface consists of the signal lines IN, OUT, CLK, and CLK_BAR. Therefore, a simple VHDL interface description can be written as shown below.

```
with Latch_package; use Latch_package;
entity Latch
 (BIT_IN: in Z_bit;
          BIT_OUT: buffer Z_BIT;
          CLK, CLK_BAR: in CLK_SIGNAL := '0') is
        end LATCH;
```

In this example "BIT_IN" is a signal driven by a source external to the Latch. Its value will be used by the device, but it may not be changed within the boundaries of the latch, port mode *in* is the read-only mode. The port BIT_OUT is the output signal and also the signal source for the feedback loop. The port mode *buffer* is used because it requires the signal source be interior to the body, but also allows the value to be referenced within the body. Since it may not be driven by a source external to the body, it is read-only for outputs. The clock signals, CLK, and CLK_BAR are of type CLK_SIGNAL.

This is a user defined type and the context clause (with .. use ..) implies that it has been defined in the package Latch_package. Node A illustrates the requirement for a bus resolution function. This node may be driven by two separate components, the T- gate and the clocked inverter. A detailed architecture description is shown below.

```
architecture one_description of Latch is

 description_blk:
 block

 -- The double dash is the comment delimiter in VHDL.
 -- The component declarations provide a copy of the device for
 -- use within the body.

component T_gate port(A: in Z_bit; B: out Z_bit; C: in bit);
component Inverter port(C: in Z_bit; D: out bit);
component TRI_STATE_INVERTER port(A: in Z_bit; B: out Z_bit; C: in
          bit);


-- This is a "block configuration" for the T_gate used within this
-- description. The use is a binding indication which ties
-- together the predefined device T_GATE_INTERFACE to the label T1
-- in the component instantiation statements. The ports listed in
-- the T_gate entity description are tied to those listed within the
-- component statement above. Finally, the body identifies a
-- particular architectural body to be used with the entity. The
-- other components are configured in a similar manner.

for T1: T_gate use
 entity (T_GATE_INTERFACE)
 port map (T_gate_in => A; T_gate_out => B; Control => C)
 body (a_behavior);
end for;


signal A: atomic LATCH_RESOLVE Z_BIT;
          -- LATCH_RESOLVE is the bus resolution
          -- function which will be used to
          -- determine the value of node A.
signal tmp: bit;

       begin

        A <= IN when CLK = '1';
```

```
OUT <= B;

block (CLK = '0' and not OUT'stable)

        -- this is the guard statement associated
        -- with this block;
 A <= memoried tmp;


-- This signal assignment statement will only execute when the guard
-- is true. Note two assignments to node A. If there were other
-- statements within this block that did not use the word memoried,
-- they would execute regardless of the value of the guard.
-- Event scheduling could be minimized
-- by guarding the input node with the boolean expression (not
-- IN'stable) and using a memoried signal assignment statement to
-- signal A. The output of the latch will remain the same unless the
-- input changes, without requiring event scheduling on every clock
-- transition.

 end block;


 T1: T_GATE port (IN, A, CLK);
 I1: INVERTER port (A, B);
 TRI1: TRI_STATE_INVERTER port(B, tmp, CLK_BAR);

end block description_block;

end one_description;
```

Declaring A to be *atomic* tells the compiler that that node is driven by more than one source and the function LATCH_RESOLVE will be used to determine its value. The function is located within the package Latch_package as shown below. Once the latch is built and tested it may be declared in the same manner as t' T_gate in this example.

```
package Latch_package is

  type CLK_SIGNALS is (CLK, CLK_BAR);
type CONTROL_SIGNAL is (RST, SH_RIGHT, LOAD);
type Z_bit is ('0', '1', 'Z');

function R_LATCH_RESOLVE (RST, BIT_IN, L_BIT: Z_bit)
        return Z_bit is

  constant RST_SIGNAL :Z_bit:=0;  -- This assigns a value of '0'
                                  -- to RST_SIGNAL.

    if (RST >= 1) then
      return RST_SIGNAL;
    elsif BIT_IN >= ('0' or '1') then
        return BIT_IN;
      else
        return L_BIT;
    end if;

  end R_LATCH_RESOLVE;

  function LATCH_RESOLVE (array<> of Z_BIT)
          return Z_bit is

     for I in input'low to input'high loop
       if input(I) /= 'Z' then
         output := input(I);
         exit;
       end if;
     end loop;

    return output;
  end LATCH_RESOLVE;

end Latch_package;
```

The function R_LATCH_RESOLVE is a function which would be called to resolve the inverter input value to a circuit as shown in Figure 3-10. The conditional signal assignment calling the function to return the output value would be written as follows:

```
A <= R_LATCH_RESOLVE(BIT_IN, RST, INVERT_OUT);
```

Note from the circuit diagram that the RST is being implemented in a behavioral

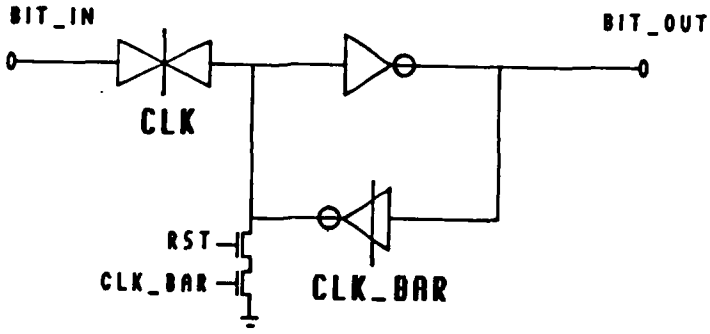


Figure 3-10. Resettable CMOS Latch

fashion, rather than a description of the circuit (structural) implementation.

This is a very detailed, in-depth description of a latch. A much more concise, compact VHDL description can be written which will execute much more efficiently. This description will model the function that the latch performs, rather than the subcomponents which implement that function. Behavioral descriptions will focus entirely on function at the expense of detail. An alternate latch description is shown below

```
architecture BEHAVIOR of LATCH is
 block( not bit_in'stable)
  begin
   process( guard, clk, bit_in)
    begin
     if (guard and bit_in /= Z) then
      enable clk;
     else disable clk;
     end if;
     if (clk = '1') then
      bit_out <= not bit_in;
     end if;
   end process
  end block;
end BEHAVIOR;
```

This description uses a process statement to assign the inverted input to the output. Several conditions are placed upon this assignment in order to minimize the number of transactions placed in the driver for signal bit_out. In general, we wish to avoid scheduling unless the new output value is different from the previous one. In order for the output to get assigned a value, these conditions must be met: 1). the input value must have changed. 2). the input must not be the high impedance value, and 3). the clock must be high. This description models the same function as the preceding example, but it eliminates transactions caused by the _not transitions, and it only executes if it will cause a different output to be put in the driver.

3.7. Complete SIPO Modeling

Finally we are in a position to do a complete SIPO description in VHDL. This section will pull together the previous examples, as well as incorporate the principles used in the last latch example, that of trying to avoid unnecessary CPU overhead caused by redundant event scheduling. The methodology used in this section will parallel the methodology used to do the complete WFT16 modeling.

In general, we wish to model and simulate circuits at a level of detail sufficient to observe the functionality and operability of the unit cell, but not to the level of every

signal node switching as the clock "ticks".

The SIPO was decomposed into a single cell, the SIPO_CELL, in section 3-1. Figure 3-2 shows that the primary components are MSFFs and T_gates. As stated earlier, transmission gates (T_gates) are primarily used to gate inputs with control signals or clock pulses. Thus, we shall model them behaviorally with conditional signal assignment statements. The MSFFs on the other hand, are built from two latches of the type modeled in the previous section. Instead of building a MSFF from two instantiations of a latch, we shall use the same principles to model the MSFF behaviorally.

```
architecture behavior of MSFF is
block
 signal mid : bit;
 begin
  block(not bit_in'stable)
  begin
   process(guard, bit_in, clk2)
   begin
    if (guard and bit_in  = 'Z')
     enable clk2;
    else
       disabl  clk2;
    end if;
    if (clk2 = '1') then
     mid  <=  not convz_b(bit_in);
    end if
   end process
  end block

  block (not mid'stable)
  begin
   process(guard, mid, clk1)
   begin
    if (guard)
     enable clk1;
    else
       disable clk1;
    end if;
    if (clk1 = '1') then
     bit_out <= not convb_z(mid);
    end if;
   end process;
  end block;
end block;
end behavior;
```

Using this description as a building block, we may now efficiently model the SIPO_CELL.

```
architecture MIXED of SIPO_CELL is

block
  component MSFF port (a: in Z_bit; CLK2, CLK1: in clk_signal,
                z: out bit);
-- defer configuration of MSFF, do in a separate configuration body.

  begin

    process(shift_right, ser_in)
    begin
      if (((ser_in'stable nor (ser_in = 'z'))
      and shift_right = '1') then
          from_ser <= ser_in;
      end if;
  end process;

    process (parallel_in, ser_out, latch, shift_down)
    begin
      if (shift_down nor latch) then
        disable ser_out, parallel_in;
      else enable ser_out, parallel_in;
      end if;
      if ((shift_down = '1') and (not parallel_in)) then
        to_parallel <= parallel_in;
      elsif ((latch = '1') and not (ser_out'stable)) then
        to_parallel <= ser_out;
      end if;
    end process;

    ff_ser: MSFF port(from_ser, CLK2, CLK1, ser_out);
    ff_par: MSFF port(to_parallel, CLK2, CLK1, p_out);

  end block;
end MIXED;
```

Using this mixed description of the SIPO_CELL the SIPO may be described as an array of these cells. Modeling the SIPO as an array [16][24] of SIPO_CELLs must be done in two steps. First, construct a [1][24] row of cells, and then use this row (instantiate) sixteen times to build an array [16][1] of rows. The SIPO_CELL is configured at the row level. Since our goal was to observe the WFT16 at the functional level, the SIPO_CELL is the highest level at which we will attempt to model things behaviorally. Above this level, things will be modeled at a purely structural level. It is possible to use the

hierarchical modeling facilities of VHDL to model behaviorally at much higher levels, but that will not be done here. The interface description for the SIPO_ROW is shown below.

```
with SIPO_PACKAGE; use SIPO_PACKAGE;
entity SIPO_ROW

 (BIT_IN: in z_bit;
  WORD_IN: in 24_bit_vector;
  CLK2, CLK2_NOT, CLK1, CLK1_NOT: in clk_signals := '0';
  SHIFT_RIGHT, SHIFT_DOWN, LATCH: in control;
  WORD_OUT: out 24_bit_vector)) is

end SIPO_ROW;

architecture Structure of SIPO_ROW is
block
signal SERIAL_INT: bit_vector(22 downto 0);
begin


 for i in (23 downto 0) generate
  if i = 23 generate
   SIPO(23): SIPO_CELL
   port(SERIAL_IN, PARALLEL_IN(i), CLK2, CLK2_NOT, CLK1,
       CLK1_NOT, SERIAL_INT(i), PARALLEL_OUT(i));
  end generate;

  if (( i < 23 ) and ( i > 0 )) generate
   SIPO(i): SIPO_CELL
   port(SERIAL_INT(i+1),  PARALLEL_IN(i), CLK2, CLK2_NOT, CLK1,
       CLK1_NOT, SERIAL_INT(i+1), PARALLEL_OUT(i));
  end generate;

  if i = 0 generate
  SIPO(0): SIPO_CELL
   port(SERIAL_INT(i+1), PARALLEL_IN(i), CLK2, CLK2_NOT, CLK1,
       CLK1_NOT, PARALLEL_OUT(i));
  end generate;

 end generate;
end block;
end structure;
```

The final step, generation of the entire SIPO array as a set of SIPO_ROWs, is similar to the generation of SIPO_ROW. A special body, SIPO_TOP, will be used as the topmost row in the array. Signal declarations are also required for the parallel inputs

and outputs of the rows internal to the structure.

Modeling the SIPO as an array of cells requires several steps, and different architectural bodies and interfaces. The top level interface description, shown below, is the final product, so far as the rest of the circuit is concerned, of the description process. The detail shown in the circuit modeling is buried in the input-output transform of the SIPO.

```
entity SIPO
( WORD_IN: in 24_bit_vector;
  SERIAL_OUT :buffer 16_bit_vector;
  CLK2, CLK2_NOT, CLK1, CLK1_NOT: in clk_signal := '0';
  SR_SIPO, SD_SIPO, LATCH_SIPO: in control; ) is
end SIPO;
```

The architecture could just as easily (actually much more easily), have been modeled behaviorally. As long as the output bit stream from both simulations look the same, the level of detail of the description is irrelevant.

CHAPTER 4

VHDL Modeling

4.1. Overview

This chapter will present the structural decomposition of the 16-point WFTA (WFT16) processor leading to the VHDL modeling of its primary circuit components. A top down decomposition will impose a signal flow on the system which can be used to define the VHDL interface entity. Once the processor is decomposed into its primary cell structures, a hierarchical description of the chip will be facilitated by a bottom-up cell description.

4.2. 16-Point WFTA Processor

The 16-point Winograd algorithm was discussed in Chapter 2. The basic architecture for all of the Winograd processors consists of input/output registers, arithmetic circuitry, special cells for parity and rounding operations, address storage ROMs, and a control sequencer. Primary differences in the actual implementation of the different processors result from different numbers of arithmetic operations in the pre-, post addition array, and the height of the serial multiplier array. The desire to balance the latency between all the processors in the pipeline would require different data word lengths to compensate for the different array sizes.

4.3. Operation.

The processor architecture is a pipelined bit-serial machine. The major processing blocks: input and output registers, preadders, multipliers, postadders, and parity circuitry form the first level of decomposition. Figure 4-1 shows this level of decomposition for the WFT16 processor superimposed over a signal flow graph.

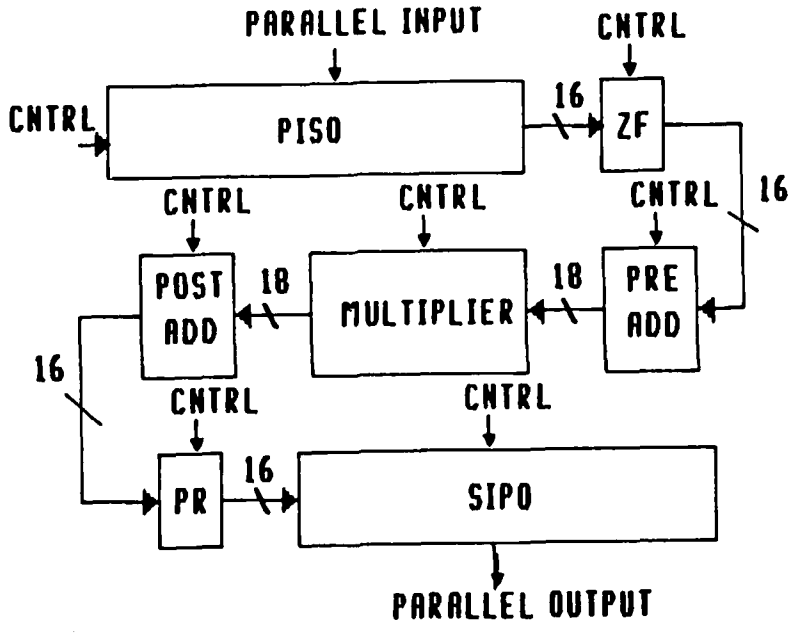


Figure 4-1. Decomposition and Signal Flow of the WFT16 Architecture

The processor can be divided into two separate identical sections, real and imaginary. These sections are independent through the last column of the post adders. In this column, real and imaginary data is added/subtracted to form the complex outputs. Since the two sides are mirror images, only one side will discussed and modeled, with the understanding that the other side performs exactly the same operations, in the same sequence, only with a different set of data.

The input register is a parallel-in, serial-out (PISO) register, twenty four bits wide by sixteen words deep. Input data is twenty-three bits of data and one parity bit. Every other clock cycle the PISO gets a new word from one of the two off-chip input memories, using an address from the XROM. The signal SD_PISO is used to shift the pre-existing words in the parallel portion down one level. After thirty-two clock cycles, the PISO is full and the signal LATCH_PISO goes high to transfer the word-parallel data into the

serial shift register. This empties the parallel portion of the PISO, and the cycle repeats itself for as long as the operate flag remains high. Words in the serial register will be shifted into the parity check and zero fill cell (PC/ZF), least significant bit first, one bit per word, while the signal SR_PISO is high. To allow for numerical growth through the arithmetic pipeline, the data is extended to a thirty-two bit word length in the PC/ZF cell. Parity is checked and the parity bit stripped in this cell. The PC/ZF cell has inserted one half clock cycle delay. Some zeroes are inserted prior to the LSB to scale up the data to enhance the signal to noise ratio. Sign extensions are appended after the MSB in order to prevent arithmetic overflow. The reader is referred to [17] and [4] for more information.

The number of zero fills and sign extensions are determined by the adaptive scaling algorithm which takes into account the relative magnitude of the input data. Each 4080 point data set is associated with a scale factor which reflects the magnitude of the largest number in the input data set. The scale factor is the smallest number of sign extensions of any number in the set. To avoid overflow, the largest number (scale factor 0) requires five sign extensions. Data sets composed of smaller numbers can replace unneeded sign extensions by zeroes to enhance numerical performance.

The arithmetic section actually implements the Winograd Fourier Transform. To generate the multiplicand from the output of the PC/ZF up to four sequential addition/subtraction operations may be needed. Multiplicands generated in less than four operations remain aligned with the other elements in the bit-vector through the adder/subtractor columns by replacing the one-delay wide A/S cells with MSFFs. Most circuit components in the WFT16 have an input $\phi 2$ latch and an output $\phi 1$ latch. Exceptions to this rule are the PC/ZF which is a $\phi 2$ latch preceded by some combinational logic, and the adder subtractors (A/S). The A/S are reversed, data enters through a $\phi 1$ latch, and leaves through a $\phi 2$ latch. To balance the pipeline with an equal number of $\phi 2$, and $\phi 1$ latches, some extra latches are put at each end of the adder arrays. A pipeline view of the preadd section is shown in Figure 4-2.

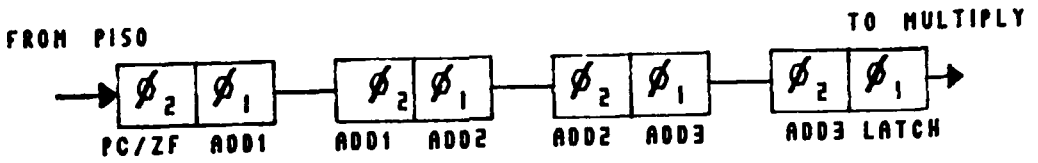


Figure 4-2. Preadd Pipeline

The fourth column operation only involves two data words, the result of which is the DC components of the Fourier transform. This is the last arithmetic operation to be performed on these two bit streams. They will travel through the rest of the pipeline through delay MSFFs. Since the sum and difference of this operation pass through a trivial multiplier ($\times 1$), the last adder/subtractor (A/S) column can be eliminated and the sum/difference of these two terms computed in the first column of the postadd array. This will reduce pipeline latency one clock cycle and eliminate thirty-five MSFFs. There is a one clock cycle latency through each column of the preadders, thus the preadd section of the WFT16 introduces four cycles latency into the pipeline.

The multiplier array consists of an array [18][14] of multiplier cells. The 28 bit Winograd coefficients are encoded into fourteen cells using Booth's quaternary encoding algorithm. Each bit of the reduced coefficient represents one bit of the serial multiplier. Since each multiplier cell requires three delay stages, there are a total of forty-two cycles of latency through the multiplier.

The postadder, like the preadder, requires three columns of adders. In column one, the add operation, deferred when the fourth column of the preadd array was eliminated, is performed. Data is either real or imaginary through the first two columns of the post adder. In the third column the two streams are mixed, resulting in complex outputs. The next stage is the parity generation, arithmetic rounding cell. At this point the

thirty-two bit results carried through the arithmetic pipeline are rounded down to twenty-three bits. The PR cell calculates odd parity on these twenty-three bits which is then appended to make a twenty-four bit word. The diagram of the postadd portion of the pipeline is shown in Figure 4-3. The output leads into the serial in, parallel out (SIPO) register. The SIPO has the same organization as the PISO, only the data enters bit serial, and leaves word parallel.

After the MSB (which is the parity bit) has entered the SIPO the signal LATCH_SIPO rises and drops the bits into the parallel portion of the SIPO. Every other clock cycle the complex output is sent to the output RAM, the memory address again are supplied by the XROM.

4.4. Processor Decomposition

Any one section of the processor is continually operating on a one bit slice of a thirty-two bit vector. Latency through the pipeline is 119 clock cycles, but once a word enters the PISO it is associated with fifteen other bits in the same position in their respective data words. This alignment is maintained throughout the pipeline.

The WFT16 processor can be decomposed into parallel columns of functional computation units. The height of the column would represent the number of bit streams (or wires) crossing the interface. The second level of decomposition is shown in Figure 4-4. VHDL interface descriptions could be written to cover the number of bits coming across

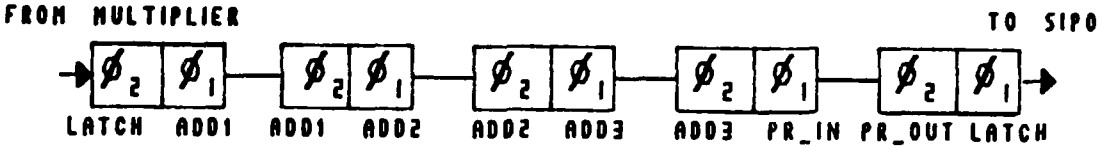


Figure 4-3. Postadd Pipeline

the interfaces from each column, and the control signals required per column for this level of decomposition. The next level of decomposition will break the columns into their constituent processing elements. These processing elements are the primary building blocks for the WFTA processor. A final decomposition will tear these cells down into flip flops, latches, transmission gates, and inverters. Each column in Figure 4-4 is built by stacking a number of primary circuit components. The primary circuit components for the WFT16 are the PISO_CELL, the MSFF, the A/S, the five multiplier cells, the PARITY ROUND CELL, and the SIPO_CELL. For the purposes of functionally simulating the entire circuit, these cells will be the highest level where behavioral constructs will be used. Above this level, at the column or block level, the descriptions will be purely structural. The VHDL descriptions of these cells are given in Appendix 1. In addition to The latch described in Chapter 3, the lower level subcomponents, which can be used to structurely model the primary cells, are also located in Appendix 1.

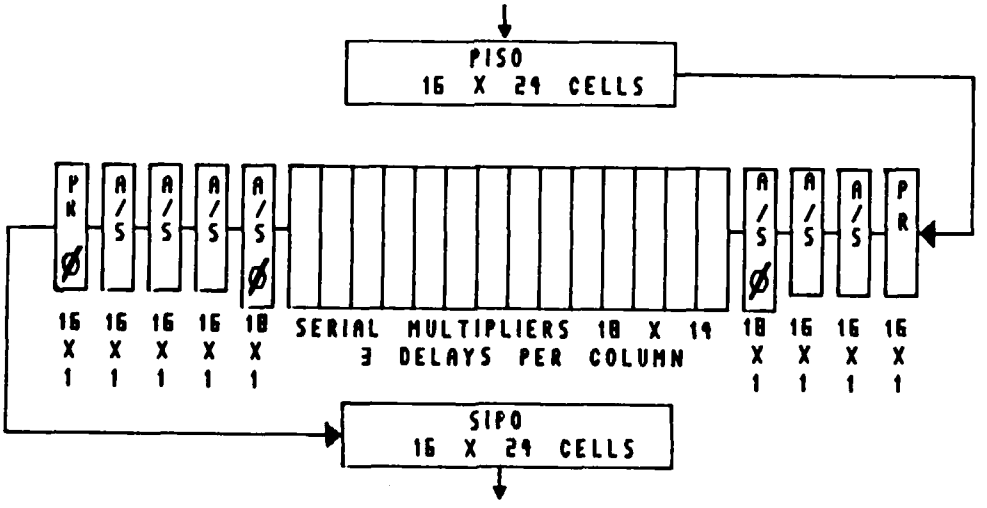


Figure 4-4. Column Form of WFT16 Processor

These subcomponents are the transmission gate (T_gate), the Z_inverter, and the Set Reset Flip Flop (SRFF).

CHAPTER 5

WFT16 Simulation Program

5.1. Overview

The VHDL is being developed to model and simulate the VLSI and VHSIC circuits currently being designed for future defense needs. One of the needs that it is intended to fill, that of design verification, is one that the WFTA design group currently requires. Current simulation tools, such as N.2, and N.mpc are not suitable to simulate designs such as the WFT16 processor at the functional level. Furthermore, the run times of of logic level simulators, such as RNL, become excessive as the size of the circuit increases. The characteristics of the WFT16 architecture make it amenable to modeling and simulation using another approach, that of using a high order language with the necessary bit-level operators to develop a custom simulation tool.

The main goal of the simulation was to verify that the 16-point processor implements the 16-point Winograd Fourier Algorithm using the circuits and control signal interactions built into the chip design. By viewing the processor as a set of bit streams, traveling lock-stepped with respect to each other through the pipeline, it is possible to see the basic form of a high level modeling and simulation program. The interaction between the bit streams is specified by the 16-point Winograd algorithm and implemented using the hardware structures described in the preceding chapter. A more detailed description of the design and operational characteristics of these circuits is available in [4]. This chapter will describe the programs which are used to simulate the processor, and the data structures used to form the link between the model and the actual circuits.

Simulation Description

The simulation was designed and coded using the decomposition of the processor outlined in Chapter 4. This allows the output of the programs developed in this simulation to be compared directly with the output of a VHDL simulator. It is also directly compatible with the algorithmic simulator developed by Taylor [17]. The output of the simulator is a stream of bits for each slice of the processing elements. Follow on efforts which implement testability into the processor can use this output to generate test vectors for hardware testing.

The simulation consists of five programs which execute in sequence under the control of a shell script to simulate the WFT16 processor. The processor architecture was partitioned in the manner shown in Figure 5-1. This partitioning allows for incremental development of the simulation using outputs from previously tested modules. It also limits the size of the individual programs resulting in faster compilation and run times during program development. The output of each column is written to a file for analysis during coding and future test vector generation.

The programs are listed by name and the processor blocks which they simulate:

CS.C: The Control Sequencer.

C_CNTRL.C: The arithmetic reset and multiplier control circuitry.

PRE_WFTA.C: The PISO, ZF/PC, and three preadd columns.

MULTIPLY.C: The serial multiplier.

POST_WFTA.C: The postadder columns, parity_round circuit and the SIPO.

Programs were also written to aid in data analysis. The numerical performance of the WFT16 was simulated by [17]. A program was developed that performed the WFT at the algorithmic level, using double precision integers and the WFT16 equations. Taylor wrote a decimal-to-binary conversion program which was modified to compute odd parity and append it following the MSB of each input word. It is used to convert his input data sets into a form usable as input to this simulation. The loop between the out-

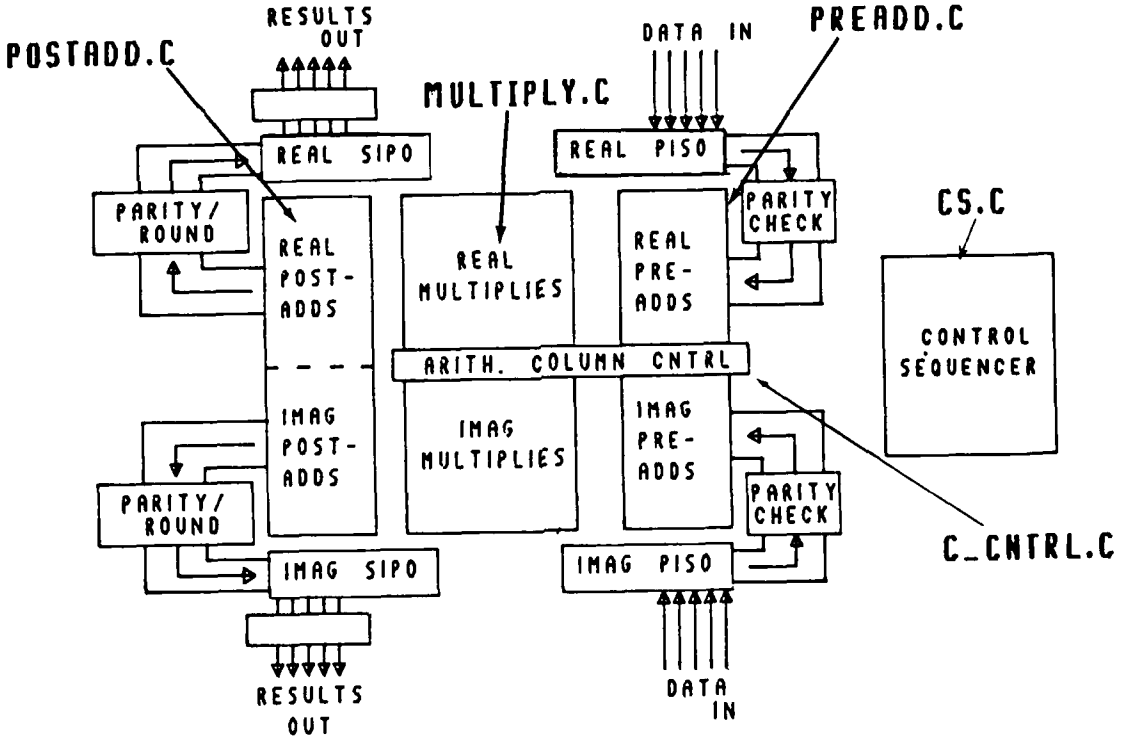


Figure 5-1. Partitioning of the Processor for Simulation

put of this simulation and the algorithmic simulation was closed by writing a program which converted the binary outputs of the simulation into decimal for comparison against the results of Taylor's numerical WFT simulation.

5.3. Time

The representation of time in VHDL is done with the physical type *time* which could be a variable length or even infinitesimally small time unit. The simulator kept track of events and transactions scheduled to occur. In this simulation, the artifice of VHDL time is replaced by a spatial separation of events. Events that are scheduled to

occur simultaneously are textually grouped together. The WFT16 processor controls the hardware with a two-phase clock. The events which are scheduled to occur at the same point in time can be roughly partitioned into events which are scheduled to happen on one of the two clock phases. The simulation uses two counters to model the system clock. The master clock, kept in the control sequencer, is appended to every control word. Every clock cycle the master clock is compared against an internal clock, kept in each program. If they are not identical the simulation program will issue a non-synchronization message and terminate. All column outputs written to files are also tagged with internal clock time. When this data is used by another program the time tag is checked against its internal clock in the same procedure outlined above. This method keeps the pipeline lock-stepped during file communication. The pipelined architecture can be modeled by a program looping structure which sequentially runs through all bit manipulation and movements (shifting) operations in the pipeline. The internal counter is incremented every cycle, which is compared against a limit to determine when to terminate the simulation.

A clock cycle can be defined as a $\phi 2$ event which is followed by a $\phi 1$ event. This definition is necessary because of the sequential nature of the simulation. The program operates in a loop, first $\phi 2$ events occur, then $\phi 1$ events occur. The process repeats itself for as many cycles as control signals are available. In the hardware, operations occur concurrently based on the phase of the clock, $\phi 2$ and $\phi 1$ events are separated in time, in the simulation these events are separated textually. A $\phi 2$ event occurs when data available at the input is gated into a $\phi 2$ latch. Any combinational logic which occurs between a $\phi 1$ latch and a $\phi 2$ latch is also defined as a $\phi 2$ event. $\phi 1$ events are defined in a similar manner. To model the propagation of a bit through delay stages without a two phase clock, provisions must be made to ensure a data bit is not available to affect the inputs of the succeeding stage until one simulation cycle after it was created or modified (this is similar to a signal assignment statement not being allowed to affect the current value of a signal in VHDL). This is accomplished grouping all the $\phi 2$ events at

the beginning of the program, and all the $\phi 1$ events at the end. This forces the $\phi 2$ latches to work with the bits put into the $\phi 1$ latches at the end of the preceding simulation cycle. For example consider the last stage of the PISO and the following PC/ZF cell shown in Figure 5-2.

Inputs to the MSFFs are gated by a $\phi 2$ clock pulse, and are moved into the second latch by a $\phi 1$ pulse. The combinational logic which takes place between latches is simulated prior to the $\phi 2$ latch in the PC/ZF cell. The code would be written and executed in the following sequence:

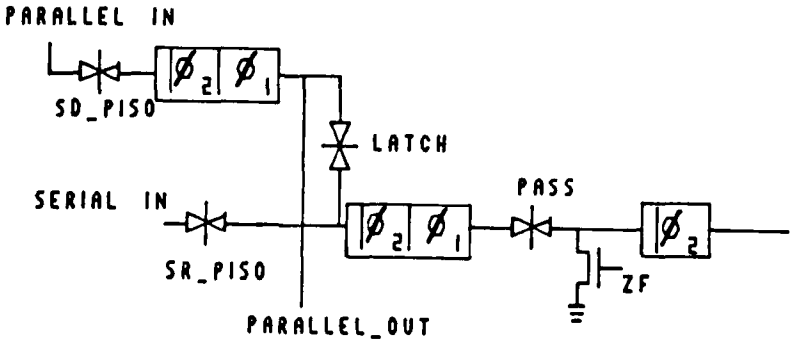


Figure 5-2. PISO and PC/ZF Stage

START:

$\phi 2$ events

Read Control Word.

PISO_CELL EVENTS

If SD_PISO high,
latch data into $\phi 2$ latch from parallel $\phi 1$ latch above.
If SR_PISO high,
latch data into $\phi 2$ latch from serial $\phi 1$ latch to the left.
If LATCH_PISO is high,
latch data from parallel $\phi 1$ latch in same PISO_CELL.

PC/ZF CELL EVENTS

If ZERO_FILL high,
put 0 into $\phi 2$ latch.
If PASS high,
pass output of PISO to the $\phi 2$ latch.
If neither signal high,
do nothing.

$\phi 1$ Events.

Move parallel and serial data in PISO from $\phi 2$ latch into the $\phi 1$ latch.

Increment internal control counter.

GO TO START and REPEAT

The data is buffered in this fashion, keeping bits from being used by the following stage until one clock cycle after they are created. This prevents the bits from racing through the simulated arithmetic pipeline. Additional functions, such as a column of adders, can be inserted into the code by separating the $\phi 2$ events and $\phi 1$ events and placing them in sequence with respect to the components they follow in the actual hardware.

5.4. Program Descriptions

This section will describe the programs and data structures used in the WFTA simulation. Data structures are used as "software circuits" in the simulation. For each of the primary cells discussed in the preceding chapter a data structure has been developed. The elements of the data structures, for the most part, represent a clocked storage element (a D-latch) in hardware. This implies that there is a correspondence between the latches of the real hardware and the variables declared in the structures. Figure 5-3 shows the correspondence between one data structure and the hardware it is supposed to model. The structure for the +1 multiplier cell, MULTX1, has one variable for each $\phi 2$ and $\phi 1$ latch. The variable tmp_sum is an exception, being used as a holding bin for the result of the addition operation. Assignment of this variable to the sumffclk2 latch is dependent on the value of the control signal sign_ext.

```
STRUCT
  int ff1clk1;
  int ff1clk2;
  int ff2clk1;
  int ff2clk2;
  int ff3clk1;
  int ff3clk2;
  int sumffclk1;
  int sumffclk2;
  int carryffclk1;
  int carryffclk1;
  int tmp_sum;
MULTX1;
```

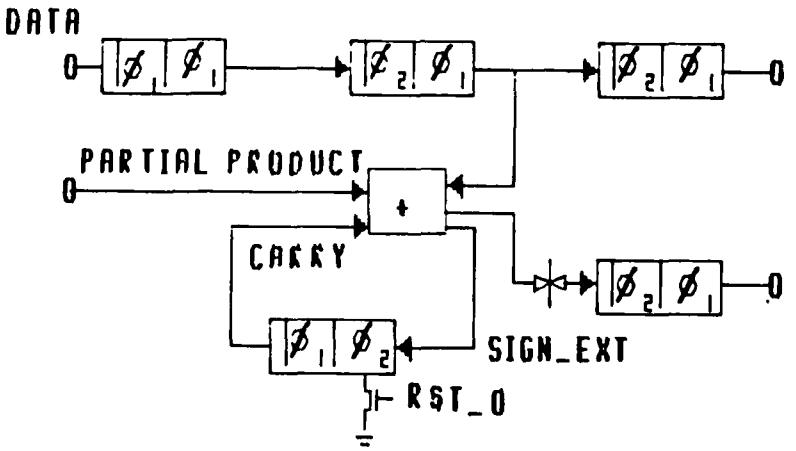


Figure 5-3. Example of a Simulation Data Structure

In addition to latched variables, signals which must travel through more than one level of logic may also declared as a variables in the data structures.

5.4.1. CS.C The operation of the control sequencer is simulated by this program. The control sequencer, shown in Figure 5-4 consists of a 32 bit ring counter, a PLA, output buffers and XROM address generation circuitry (not modeled). It generates twenty control signals which are used to control the arithmetic and I/O circuitry of the WFT processor.

The is the only program which will prompt for input, the scale factor and the number of clock cycles to simulate. The output is a file, master_control, containing the number of cycles simulated, a time tag, and the twenty control signals. The last two

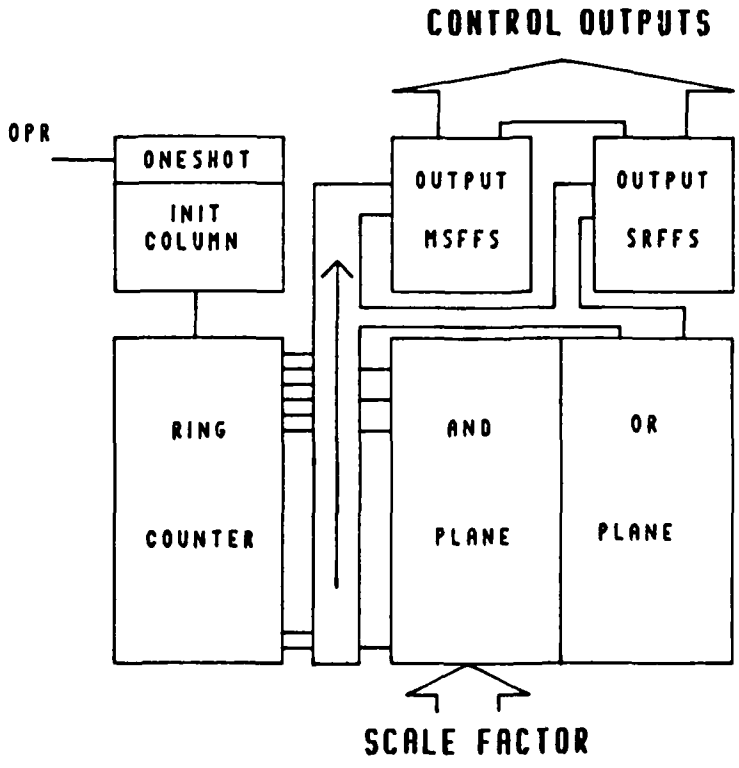


Figure 5-4. WFTA Control Sequencer

items are written to the file after every simulation cycle. These control signals are active at the beginning of every simulation cycle. This the source file for control information for all other programs used in the simulation.

The data structures used in this program are the MSFF and the SRFF (set reset flip flop). The MSFF structure contains two variable bins for holding the contents of the two latches. The SRFF contains three variables, a set, a reset, and an out variable. The Set variable is used to maintain the current value of the output interval signal. The out variable must be initialized to zero prior to the start of simulation.

The ring counter is a chain of thirty-two MSFFs connected in series. In hardware, the output of the last MSFF in the chain and the input to the first are connected by a feedback loop. This allows the bit to keep cycling through the counter while the continue signal remains high. The ring counter is modeled using a counter that rolls over mod thirty-two. If the result of the modulus operation is zero, the input to the first MSFF is set to one, if one, the input is set to zero. The bit advances one MSFF during each simulation cycle. Control signals are generated as a function of the position of the bit modeled in the controller. There are three basic types of control signals in the WFT16: pulse, fixed interval, and variable interval. Pulse signals are high for one clock pulse. These signals are assigned by reading the output tap of the MSFF representing a particular clock cycle. If the bit is in the $\phi 1$ latch of that MSFF the signal is set to one, zero otherwise. Interval signals are high over the same clock interval each time the simulation is run. These signals are modeled with a boolean expression that evaluates to true if the clock counter value is within the interval the signal is supposed to be active. If the expression evaluates to true, the corresponding control signal is set to one, zero otherwise.

The final class of signals is a function of the adaptive scaling algorithm. The interval these signals are high depends on the value of the three bit scale factor. The eight cases are modeled using an if/then control structure as shown below:

```
if condition 1
  action 0;
else if condition 2
  action 2;
   .
   .
   .
else
  action 8;
```

Where the *condition* represents the boolean value of the "anding" of the scale factor and clock counter, and the *action*, the setting or the resetting of of the set-reset variable. If the case evaluates to true, a set or reset flag is set to one. *Action* 8 is the default that occurs if none of the cases evaluate to true. The SRFF function then evaluates the three variables: set, reset, and out, setting or resetting the control signals accordingly.

5.4.2. C_CNTRL.C. This program generates the signals used to control data flow through the arithmetic pipeline. The only data structure used, a MSFF, is described in section 5.2.1. The arithmetic control circuitry consists of a chain of forty-eight MSFFs connected in series. The input to the first MSFF in the chain is the reset_add signal generated in the control sequencer. As the bit traverses the chain, it will be used as a reset signal for the carry and borrow MSFFs in the preadd and postadd arrays. It will also generate the four control signals needed for the multiplier cell; reset_0, reset_1, sign_ext, and rstdc. The output of the program is written to three files, one for the preadd array, one for the multiplier array, and one for the post add array. This file structure represents the partitioning of the arithmetic portion of the WFT16 architecture as shown in Figure 5-1.

5.4.3. PRE_WFTA.C. The PRE_WFTA.C program simulates the operation of the processor from the PISO input to the $\phi 1$ latch following the third column of the preadd array. This program capitalizes on the symmetry between the real and imaginary por-

tions of the WFT16 algorithm. Rather than write one program to compute both real and imaginary results simultaneously, the same program can be re-used with the different data sets and run twice. Since these data sets are completely independent through two columns of the postadd array, the multiplier program can also be reused in this same fashion. Structures were defined for each of the macro-cells in the preadder. They are the PISO, which consists of four elements, two per flip-flop, the ZERO_FILL which contains a MSFF structure, a latch variable, and two logic output variables. The Adder-Subtractor was broken into two sections, the input and the output and variables declared for the X, Y, carry and borrow inputs, and the SUM, DIFF, carry and borrow outputs. Comparison of the data structures shown in this Figure and the circuit diagrams of the hardware described in [4] will show a one to one matching of the variables and the outputs of circuit components. This approach leads to a natural synthesis of the simulation program from the hardware components.

The simulation of multiple cycles is done using a loop controlled by the internal clock counter. The loop condition is set by the first word of the master control file which is the number of cycles for which control signals are available. While the internal clock is less than this value, the simulation will proceed. The program is set up by reading the master_control file and preadd control word before every simulation cycle.

The PISO is implemented as a [16][24] array of PISO_CELL structures. The MSB of the input word is located in column sixteen of the array. The LSB, located in column 1, is shifted out first. The output of the PISO is sent to the PC/ZF, a column of 16 PC/ZF cells, where the parity bit is stripped and the wordlength is extended to thirty-two bits.

The preadd array, which follows the PC/ZF, is composed of three columns of adder-subtractor (A/S) cells and MSFFs Each column of the preadd array either computes the sum and difference of the inputs, or delays it for one clock cycle. The MSFFs are used as place holders to maintain bit synchronization with the other elements of the

bit vector which are passing through the A/S elements. The A/S is defined to compute the sum and difference, $x \pm y$, of the two serial input vectors. Thus, the minuend is assigned to the x variable, and the subtrahend is assigned to the y variable. The interconnections of the A/S and MSFFs of the preadd array is a function of the Winograd Algorithm and is shown in Figure 5-5.

A note concerning the usage of the reset_adder signal is in order. Unlike all the other control signals, this signal is the output of a $\phi 2$ latch in the C_CNTRL circuitry. In the A/S structure, the reset signal is "anded" with the $\overline{\phi 2}$ clock, which effectively causes the signal to be active on the following $\phi 1$ pulse. At this time the reset signal will cause both latches to be reset. The reset signal should reset the carry and borrow following the MSB arithmetic operation of the preceding data set.

5.4.4. MULTIPLY.C. The multiplier array in hardware is an [18][14] array of multiplier cells. Each cell represents one bit of the Booth's quaternary encoded binary coefficient. In software, the serial multiplier is represented by an array of data structures, each array element being one of the five possible multiplier cells. The data structures are declared to be external so that all variables will hold their value between function calls. An example of the multiplier data structure was shown in Figure 5-3.

The simulation proceeds by columns. The mult_cntrl file consists of a time tag and fourteen sets of four bits each which are the four control signals for a column. Before the $\phi 2$ event of each column of eighteen cells, the program reads in the control word for that column. Next the partial product and data bit are read into the cell structure representing a particular location in the array. Finally the function which simulates the multiplier cell is called to evaluate the bits, and shift the data through the MSFFs. This is done by a function call that has the arguments, the pointer to the data structure, and the control signals needed for that particular cell. The pointer name **pa70** imparts certain information about the location of the structure in the array. The p-**7**- means that the multiplier cell is in the seventh row of the array. The p**a**-- means that the cell is in

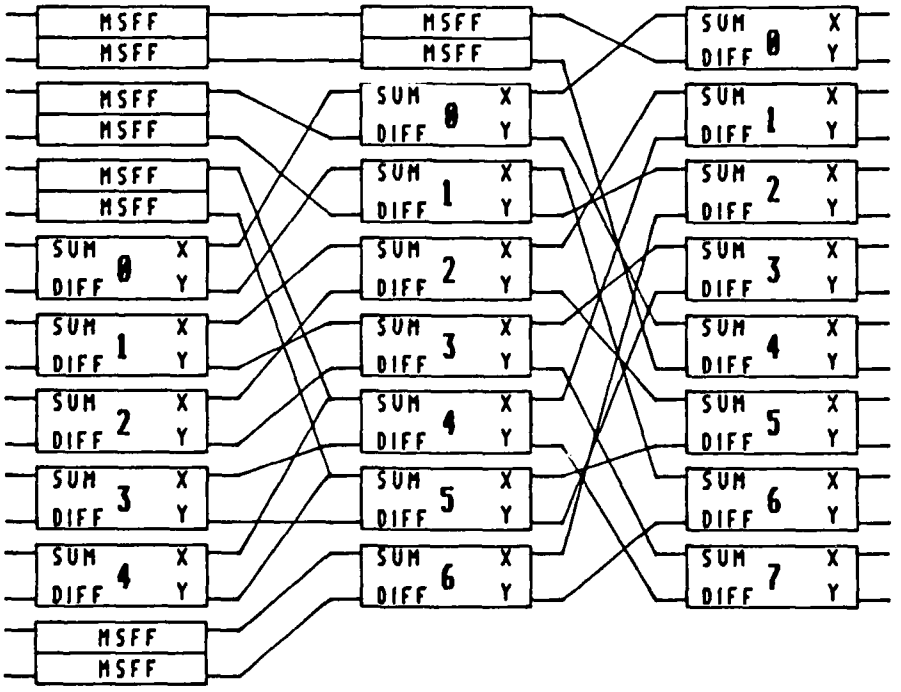


Figure 5-5. Preaddition Operations in the WFTA Processor.

the tenth column. Finally, the p--**0** means that it is a 0 multiplier cell. Therefore, this a the pointer to the multiplier array element which is in the tenth column of the seventh row, and calls the m0 function. The other multiplier identifiers are: p--1 for the +1 multiplier, p--2 for the +2 multiplier, p--n for the -1 multiplier, and p--q for the -2 multiplier. Theory of operation of the multiplier cells is covered in detail in [4] and will not be covered here.

5.4.5. POST_WFTA.C. This program simulates the WFTA pipeline from the output of the multipliers to the output of the SIPO. It is a dual program in that both the real and imaginary operations are simulated simultaneously. The A/S elements are the same as the preadders, and the SIPO is essentially the same as the PISO. The only totally new data structure used is the parity round cell. The parity round cell consist of several levels of combinational logic, variables were declared for the outputs of the logic as well as the standard latch variables. The interconnection of the postadd columns is shown in Figure 5-6. Results from the imaginary and real sections of the processor are mixed in the third column of the postadders.

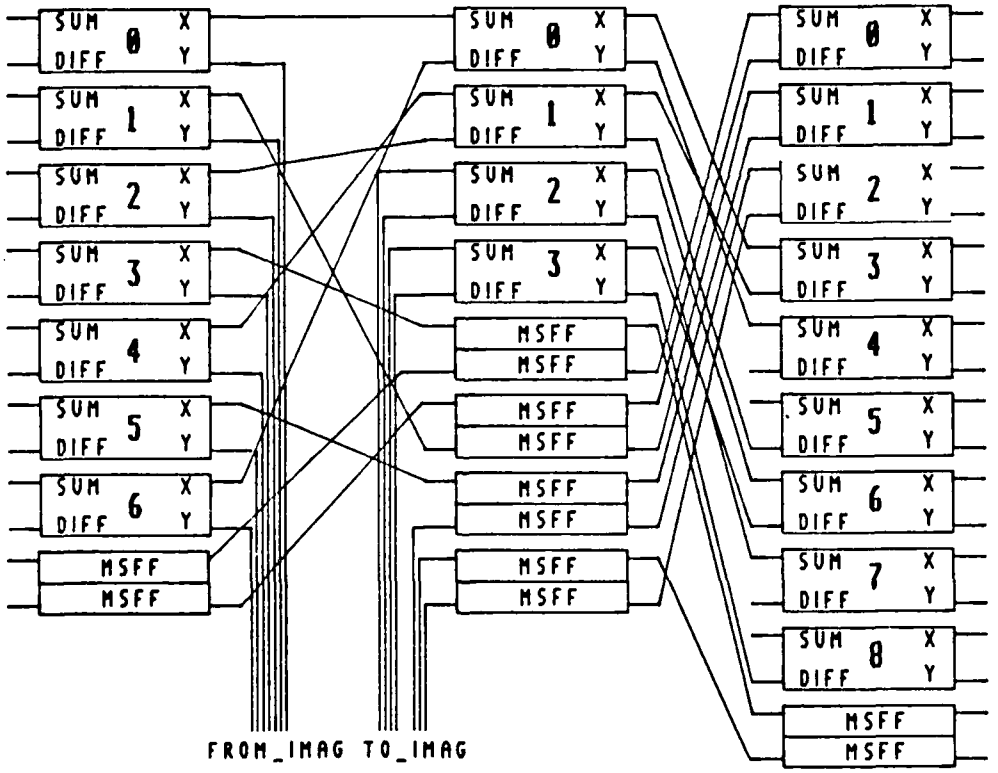


Figure 5-6. Post Addition Operations in the WFTA Processor

The outputs of the real and imaginary columns, and the output of the both SIPOs are written to a output files.

5.5. Generation of a WFT Simulation

This section will discuss considerations involved in building a WFT simulation from the data structures and functions defined in earlier sections. The cells were designed to be single independent units. Larger computational units may be created merely by declaring more instances of the cell structure. The key parameter in all the programs is the number of clock pulses which constitute a simulation cycle. The 16-point architecture uses 32, the 15 would use 30, and the 17 would use 34 clock pulses. The loop control functions are all done with modulo (number of clock pulse) arithmetic. An important point regarding the control signals is that the signals are set high in the control sequencer on the cycle which they are supposed to be used. In actual hardware, the signals might be created a clock cycle ahead of time and buffered in an output MSFF. The timing diagram used as a source document should be examined to determine which interpretation was used in generating the diagram. The constructs used in modeling the control sequencer were selected to allow changes to be made in the timing diagram without requiring extensive changes to the simulation.

The PRE_WFTA.C and POST_WFTA.C adder-subtractor (A/S) elements are interconnected using the equations from the Winograd program written by Taylor and the coefficients used to generate the multiplier array were obtained from [4]. The outputs from the algorithmic simulation program were also used to verify the results of the simulation.

Implementing the serial multiplier array as a fixed array of data structures is a flexible and easily understandable approach. The coefficient encoding can be changed without having to redo the entire array. The major difficulty encountered in constructing the simulation was the timing of events across the program boundaries. Reading data from files is normally a $\phi 2$ event, (the start of the simulation cycle). On the other

hand, writing to a file is a $\phi 1$ event, (the end of the cycle). The clock time appended is the cycle which the data was created. However, when a file is read, the program treats the data as being applicable for that simulation cycle. Problems arise when the output of one program, such as PRE_WFTA.C is used as the input to the MULTIPLY.C program. The multiplier treats the preadd outputs as input data valid on the same clock cycle that is was created. The effect of this is that the data is arriving one cycle before it was created. In many cases, the effect is barely noticeable, showing up as an error in the LSB of some of the answers, and very hard to detect. In some answers, those with just the right number of sign extensions, the fact that the control signals and data are out of synchronization by one cycle causes the MSB of the data, the sign bit, to overflow, changing the sign of the intermediate result. The fix was simple, once the problem was identified. Data read across program boundaries was defined to be $\phi 1$ event so data effectively was being read at the end of the simulation cycle which it written to the file. The source document for the control signals defined the clock cycle that the signals were to be active. The CS.C program generated them on this cycle, therefore this problem did not affect them. Once this problem was detected and corrected, building the complete simulation essentially consisted of interconnecting the data structures in the manner specified by the 16-point WFT algorithm, and debugging programming errors.

5.6. Simulation Scenario

C shell scripts were written to automate the execution of the simulation programs. Execution was subdivided into two scripts, generation of the control signals, and simulation of arithmetic operations.

The script **control** executes the programs CS.C and C_CNTRL.C. CS.C is the only program that requires input from the keyboard. It will prompt for the number of clock cycles to simulate and the scale factor to the input data set. The scenario is shown in Figure 5-7. Control files are generally good for multiple simulations so they do not have to be regenerated unless the scale factor of the input data changes.

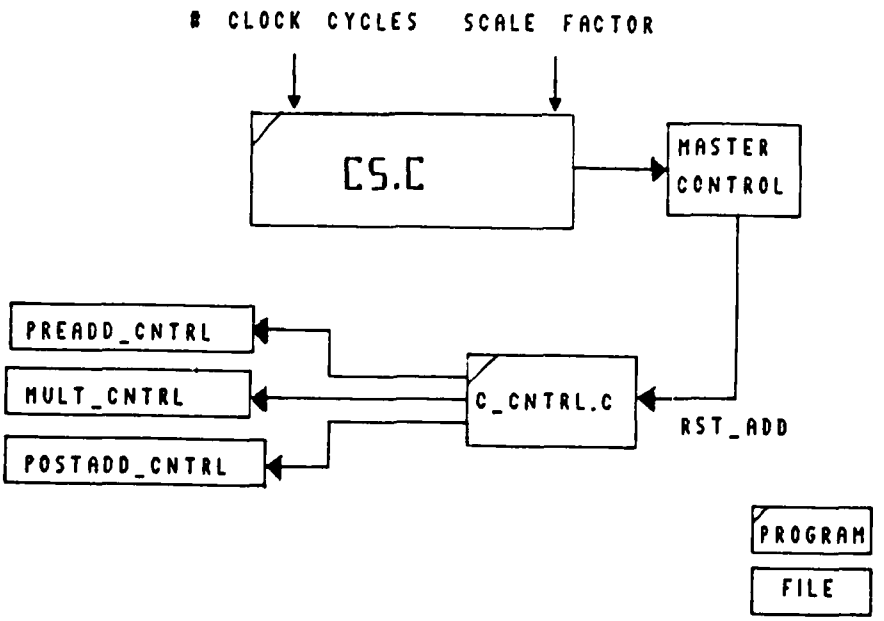


Figure 5-7. Control Generation Simulation Scenario

The script **demo** runs the arithmetic simulation programs. The scenario that the script executes is shown in Figure 5-8 In addition, the script also runs the output format programs which convert the binary streams into integers. In the absence of any operator action, the converted output will scroll across the screen, so the normal procedure is to redirect the screen output to a data file with the command: demo >& tst_output, which will send the output to the file tst_output.

The code used to simulate the WFT16 is included in Appendix 2.

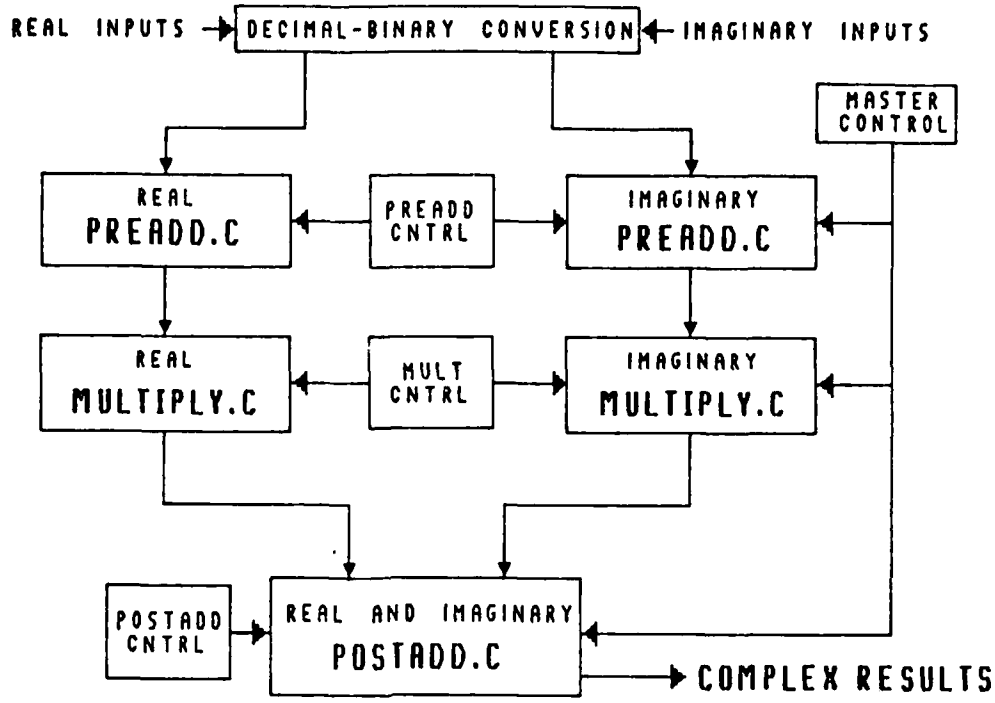


Figure 5-8. Arithmetic Pipeline Simulation Scenario

CHAPTER 6

Summary and Conclusions

6.1. Overview

This thesis addressed the problem of modeling and simulation of a VHSIC class signal processor. A new Hardware Description Language (VHDL) is being developed to address this need by the VHSIC program office. VHDL is intended to be a medium to communicate design intent for large, complex designs in a concise manner. Perhaps more importantly, the VHDL code used to describe the circuit also serves as the input to a hierarchical simulator. The simulator allows a design to be simulated at different levels of abstraction within the same design entity. This is a key concern when discussing large designs which may be composed of hundreds of thousands of transistors.

VHDL was originally targeted to be the vehicle used to simulate the design of a signal processor that embedded the Winograd Fourier Transform Algorithm in a pipelined architecture. An early delivery of the VHDL simulator failed to materialize, so a custom simulation tool was developed to perform this task. The work done in decomposing and modeling the circuit using VHDL translated well into the new simulation - the C simulation. This simulation modeled the processor at the bit level, using hardware-like data structures. It was used to validate the architecture, cell functionality, and control/timing interaction of the WFT16 processor. Insights obtained during the development and coding of the simulation was also useful in correcting errors which had slipped into the processor design.

6.2. VHDL

VHDL was applied to the problem of modeling the WFT16 at a level where the functionality of the individual circuits may be observed. The functional level was deter-

mined to be the level at which bits could be seen passing through latched storage locations. This visibility was achieved by decomposing the circuit into its smallest functional processing components, and then modeling these components. It was found to be quite easy to model low level cells, such as inverters, T_gates, and latches, and to build progressively larger circuit models by instantiating the previously described subcomponents. This approach will model and simulate circuit operation at the very lowest level of detail. The CMOS latch was modeled at the transistor behavior level using this approach. However, the run time of circuit simulations containing large numbers of devices modeled at this level is expected to be excessively long. VHDL provides a hierarchical approach, the circuit could be modeled at a higher level of abstraction for the purposes of simulation. This alternate approach, modeling only the bare functionality, coupled with limiting redundant event scheduling, should allow large circuits to simulate much more efficiently.

The WFT16 processor was decomposed into a set of lower level behaviors, and modeled at both the functional and structural levels. The structural description is useful for seeing the architecture of a cell, while the functional description is more abstract. Functional descriptions may provide a clear picture of the device behavior, but their primary purpose is to aid efficient simulation. Thus, two VHDL architectural bodies were written for most cells, one to document the architecture, and the other to describe the function and be used to drive a simulation. The MSFF was found to be the highest level which could be efficiently modeled using this approach, modeling the functionality, while preserving some structural flavor of the design. Higher level cells, modeled at the functional level, would instantiate a MSFF as part of the overall design.

The basic concepts behind the VHDL were found to be relatively simple. The sytax allows the VHDL descriptions to be written which are clear and concise. It appears to be difficult to write a description which would not be fairly readable and understandable to someone with a basic knowledge of the language. However, this absence of complexity leads to descriptions which are tedious to write. There are also many areas of the

syntax where questions arise as to the actual meaning or implications of a particular construct. This is not unexpected, it arises when learning any new computer language, but the problem is exacerbated due to the youth of the language, lack of documentation and examples.

There appear to be two aspects of VHDL, design documentation, and design simulation. Of the two, only documentation is fully supported by the language at this point in time. There also appears to be two types of description that correspond to these aspects. The goals of modeling and simulating a circuit using VHDL seem to be facilitated by separate approaches. A purely structural description will not simulate as efficiently as will one specifically written for that purpose. On the other hand, circuit descriptions written with an eye towards minimizing simulation time will not be as clear in describing the circuit structure. The MIXED type of architectural description alleviates this problem somewhat, but the actual improvement using abstract descriptions alone is not known.

6.3. C Simulation

The WFT16 processor was modeled and simulated using the C programming language. Primitive cell structures were defined to model each of the primary circuits at the bit level. These primitive cells were then declared and interconnected using the 16-point WFTA as a netlist. A clock was defined that was used to march bit streams through the this cell structure. In this fashion, the WFT16 architecture was shown to perform the 16-point DFT, thereby validating the architecture, the results of Taylor's numerical simulation and the signal to noise ratio projections.

6.4. VHDL Recommendations

The run time of any VHDL driven simulation needs to be quantized, using both functional and structural descriptions. The improvement in run time for such techniques as limiting event scheduling unless the input/output transform will cause an

event on the output, and removing redundant circuit functions, (such as the feedback loop of a static latch) should be studied. If the difference is not significant, the designer will have the flexibility to run a more structurally flavored circuit models (easier to write) in the simulation. Finally, the descriptions should be validated to ensure that they do in fact perform identical functions and may be interchanged at will.

At this point in VHDL's life cycle the documentation aspect is fully supported by the syntax. This capability should be used to document the cell structure and other parameters of the cells developed during the VLSI courses. These include the name of the cell, the name of subcells, and design information which would be useful in automating the layout process at some point in the future. The yearly turnover of personnel in the AFIT environment, as well as the complexity of the cells, require that clear, structured documentation exist to aid the continuity of research effort. Thus the major recommendation in this area is that the VHDL should be used to document the CMOS cells which were built over the course of the last year, and in future years, by all VLSI design groups.

6.5. Simulation Recommendations

Although the simulation was designed to simulate the WFT16 processor, the design philosophy is applicable to the other processors in the PFA pipeline, the WFT15, WFT17 and also to other architectures which have lock-stepped, bit serial pipelines. It models the hardware at the bit level, and has the primary advantage that the run time of the simulation is very short, under one CPU minute, to run several 16-point data sets through the pipeline. The C simulator is a tool which should grow along with the research in pipelined serial signal processing architectures. Any design which uses the cells designed in the WFT16 effort can use the structures and concepts of the simulator. At this time, a class project is developing a program which will layout and simulate the multiplier array for the WFT processors. Future projects in this area could include automating the layout of the other WFT processors, leading to computer generated sig-

nal processor layout.

6.6. Conclusions

The WFT16 architecture has the potential of an order of magnitude improvement in processor throughput over existing designs. Based on the research discussed in this report, and the reports of the other members of the design team,[17], [4], [13], there exists a high degree of confidence that the WFT16 processor will work, as expected, on first silicon.

Appendix 1 VHDL Modeling

The WFTA processor was decomposed into its primary circuits in the Chapter 4. The very lowest level of decomposition shows that all the circuit elements are built from transmission gates and the basic logic gates. These cell are linked together to build latches, logic gates and flip flops. This appendix contains the VHDL models the primary circuits. The devices which will be modeled range from transmission gates to the booths multiplier cells.

1.1. Transmission Gate This is a behavioral description of a transmission gate. It actually needs both senses of the control signal to drive the CMOS 'P' and 'N' transistors, but since the inverted signal does not perform any independent function, it is not included in the port list or architectural description. The T_gate is sensitive to both the input signal switching and also transitions on the control line. Therefore, both these signals are included in the process sensitivity list. The T_GATE is sensitive to both the input and control signals. However if control = '0' then the output will be not change regardless of the value of the input. The process statement reflects this consideration. If the control has not just changed to '1' the output will not reflect the input. As soon as control switches to '1' then the input will be enabled. As long as the control remains high the output will reflect the input, when it falls the input signal will be disabled and not be allowed to cause events in the transaction queue.

```
--*************************************************************************
--
--      DATE:  29 JULY 1985
--
--      TITLE: Transmission Gate Descriptions
--      FILENAME:t_gate.v
--      LANGUAGE: VHDL
--      ENTITY:
              entity T_gate
              ( bit_in: in Z_bit;
                control: in CONTROL;
                bit_out: out Z_bit;
              ) is
              end T_gate;
--
--*************************************************************************/

       architecture BEHAVIOR_1 of T_gate is


       block
         begin

         process(bit_in, control)
         begin
           if (control and not control'stable) then
             enable bit_in;
           end if;
           if (not control) then
             disable bit_in;
           end if;

             bit_out <= bit_in;
```

```
 end process;

end block;

end BEHAVIOR_1;
```

1.2. Z_INVERTER

The Z_INVERTER accepts inputs of type Z_BIT and returns type bit. '0' and '1' inputs are negated and mapped to the output in the normal fashion. A 'Z' input has no effect on the value of the output, the output will retain its previous value. The port mode "buffer" is used to allow assignment of the output to itself when the input is 'Z'. This device is used in the latch in order to behaviorally model a high impedance input.

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE:  Z INVERTER
--      FILENAME: z-invarch.v
--      LANGUAGE: VHDL
--
--      ENTITY:
                entity Z_INVERTER
                  (bit_in : in Z_BIT;
                   bit_out: buffer BIT) is
                end Z_INVERTER;
--
--*************************************************************************/

 architecture BEHAVIOR of Z_INVERTER is

   block
     begin
     process (bit_in)
     begin

       if(bit_in = 'Z') then
         bit_out <= bit_out;
       elsif (bit_in = '1') then
         bit_out <= '0';
       else
         bit_out <= '1';
       end if;

     end process;
   end block;

 end BEHAVIOR;
```

1.3. Full Adder-subractor

This section contains two descriptions of the full adder-subtractor which is used in the pre- and post-add arrays. The first is a completely structural description listing all the logic gates and interconnections. The second is a boolean algebra description of the functionality of the circuit.

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: LOGIC LEVEL DESCRIPTION OF AN ADDER/SUBTRACTOR
--      FILENAME: add-logic.v
--      PROJECT: THESIS
--      LANGUAGE: VHDL
--
--      ENTITY:
                entity ADD_SUB2
                  (A, A_NOT, B, CY, BY: in bit;
                   SUM, DIFF, CY_OUT, BY_OUT: out bit) is
                end ADD_SUB2;
--
--      FUNCTION:
--     This is a pure structural description of the
--     ADDER/SUBTRACTOR cell used in the PREADD and POSTADD columns
--     of the WFTA.  Because of the cmos transmission gates used,
--     it is necessary to input A and A_NOT.
--
--
--
--*************************************************************************/

architecture PURE_STRUCTURE of ADD_SUB_CELL

PURE_STRUCTURE:

  block

    component OR_GATE port (A,B: in bit; C: out bit);
    component AND_GATE port (A, B: in bit; C: out bit);
    component XOR_GATE port (A, B: in bit; C: out bit);
    component XNOR_GATE port (A, B: in bit; C: out bit);
    component INVERTER port (A: in bit; C: out bit);


    signal S5, S6, T1, T2, T3: bit;
    signal S1, S2, SUM, CY_OUT, DIFF, BY_OUT: atomic WIRED_OR bit;

  begin

-- signals S1, and S2 are common to both the adder and subtractor

        CI: INVERTER port(B, B_NOT);
        CA1: AND_GATE port (A, B_NOT, S2);
        CA2: AND_GATE port (A_NOT, B, S2);
        CA3: AND_GATE port (A, B, S1);
```

```
        CA4: AND_GATE port (A_NOT, B_NOT, S1);

        -- the gates labeled C.. make up the XOR, XNOR functions
        -- given by X1, X2 below.  this was done to make this description
        -- compatible with the actual circuitry implemented in CMOS
        --X1:  XNOR_GATE port (A,B,S1);  -- S1 = A xor B
        --X2:  XOR_GATE port (A,B,S2);  -- S2 = A xnor B

-- adder section
        A1:  AND_GATE port (CY, S1, SUM);  -- CY (A xnor B)
        I1:  INVERTER port (CY, T1);       -- CY'
        A2:  AND_GATE port (S2, T1, SUM);  -- CY' (A xor B)

        O1:  OR_GATE port (A, B, S5);      -- (A or B)
        A3:  AND_GATE port (A, B, S6);     -- (A and B)
        A4:  AND_GATE port (S6, T1,CY_OUT); -- CY' (A and B)
        A5:  AND_GATE port (CY, S5,CY_OUT); -- CY (A or B)

-- subtractor section

        A6:  AND_GATE port (BY, S1, DIFF);  -- BY (A xnor B) = DIFF
        I2:  INVERTER port (BY, T2);                -- BY'
        A7:  AND_GATE port (S2, T2, DIFF);  -- BY' (A xnor B) = DIFF
        I3:  INVERTER port (A, T3);         -- BY (A xnor B) = BY_OUT
        A8:  AND_GATE port (BY, S1, BY_OUT);  -- BY (A xnor B) = BY'
        A9:  AND_GATE port (T3, S2, BY_OUT);  -- BY (A xnor B) = BY_OUT

    end block;

end PURE_STRUCTURE;
```

```
-- This is a boolean algebra description of the adder/subtractor cell
architecture  LOGIC_STRUCTURE of ADD_SUB_CELL is
  block
     signal S1, S2: bit  --temporary signals for xor, xnor resources
    begin
         S1 <= A xnor B;
         S2 <= A xor B;
         --adder section
         SUM <= (S1 AND CY_IN) or (S2 and not CY_IN);
         CY_OUT <= ((A or B) and CY_IN) or (A and B and not CY_IN));
         --subtractor section
         DIFF <= (S2 and not BY_IN) or (S1 and BY_IN);
         BY_OUT <= (S1 and BY_IN) or (S2 and not A);
    end block;
end LOGIC_STRUCTURE;
```

1.4. Resettable CMOS Latch

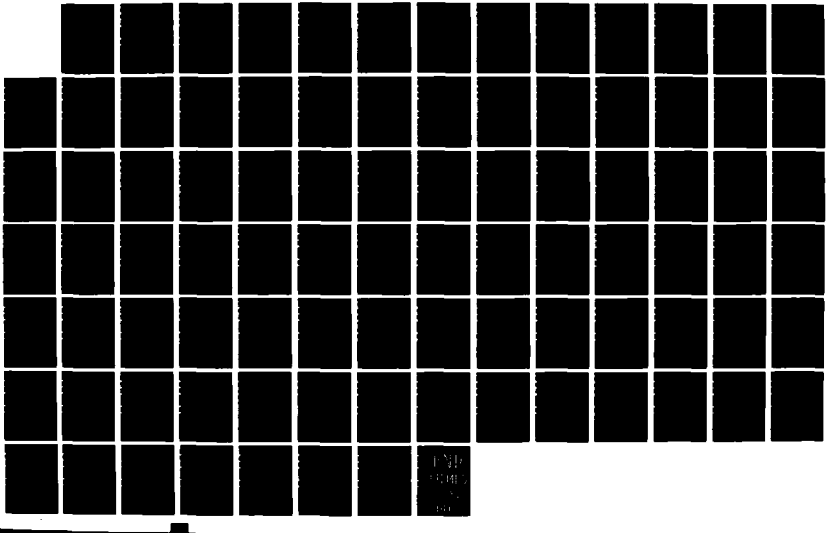
The resettable latch is used as the front end of the resettable MSFF. The reset signal is "anded" with clock_not to avoid fighting at the input node. The VHDL code follows the normal latch, but with the addition of the reset signal to the interface declaration and the process statement. The reset signal is meant to take precedence over the input, (a direct connection to ground will drain any charge on the node) for this reason it is listed first in the signal assignment statement.

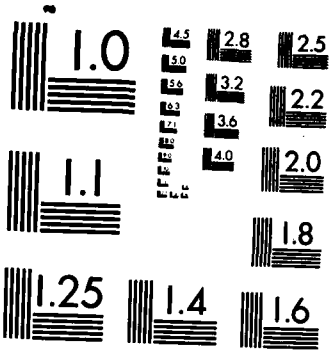
```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE:  RESETABLE CMOS LATCH
--      FILENAME: rlatcharch.v
--      LANGUAGE: VHDL
--
--      ENTITY:

              entity RESET_LATCH
               (bit_in: in Z_BIT;
              CLK : in clk_signal;
              CLK_NOT : in clk_signal := 1;
                reset: in CONTROL;
                bit_out: buffer BIT ) is
              end RESET_LATCH;
--
--*************************************************************************/

architecture MIXED_DESCRIPTION of RESET_LATCH is

block

 signal t_gate_out; : Z_BIT;
      l_fdbk, invert_out: BIT;

 component T_GATE port (a: in Z_BIT; cntrl: in CONTROL; x: out Z_BIT);
 component Z_INVERTER port(b: in Z_BIT; y: buffer Z_BIT);
 component INVERTER port(c: in BIT; z: out BIT);

 for all: T_GATE  use          -- This is a mapping between the ports declared
                               -- win the component declarations and the
                               -- formal ports listed in the interface
                               -- declaration.
  entity (T_GATE)
  portmap(BIT_IN => a, CONTROL => cntrl, BIT_OUT => y)
  body (BEHAVIOR);
 end for;

 for all: Z_INVERTER use
  entity (Z_INVERTER)
  port map( BIT_IN => b, BIT_OUT => y)
  body (BEHAVIOR);
 end for;

 for all:INVERTER use
```

AD-A164 049 MODELING AND SIMULATION OF A SIGNAL PROCESSOR IMPLEMENTING THE WINOGRAD F.. (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. J M COLLINS

UNCLASSIFIED DEC 85 AFIT/GE/ENG/85D-9 F/G 9/5 NL

2/2





```
    entity  (INVERTER)
    body (<<library>>)
  end for;

  begin
    T1: T_GATE port(IN, CLK, T_GATE_OUT)
    Z1: Z-INVERTER port(INVERT_IN, BIT_OUT);
    I1: INVERTER port (BIT_OUT, INVERT_OUT);
    T2: T_GATE port (INVERT_OUT, CLK_NOT, L_FDBK);

      process( RESET, T_GATE_OUT, L_FDBK ,CLK,CLK_NOT)
        if (not RESET'stable or  not T_GATE_OUT'stable or not L_FDBK'stable)
        then
            enable CLK, CLK_NOT;
        else
            disable CLK, CLK_NOT;
        endif;

        if ((reset = '1') and (CLK_NOT = '1')) then
          INVERT_IN <= '0';
        elsif (T_GATE_OUT /= 'Z') then
          INVERT_IN <= T_GATE_OUT;
        else
          INVERT_IN <= L_FDBK;
        end if;

      end process;

  end block;

end MIXED_DESCRIPTION;
```

1.5. Set-Reset Flip Flop

The set reset flip flop (SRFF) is used to maintain interval signals in the control sequencer and to store the parity error flag in the PC/ZF column. The SRFF is composed of three latches and some CMOS transistors. This cell can be modeled in two ways, by instantiation, and process statements. The process statement may be a little more unwieldly but it should execute more efficiently.

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE:  SET-RESET ARCHITECTURE
--      FILENAME: srff-arch.v
--      LANGUAGE: VHDL
--
        ENTITY:

--              entity SRFF
--              ( OPERATE: in BIT;
--                SET, RESET: in Z_BIT;
                CLK2,  CLK1 : in clk_signal;
                CLK2_NOT,  CLK1_NOT : in clk_signal := 1;
--                SR_OUT: buffer BIT) is
--                assert (not (set and reset))
--                  report " SET AND RESET ARE BOTH HIGH SIMULTAENOUSLY"
--                  severity error;
--              end SRFF;
--
--      FUNCTION:
--
--
--
--*************************************************************************/

architecture BEHAVIOR of SRFF is

block


signal PASS_RESET, SET_OUT, RESET_OUT, TO_OUT: Z_BIT;

begin

  L_S: LATCH port (SET,CLK2, CLK2_NOT, SET_OUT);
  L_R: LATCH port (PASS_RESET, CLK2, CLK2_NOT, RESET_OUT);
  L_OUT: LATCH port (TO_OUT, CLK1, CLK1_NOT, SR_OUT);

  PASS_RESET <= RESET when OPERATE = '1'
                else '1';

  TO_OUT <= SET_OUT when SET = '1'
            else RESET_OUT;

er d block
```

```
end BEHAVIOR;
```

```
architecture BEHAVIOR of SRFF is

block

component LATCH port(A: in Z_bit; CLK: in clk_signal; X: buffer bit);

        for all: LATCH use
          entity (LATCH)
          port map (bit_in => A, CLK => CLK, CLK_NOT => open, bit_out => X)
          body (STRUCTURE);
        end for;

signal PASS_RESET, SET_OUT, RESET_OUT, TO_OUT: Z_BIT;

begin
    -- If either the set or reset have just changed the process will be
    -- sensitive to clock transitions.

    process(OPERATE, SET, RESET,CLK2)
    begin
     if (OPERATE = '1')
         enable SET, RESET;
     else
         disable SET, RESET;
     end if;

     if not(SET'STABLE or RESET'STABLE) then
         enable CLK2;
     end if;

     SET_OUT <= SET;
     RESET_OUT <= RESET;

    end process;


    process(SET_OUT, CLK1)
    begin
     if (not SET_OUT'stable or not RESET_OUT'stable) then
         enable CLK1;
     else
         disable CLK1;
     end if;

     if SET_OUT = '1' then
        SR_OUT <= '1'
     elsif RESET_OUT = '1' then
           SR_OUT <= '0'
     else
        SR_OUT <= SR_OUT;
     end if;

    end process;

  end block;
```

```
end BEHAVIOR;
```

1.6. Master Slave Flip/Flop

The MSFF is composed of two latches connected by the signal s2. The type conversion function, convz-b, is used to convert the input signal of type Z_bit to type bit which phi_one latch expects. This will be a common problem throughout the cell descriptions. An explicit conversion mechanism is used to confirm that the design intent was to connect a tristate signal to the input of a gate. Once more process scheduling is optimized by enabling the signal s2 only if s1 has just changed state, if not the signal s2 will remain stable and will not fire a transaction.

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: MASTER SLAVE FLIP/FLOP ARCHITECTURE.
--      FILENAME: msff-arch.v
--      OPERATING SYSTEM: VMS
--      LANGUAGE: VHDL
--
--      ENTITY:
              entity MSFF
              ( bit_in: in Z_BIT;
                CLK2, CLK1: in clk_signal;
                CLK2_NOT, CLK1_NOT: in clk_signal := 1;
                bit_out: out BIT) is
              end MSFF;
--
--      FUNCTION:
--
--      this is a description of a
--      non-resettable  flip flop. The signal s1 connects
--      the PHI 1 and PHI 2 latches.
--
--*************************************************************************/
architecture STRUCTURE of MSFF is

 block

 component LATCH port(A: in Z_bit; CLK: in clk_signal; X: buffer bit);

       for all: LATCH use
         entity (LATCH)
         port map (bit_in => A, CLK => CLK, CLK_NOT => open, bit_out => X)
         body (STRUCTURE);
       end for;
   -- configuration of latch using a block configuration statement

  signal s1: BIT;        -- local signal within the MSFF
         s2: Z_BIT;

  begin

    L1: LATCH port (bit_in, clk2, s1);
    L2: LATCH port (s2,  clk1, bit_out);

    process(s1,clk1,clk1_not)
```

```
    begin
     if (not s1'stable) then
         enable clk1;
     else
         disable clk1;
     s2 <= convb-z(s1);
                                -- the output of the phi1 latch must be
                                -- converted to type Z_BIT to avoid a type
                                -- clash.
     end if;
    end process;

  end block;

end PURE_STRUCTURE;
```

```
--*************************************************************************/
architecture BEHAVIOR of MSFF is
-- This is an alternate modeling of the MSFF.
-- Note the simplicity of the modeling, it will show the same behavior
-- at the interface ports as the much more detailed model above.
-- The level of detail required in the simulation determines which
-- VHDL modeling approach should be taken, simple or complex.

block( not bit_in'stable)

   signal s1: BIT;          -- local signal within the MSFF

  begin

        s1 <= memoried bit_in when CLK2 = 1;
    bit_out <= s1 when CLK1 = 1;

  end block;

 end MIXED;
```

1.7.
PISO

This is the input shift register cell for the WFT processors. Data enters word parallel, and leaves bit serial. Input data is 23 bits numeric, one bit parity. Latch causes the input to be moved down into the serial path. Shift Right moves the data out serially. Shift down moves the data down one level in the register.

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: Parallel In, Serial Out Shift Register
--      FILENAME: piso
--      LANGUAGE: VHDL
--
--      ENTITY:
            entity PISO_CELL
             (P_IN, S_IN: in Z_BIT;
             CLK2,  CLK1 : in clk_signal;
             CLK2_NOT,  CLK1_NOT : in clk_signal := 1;
             P_SHIFT_DOWN, P_SHIFT_RIGHT, P_LATCH: in CONTROL;
             S_OUT: buffer BIT;
             P_OUT: buffer BIT) is

             assert (not (P_LATCH and P_SHIFT_RIGHT))
              report "LATCH_PISO AND SHIFT_RIGHT_PISO ARE BOTH HIGH"
             severity warning;

            end PISO_CELL;
--
--*************************************************************************/

architecture STRUCTURE of PISO_CELL is

-- this is a purely structural description of the PISO cell.

block

        component MSFF port(A: in Z_bit;
                CLK2, CLK2_NOT, CLK1, CLK1_NOT: in clk_signal;
                B: buffer bit);

        component T_GATE: port (X: in Z_bit;
                CLK: in clk_signal; Y: out bit);

        for all: MSFF use
         entity (MSFF)
         port map (bit_in => A,
                  CLK2 => CLK2, CLK2_NOT => CLK2_NOT,
                  CLK1 => CLK1, CLK1_NOT => CLK1_NOT,
                  bit_out =>B)
         body (MIXED_BODY);
        end for;

        for all: T_GATE use
```

```
            entity (T_GATE)
            port map (bit_in => convb-z(X),
                  clk => clk, bit_out => Y)
            body (BEHAVIOR);
          end for;

        signal  PARALLEL_IN : Z_bit;
        signal  SERIAL_IN : atomic LATCH_RESOLVE Z_BIT;

    begin

        T1: T_GATE port(P_IN, P_SHIFT_DOWN, PARALLEL_IN);
        M1: MSFF port(PARALLEL_IN, CLK2, CLK2_NOT, CLK1, CLK1_NOT, P_OUT);
        T1: T_GATE port(S_IN, P_SHIFT_RIGHT, SERIAL_IN);
        T1: T_GATE port(P_OUT, P_LATCH, SERIAL_IN);
        M2: MSFF port(SERIAL_IN, CLK2, CLK2_NOT, CLK1, CLK1_NOT, S_OUT);
    end block;

end STRUCTURE;
```

```
-- this is a mixed behavioral/structural description of the unit piso cell
-- note the open clk_not ports.

architecture MIXED of PISO_CELL is

    block

        component MSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            X: buffer bit);

        for all: MSFF use
          entity (MSFF)
        port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
                  CLK1 => CLK1, CLK1_NOT => open, bit_out => X)
        body (BEHAVIOR);
       end for;

      signal PARALLEL_IN, SERIAL_IN: in Z_BIT;

      begin

          P_FF: MSFF port (PARALLEL_IN, CLK2, CLK1, P_OUT);
          S_FF: MSFF port (SERIAL_IN, CLK2, CLK1, S_OUT);

      process(P_SHIFT_DOWN, P_IN)
        begin
           if (P_SHIFT_DOWN = '1') then
             enable P_IN;
         else
             disable P_IN;
         end if;

           if P_SHIFT_DOWN = '1' then
            PARALLEL_IN <= P_IN;
           end if;

      end process;

      process(P_SHIFT_RIGHT, P_LATCH, S_IN, P_OUT)
        begin
         if (P_SHIFT_RIGHT = '1' or P_LATCH = '1') then
             enable P_OUT, S_OUT;
         else
             disable P_OUT, S_OUT;
         end if;

        if (P_SHIFT_RIGHT = '1') then
            SERIAL_IN <= S_IN;
        elsif
            (P_LATCH = '1') then
              SERIAL_IN <= convb-z(P_OUT);
        else
             SERIAL_IN <= 'Z';
        end if;
```

```
    end process;

  end block;

end MIXED;
```

1.8. Adder-Subtractor

This is the structure of the adder-subtractor used in the WFT pre-adds and post-adds. It is made up of a combination logic adder/subtractor, CLK1 latches for the input bits, two MSFF's, to hold the CARRY and BORROW for the next data bits, and CLK2 latches for holding the result on the way to the next stage. The carry and reset msff latches are reset on the leading CLK2 latches. It should be noted that the inputs are inverted and inverted again at the outputs.

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: ADDER/SUBTRACTOR CELL
--      FILENAME: add.v
--      PROJECT: THESIS
--      LANGUAGE: VHDL
--
--      ENTITY:
        with WFTA_PACKAGE; use WFTA_PACKAGE;
              entity ADD_SUB_CELL
              ( BIT_X, BIT_Y : in bit;
                RST: in control;
                CLK2,  CLK1 : in clk_signal;
                CLK2_NOT,  CLK1_NOT : in clk_signal := 1;
                SUM, DIFF: buffer bit) is
               end ADD_SUB_CELL;
--
--      FUNCTION:
--
--
--*************************************************************************/

architecture PURE_STRUCTURE of ADD_SUB_CELL is

 PURE_STRUCTURE:
  block

   component ADD-SUB port(A, A_NOT, B, CY_IN, BY_IN: in bit;
                          SUM, DIFF, CY_OUT, BY_OUT: out bit;

    for RA1: ADD-SUM
       entity (ADD-SUB)
       port map (A => A, A_NOT => A_NOT, B => B, CY_IN => CY,
                 BY_IN => BR_IN, SUM => SUM, DIFF => DIFF,
                 CY_OUT => CY_OUT, BY_OUT => BR_OUT)
     body (LOGIC_STRUCTURE);
    end for;

       component RMSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            RST: in control, X: buffer bit);

       for all: RMSFF use
         entity (RMSFF)
       port map (bit_in => convb_z(A); CLK2 => CLK2, CLK2_NOT => open,
             CLK1 => CLK1, CLK1_NOT => open, RST_FF => RST,
```

```
                    bit_out => X)
        body (BEHAVIOR);
      end for;


    component LATCH port(A: in bit; CLK, CLK_NOT: in CLK_SIGNAL;
                         B: buffer bit);

    for all: LATCH
      entity (LATCH)
      port map (BIT_IN => A, CLK => CLK, CLK_NOT => open, BIT_OUT => B)
      body (BEHAVIOR);
    end for;

    component INVERTER port(A: in bit; B: out bit);

    for all: INVERTER
      entity (INVERTER)
      port map (open)
      body (<<LIBRARY>>);
    end for;

-- signals for real section

    signal S1, SA, SB, S_NOTA, S_CY, S_BY, S_SUM, S_DIFF, S_CY_OUT,
           S_BY_OUT:  bit;


    begin

                                  -- adder/subtractor
      RI1: INVERTER port(A_IN, S1);
      RL1: LATCH port (A_IN, CLK1, SA);
      RL2: LATCH port (B_IN, CLK1, SB);
      RL3: LATCH port (convb-z(S1), CLK1, S_NOTA);
      RA1: ADD_SUB port(SA, S_NOTA, SB, S_CY, S_BY, S_SUM, S_DIFF,
                        S_CY_OUT, S_BY_OUT);

      RFF1: R_MSFF port(convb-z(S_CY_OUT), CLK2, CLK1, RESET, S_CY);
      RFF2: R_MSFF port(convb-z(S_BY_OUT), CLK2, CLK1, RESET, B_CY);
      RL4: LATCH port(convb-z(S_SUM), CLK2, SUM);
      RL5: LATCH port(convb-z(S_DIFF), CLK2, DIFF);

  end block;

end PURE_STRUCTURE;
```

1.9.
Zero Multiplier

The following descriptions structurally describe the interconnections of the Lyons multipliers. Each description is essentially identical, which of outputs of the data Flip-Flops is used as the input to the Adder is the main difference The primary difference between the cells is what MSFF output tap the input to the adder comes from, and which control signals are used as inputs.

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: Zero Multiplier for the WFT processor
--      FILENAME: m0
--      LANGUAGE: VHDL
--
--      ENTITY:
        entity MULT_0
         (DATA_IN: in BIT;
         P_PROD_IN: in BIT;
         CLK2,  CLK1 : in clk_signal;
         CLK2_NOT,  CLK1_NOT : in clk_signal := 1;
         SIGN_EXT: in M-CONTROL;
         P_PROD_OUT, DATA_OUT: buffer BIT) is
        end MULT_0;

--
--      FUNCTION: This cell implements the 0 case for the modified
--                Lyons serial multiplier architecture.
--                There is no adder cell nor carry flip flop
--                used in this circuit.  It is primarily a shift
--                register.
--
--*************************************************************************/

architecture STRUCTURE of MULT_O is

block

        component MSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            X: buffer bit);

        for all: MSFF use
          entity (MSFF)
        port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
                CLK1 => CLK1, CLK1_NOT => open, bit_out => X)
        body (BEHAVIOR);
      end for;

    signal FF0_OUT, FF1_OUT, FF2_OUT: bit
           PROD_IN: Z_BIT;

    begin

      FF0: MSFF port(convb-z(DATA_IN), CLK2, CLK1, FF0_OUT);
```

```
        FF1: MSFF port(convb-z(FF0_OUT), CLK2, CLK1, FF1_OUT);

        FF2: MSFF port(convb-z(FF1_OUT), CLK2, CLK1, DATA_OUT);

        FF_PROD: MSFF port(convb-z(PROD_IN), CLK2, CLK1, P_PROD_OUT);

          PROD_IN <= convb-z(P_PROD_IN) when SIGN_EXT = '1'
                     else 'Z';

      end block;

end STRUCTURE;
```

1.10.
Plus One Multiplier

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: Plus one serial multiplier
--      FILENAME: m1
--      LANGUAGE: VHDL
--
--      ENTITY:

              entity MULTp1
                (DATA_IN: in BIT;
                P_PROD_IN: in BIT;
               CLK2,  CLK1 : in clk_signal;
                CLK2_NOT,  CLK1_NOT : in clk_signal := 1;
                RESET_0, SIGN_EXT: in M-CONTROL;
                P_PROD_OUT, DATA_OUT: buffer BIT) is
               end MULTp1;

--      FUNCTION: Plus one serial multiplier for the WFT
--                This cell needs the reset to 0 and the
--                sign extend control signals.  The input to
--                the adder comes from the output of the second
--                MSFF in the data chain.
--
--*************************************************************************/


architecture STRUCTURE of MULTp1 is

block

   component ADD-SUB port(A, A_NOT, B, CY_IN, BY_IN: in bit;
                          SUM, DIFF, CY_OUT, BY_OUT: out bit);

    for RA1: ADD-SUB use
       entity (ADD-SUB)
       port map (A => A, A_NOT => A_NOT, B => B, CY_IN => CY,
                 BY_IN => BR_IN, SUM => SUM, DIFF => DIFF,
                 CY_OUT => CY_OUT, BY_OUT => BR_OUT)
     body (LOGIC_STRUCTURE);
    end for;

       component RMSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            RST: in control, X: buffer bit);

       for all: RMSFF use
         entity (RMSFF)
       port map (bit_in => convb_z(A); CLK2 => CLK2, CLK2_NOT => open,
              CLK1 => CLK1, CLK1_NOT => open, RST_FF => RST,
                   bit_out => X)
     body (BEHAVIOR);
    end for;
```

```
        component MSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            X: buffer bit);

        for all: MSFF use
          entity (MSFF)
        port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
                  CLK1 => CLK1, CLK1_NOT => open, bit_out => X)
        body (BEHAVIOR);
       end for;

      signal FF0_OUT, FF1_OUT, FF2_OUT: bit
             PROD_IN: Z_BIT;
      begin

       DP0: MSFF port(convb-z(DATA_IN), CLK2,  CLK1, FF0_OUT);

       DP1: MSFF port(convb-z(FF0_OUT), CLK2,  CLK1, FF1_OUT);

       DP2: MSFF port(convb-z(FF1_OUT), CLK2,  CLK1, DATA_OUT);

       A1: ADDER port(FF1_OUT, CY_IN, P_PROD_IN, CARRY, SUM);

       FF_PROD: MSFF port(convb-z(SUM_IN), CLK2,  CLK1, P_PROD_OUT);

       FF_CARRY: R_MSFF port(convb-z(CARRY), CLK2,  CLK1, RESET_0, CY_IN);

       SUM_IN <= convb-z(SUM) when SIGN_EXT = '1'
                 else 'Z';

      end block;

end STRUCTURE;
```

1.11.
Plus Two Multiplier

```
--*****************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: plus two multiplier
--      FILENAME:m2
--      LANGUAGE: VHDL
--
--      ENTITY:

              entity MULTp2
                (DATA_IN: in BIT;
                P_PROD_IN: in BIT;
                CLK2,  CLK1 : in clk_signal;
                CLK2_NOT,  CLK1_NOT : in clk_signal := 1;
                RESET_0, RSTDC, SIGN_EXT: in M-CONTROL;
                P_PROD_OUT, DATA_OUT: buffer BIT) is
              end MULTp2;

--      FUNCTION: implements the +2 multiplier for the WFT
--              This cell takes the input to the adder from
--                the output of the last flip flop in the chain.
--                it also requires the signal rstdc which is
--                "anded" with the data path input.
--
--*****************************************************************************/

architecture STRUCTURE of MULTp2 is

block

   component ADD-SUB port(A, A_NOT, B, CY_IN, BY_IN: in bit;
                          SUM, DIFF, CY_OUT, BY_OUT: out bit;

    for RA1: ADD-SUB use
       entity (ADD-SUB)
       port map (A => A, A_NOT => A_NOT, B => B, CY_IN => CY,
                 BY_IN => BR_IN, SUM => SUM, DIFF => DIFF,
                 CY_OUT => CY_OUT, BY_OUT => BR_OUT)
      body (LOGIC_STRUCTURE);
    end for;

       component RMSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            RST: in control, X: buffer bit);

       for all: RMSFF use
         entity (RMSFF)
       port map (bit_in => convb_z(A); CLK2 => CLK2, CLK2_NOT => open,
              CLK1 => CLK1, CLK1_NOT => open, RST_FF => RST,
                  bit_out => X)
      body (BEHAVIOR);
     end for;
```

```
        component MSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            X: buffer bit);

        for all: MSFF use
          entity (MSFF)
        port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
                  CLK1 => CLK1, CLK1_NOT => open, bit_out => X)
        body (BEHAVIOR);
      end for;

    signal FF0_OUT, FF1_OUT, FF2_OUT: bit
           DATA_PATH_IN, SUM_IN: BIT;

    begin

      DP0: MSFF port(convb-z(DATA_IN), CLK2, CLK1, FF0_OUT);

      DP1: MSFF port(convb-z(FF0_OUT), CLK2, CLK1, FF1_OUT);

      DP2: MSFF port(convb-z(FF1_OUT), CLK2, CLK1, DATA_OUT);

      A1: ADDER port(DATA_PATH_IN, CY_IN, P_PROD_IN, CARRY, SUM);

      FF_PROD: MSFF port(convb-z(SUM_IN), CLK2, CLK1, P_PROD_OUT);

      FF_CARRY: R_MSFF port(convb-z(CARRY), CLK2, CLK1, RESET_0, CY_IN);

      SUM_IN <= convb-z(SUM) when SIGN_EXT = '1'
                else 'Z';

      DATA_PATH_IN <= (RSTDC and FF2_OUT);

    end block;

end STRUCTURE;
```

1.12.
Negative One Multiplier

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: Negative one multiplier cell
--      FILENAME: multn1.v
--      LANGUAGE: VHDL
--
--      ENTITY:

          entity MULTn1
          ( DATA_IN: in BIT;
            P_PROD_IN: in BIT;
              CLK2,  CLK1 : in clk_signal;
              CLK2_NOT,  CLK1_NOT : in clk_signal := 0;
            RESET_1, SIGN_EXT: in M-CONTROL;
            P_PROD_OUT, DATA_OUT: buffer BIT) is
           end MULTn1;
--
--      FUNCTION:  This is the negative 1 multiplier cell.  The carry in
--                 the negative cells are reset to one instead of zero
--                 as in the positive case.
--
--*************************************************************************/

architecture STRUCTURE of MULTn1 is

block

   component ADD-SUB port(A, A_NOT, B, CY_IN, BY_IN: in bit;
                          SUM, DIFF, CY_OUT, BY_OUT: out bit;

    for RA1: ADD-SUB use
       entity (ADD-SUB)
       port map (A => A, A_NOT => A_NOT, B => B, CY_IN => CY,
                 BY_IN => BR_IN, SUM => SUM, DIFF => DIFF,
                 CY_OUT => CY_OUT, BY_OUT => BR_OUT)
     body (LOGIC_STRUCTURE);
    end for;

       component MSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                           X: buffer bit);

       for all: MSFF use
         entity (MSFF)
       port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
              CLK1 => CLK1, CLK1_NOT => open, bit_out => X)
       body (BEHAVIOR);
      end for;


       component RHMSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                             RST_1: in M-CONTROL; X: buffer bit);
```

```
        for all: RHMSFF use
          entity (RHMSFF)
        port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
                  CLK1 => CLK1, CLK1_NOT => open,
                  RST_1 => RST_1, bit_out => X)
        body (BEHAVIOR);
      end for;

    signal FF0_OUT, FF1_OUT, FF2_OUT: bit
           ADD_IN, SUM: bit;

    begin

      DP0: MSFF port(convb-z(DATA_IN), CLK2, CLK2_NOT, CLK1,
                     CLK1_NOT, FF0_OUT);

      DP1: MSFF port(convb-z(FF0_OUT), CLK2, CLK2_NOT, CLK1,
                     CLK1_NOT, FF1_OUT);

      DP2: MSFF port(convb-z(FF1_OUT), CLK2, CLK2_NOT, CLK1,
                     CLK1_NOT, DATA_OUT);

      A1: ADDER port(ADD_IN, CY_IN, P_PROD_IN, CARRY, SUM);

      FF_PROD: MSFF port(convb-z(SUM_IN), CLK2, CLK2_NOT, CLK1,
                     CLK1_NOT, P_PROD_OUT);

      FF_CARRY: R_MSFF port( convb-z(CARRY), CLK2, CLK2_NOT, CLK1, CLK1_NOT,
                             RESET_1, CY_IN);

      SUM_IN <= convb-z(SUM) when SIGN_EXT = '1'
                else 'Z';

      ADD_IN <= not(FF1_OUT);

    end block;

end STRUCTURE;
```

1.13.
Negative Two Multiplier

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE:  Negative two multiplier
--      FILENAME: multn2.v
--      LANGUAGE: VHDL
--
--      ENTITY:

          entity MULTn2
          (  DATA_IN: in BIT;
             P_PROD_IN: in BIT;
               CLK2,  CLK1 : in clk_signal;
               CLK2_NOT,  CLK1_NOT : in clk_signal := 0;
             RESET_1, RSTDC, SIGN_EXT: in M-CONTROL;
             P_PROD_OUT, DATA_OUT: buffer BIT) is
           end MULTn2;
--
--*************************************************************************/


architecture STRUCTURE of MULTn2 is

block

   component ADD-SUB port(A, A_NOT, B, CY_IN, BY_IN: in bit;
                          SUM, DIFF, CY_OUT, BY_OUT: out bit;

    for RA1: ADD-SUB use
       entity (ADD-SUB)
       port map (A => A, A_NOT => A_NOT, B => B, CY_IN => CY,
                 BY_IN => BR_IN, SUM => SUM, DIFF => DIFF,
                 CY_OUT => CY_OUT, BY_OUT => BR_OUT)
      body (LOGIC_STRUCTURE);
    end for;

       component MSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                           X: buffer bit);

       for all: MSFF use
         entity (MSFF)
       port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
              CLK1 => CLK1, CLK1_NOT => open, bit_out => X)
      body (BEHAVIOR);
     end for;

       component RHMSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                             RST_1: in M-CONTROL; X: buffer bit);

       for all: RHMSFF use
         entity (RHMSFF)
       port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
```

```
                CLK1 => CLK1, CLK1_NOT => open,
                RST_1 => RST_1, bit_out => X)
          body (BEHAVIOR);
        end for;

      signal FF0_OUT, FF1_OUT, FF2_OUT: bit
            DATA_PATH_IN, SUM_IN: BIT;

      begin

        DP0: MSFF port(convb-z(DATA_IN), CLK2, CLK1, FF0_OUT);

        DP1: MSFF port(convb-z(FF0_OUT), CLK2, CLK1, FF1_OUT);

        DP2: MSFF port(convb-z(FF1_OUT), CLK2, CLK1, DATA_OUT);

        A1: ADDER port(DATA_PATH_IN, CY_IN, P_PROD_IN, CARRY, SUM);

        FF_PROD: MSFF port(convb-z(SUM_IN), CLK2, CLK1, P_PROD_OUT);

        FF_CARRY: R_MSFF port(convb-z(CARRY), CLK2, CLK1,
                                          RESET_0, CY_IN);

        SUM_IN <= convb-z(SUM) when SIGN_EXT = '1'
                  else 'Z';

        DATA_PATH_IN <= (RSTDC nand FF2_OUT);

      end block;

end STRUCTURE;
```

1.14.
Parity Round Cell

```
--*************************************************************************
--
--      DATE:  29 AUG 1985
--
--      TITLE: PARITY ROUND CELL
--      FILENAME: prcell.v
--      LANGUAGE: VHDL
--
--      ENTITY:
          entity PRCELL
          (  PR_IN: in bit;
           P_CALC, R_CALC, P_APPEND: in CONTROL;
             CLK2,  CLK1 : in clk_signal;
             CLK2_NOT,  CLK1_NOT : in clk_signal := 1;
           PR_OUT: buffer bit) is
          end PRCELL;
--
--      FUNCTION:  THIS CELL COMPUTES THE PARITY BIT AND ROUNDS THE RESUTL
--           OUT OF THE POST-ADDERS.
--
--*************************************************************************/

architecture mixed of prcell is

block

      component LATCH port(A: in Z_bit; CLK: in clk_signal;
                           X: buffer bit);

        for all: LATCH use
          entity (LATCH)
          port map (bit_in => A, CLK => CLK, CLK_NOT => open, bit_out => X)
          body (STRUCTURE);
        end for;

        component MSFF port(A: in Z_bit; CLK2, CLK1: in clk_signal;
                            X: buffer bit);

        for all: MSFF use
          entity (MSFF)
        port map (bit_in => A; CLK2 => CLK2, CLK2_NOT => open,
                CLK1 => CLK1, CLK1_NOT => open, bit_out => X)
        body (BEHAVIOR);
      end for;

   signal ROUND_AND, ROUND_OR, ROUND_OUT, IN_XOR, PARITY_XOR,
          PARITY_OR, PARITY_OUT: bit;

 begin
                                -- ROUNDING SECTION

L_IN: LATCH port( PR_IN, CLK1, BIT_IN);
    IN_XOR <= (BIT_IN xor ROUND_OUT);
```

```
    ROUND_AND <= (BIT_IN and ROUND_OUT);
    ROUND_OR <= (ROUND_AND or R_CALC);
RND_MSFF: MSFF port(ROUND_OR, CLK2, CLK1, ROUND_OUT);

                                -- PARITY SECTION

PARITY_XOR <= (IN_XOR or PARITY_OUT);
PARITY_OR <= (PARITY_XOR or P_CALC);
PAF_MSFF: MSFF port(PARITY_OR, CLK2, CLK1, PARITY_OUT);
    TO_OUT <= IN_XOR when P_APPEND = '1'
              else PARITY_OUT;
L_OUT: LATCH port(TO_OUT, BIT_PR_OUT);

end block;

end MIXED;
```

BLANK PAGE

Appendix 2

C Simulation Programs

This appendix contains the programs used to simulate the WFT16 processor. In addition, the binary-decimal and decimal-binary conversion programs are also included. The programs are listed in the order encountered in the pipeline: Control Sequencer, Column Controller, Pre_WFTA.c, Multiplier, Post Adders, and the conversion programs bin.c and form_16.c.

1.1. CS.c

```
/*************************************************************************
**      DATE:  15 AUG 1985
**
**      TITLE: Control Sequencer Simulation Program
**      FILENAME: cs.c
**      COORDINATOR: Jim Collins.
**      PROJECT: THESIS
**      FUNCTION:   simulates the control sequencer for the wfta.
**                  Requires the number of control cycles to
**                generate control signals for, and the scale factor
**                  of the input data.
**      FILES WRITTEN:  master_control: contains a time tagged control
**                      word for the wfta processor.
**     FILES INCLUDED:  sr.c:  A function which is used to evaluate
**                              the set reset (SRFF) behavior.
**
*************************************************************************/

#include <stdio.h>
#define clk_cycle 32            /* number of cycles in the counter*/
#include "sr.c"
main ()
{
 FILE *fp, *gp, *hp, *ip,  *fopen();
 int clk_count = 0;
 int clk;
 int i;
 int setpass = 0;
 int rstpass;
 int rszf;
 int tmp_pass;
 int tmp_zfill;
 int tmp_piso;
 int scale = 0;
 int pcal;
```

```
int rcal;
int tmp_rcal;
int cycles;

/* flag register which holds the control signals before they are
   written to the file  */

struct
{
  unsigned pre_bar : 1;
  unsigned inc : 1;
  unsigned load_rom : 1;
  unsigned par_rst : 1;
  unsigned r_calc : 1;
  unsigned p_calc : 1;
  unsigned p_append : 1;
  unsigned l_sipo : 1;
  unsigned sr_sipo : 1;
  unsigned sr_piso : 1;
  unsigned l_piso : 1;
  unsigned sd_sipo : 1;
  unsigned sd_piso : 1;
  unsigned mult_round : 1;
  unsigned zero_fill : 1;
  unsigned pass_out : 1;
  unsigned rst_add : 1;
  unsigned par_chk : 1;
  unsigned in_out: 1;
  unsigned up_in: 1;
} flags;


/* master slave flip flop structure, (MSFF), there are thirty two MSFFs
   in the ring counter     */

struct msff
{
  int clk2;
  int clk1;
}  ff[32], delayst, delayrst;

struct srff                /*set reset data structure*/
{
  int set;
  int reset;
  int out;
} shift_piso, zfill, pass;

hp = fopen("master_control", "w");  /*open the control file */

shift_piso.out = 1;
pass.out = 1;
delayst.clk1 = 1;
delayst.clk2 = 1;
delayrst.clk1 = 1;
```

```
  delayrst.clk2 = 1;
  zfill.out = 1;

  /*prompt for the number of clock cycles and the scale factor
    of the input data set                                    */


  printf(" HOW MANY CLOCK CYCLES DO YOU WANT TO SIMULATE?0);
  scanf("%d", &cycles);

again:
  printf(" WHAT IS THE SCALE FACTOR FOR THE INPUT DATA? 0);
  printf(" THE SCALE MUST BE BETWEEN 0 AND 70);
  scanf("%d",&scale);

  if ((scale > 7) | (scale < 0))
  {
    printf( "SCALE FACTOR IS NOT WITHIN RANGE, TRY AGAIN0);
    goto again;
  }

  printf("COMPUTING CONTROL SIGNALS FOR %d CLOCK CYCLES0,cycles);
  printf(" THE SCALE FACTOR IS %d0,scale);

  fprintf(hp,"%d0, cycles);

/*************************************************************************/

while (clk_count <=cycles)
{
  clk = clk_count % clk_cycle;  /* modulo 32 counter */

/* initialize the ring counter to simulate a bit entering
                                  on clock cycle 0     */
  if (clk == 0)
    ff[0].clk2 = 1;

  if (clk == 1)
    ff[0].clk2 = 0;

/* things which happen on clock 2*/

  for (i = 1; i<=31; i++)
    ff[i].clk2 = ff[i-1].clk1;

  delayrst.clk2 = rstpass;
  delayst.clk2 = setpass;

/* things that happen on clock 1 */

  for (i = 0; i<=31; i++)
    ff[i].clk1 = ff[i].clk2;

  delayst.clk1 = delayst.clk2;
  delayrst.clk1 = delayrst.clk2;
```

```
/* assignment of control signals: adaptive scaling algorithm uses
   the if - then construct to model the PLA and SRFF behavior   */

  if ((scale == 0) && (clk == 6))
    setpass = 1;
  else if ((scale == 1) && (clk == 7))
    setpass = 1;
  else if ((scale == 2) && (clk == 8))
    setpass = 1;
  else if ((scale == 3) && (clk == 9))
    setpass = 1;
  else if ((scale == 4) && (clk == 10))
    setpass = 1;
  else if ((scale == 5) && (clk == 11))
    setpass = 1;
  else if ((scale == 6) && (clk == 12))
    setpass = 1;
  else if ((scale == 7) && (clk == 11))
    setpass = 1;
  else
     setpass = 0;

  if ((scale == 0) && (clk == 29))
    rstpass = 1;
  else if ((scale == 1) && (clk == 29))
    rstpass = 1;
  else if ((scale == 2) && (clk == 29))
    rstpass = 1;
  else if ((scale == 3) && (clk == 30))
    rstpass = 1;
  else if ((scale == 4) && (clk == 31))
    rstpass = 1;
  else if ((scale == 5) && (clk == 0))
    rstpass = 1;
  else if ((scale == 6) && (clk == 1))
    rstpass = 1;
  else if ((scale == 7) && (clk == 1))
    rstpass = 1;
  else
    rstpass = 0;

/* call the set_reset function to evaluate any possible changes in
   the set and reset variables      */

  tmp_piso = set_reset(setpass, rstpass, shift_piso.out);
  shift_piso.out = tmp_piso;
  flags.sr_piso = shift_piso.out;

  if ((scale == 0) && (clk == 6))
    rszf = 1;
  else if ((scale == 1) && (clk == 7))
    rszf = 1;
  else if ((scale == 2) && (clk == 8))
    rszf = 1;
  else if ((scale == 3) && (clk == 9))
```

```
    rszf = 1;
  else if ((scale == 4) && (clk == 10))
    rszf = 1;
  else if ((scale == 5) && (clk == 10))
    rszf = 1;
  else if ((scale == 6) && (clk == 10))
    rszf = 1;
  else if ((scale == 7) && (clk == 10))
    rszf = 1;
  else
    rszf = 0;

  tmp_zfill = set_reset(ff[1].clk1, rszf, zfill.out);
  zfill.out = tmp_zfill;
  flags.zero_fill = zfill.out;

  tmp_pass = set_reset(setpass, rstpass, pass.out);
  pass.out = tmp_pass;
  flags.pass_out = pass.out;

/* sd_sipo and sd_piso both happen on alternatating clock cycles */

  if (clk%2 == 0)
    flags.sd_sipo = 1;
  else
    flags.sd_sipo = 0;

 if (clk%2 == 1)
    flags.sd_piso = 1;
  else
    flags.sd_piso = 0;

 * interval signals */

  if ((clk < 19) || (clk >= 28))
    flags.p_calc = 1;
  else
    flags.p_calc = 0;

  if ((clk < 19) || (clk >= 27))
    flags.r_calc = 1;
  else
    flags.r_calc = 0;

  if ((clk < 21) || (clk >= 29))
    flags.sr_sipo = 1;
  else
    flags.sr_sipo = 0;

 * pulse signals *

  flags.l_piso = ff[0].clk1;
  flags.l_sipo = ff[21].clk1;
  flags.par_chk = flags.sr_piso;
  flags.par_rst = flags.l_piso;
```

```
    flags.mult_round = ff[0].clk1;
    flags.rst_add = !ff[1].clk2;
    flags.p_append = ff[19].clk1;

/* print results to the file master_control */

    fprintf(hp, "%d0, clk_count);
    fprintf(hp,"%d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d %d0,
    flags.pass_out, flags.zero_fill, flags.par_chk,  flags.par_rst,
    flags.r_calc,   flags.p_calc,    flags.p_append, flags.mult_round,
    flags.inc,      flags.sr_piso,   flags.l_piso,   flags.sd_piso,
    flags.sd_sipo,  flags.l_sipo,    flags.sr_sipo,  flags.pre_bar,
    flags.rst_add,  flags.load_rom,  flags.in_out,   flags.up_in);

    clk_count += 1;
  }       /* end while */

}       /* end main */
```

2.1.

```
/*************************************************************************
**      DATE:  29 AUG 1985
**
**      TITLE: Column controller simulation program.
**      FILENAME: c_cntrl.c
**      COORDINATOR: Jim Collins.
**      PROJECT: WFT16 SIMULATION
**      USE:  This program generates the control files for the arithmetic
**            pipeline.  It recieves input from the file MASTER_CONTROL,
**            and outputs three files, preadd_cntrl, mult_contrl,
**            and postadd_cntrl.
**
*************************************************************************/

#include <stdio.h>
#define clk_cycle 32
main ()
{
 FILE *fp,  *hp, *ip, *jp, *fopen();
 int i, int clk, clk_count, clk_int = 0, cycles, rst_add;
 int c_word[20];

/* this is the structure which holds the control signals for all
   fourteen columns of the multiplier array                    */

 struct
 {
  unsigned reset_0  : 1;
  unsigned reset_1  : 1;
  unsigned rstdc    : 1;
  unsigned s_extend : 1;
 }  flags[14];

 struct msff              /*master-slave flip flop data structure */
 {
  int clk2;
  int clk1;
 } ;
 struct msff tmp, preadd_cntrl[4], mult_cntrl[42], postadd_cntrl[3];

/* the control pipeline is initially set to all ones, signals switch
   in response to a zero traveling throught the pipe, which
   happens every thirty-two clock-cycles.  */

 for ( i = 41; i >= 0; i--)
 {
  mult_cntrl[i].clk2 = 1;
  mult_cntrl[i].clk1 = 1;
 }

 for (i = 3; i >= 0; i--)
 {
  preadd_cntrl[i].clk2 = 1;
  preadd_cntrl[i].clk1 = 1;
```

```
  }

  for (i = 2; i >= 0; i--)
  {
   postadd_cntrl[i].clk2 = 1;
   postadd_cntrl[i].clk1 = 1;
  }
  tmp.clk2 = 1;
  tmp.clk1 = 1;


  fp = fopen("master_control", "r");  /* input word from controller */
  hp = fopen("preadd_cntrl","w");     /* control signals for multiplier column*/
ip = fopen("mult_cntrl","w");       /* control signals for multiplier column*/
  jp = fopen("postadd_cntrl","w");    /* control signals for post add column*/
  fscanf(fp,"%d",&cycles);

/* The first word in the file is used as to control the loop. */

while (clk_count < cycles)
{
  fscanf(fp,"%d", &clk_count);
  clk = clk_count % clk_cycle;
  clk_int = clk_int % clk_cycle;
  for (i = 0; i<= 19; i++)       /* read all 20 control signals which
                                    are sent out each clock cycle */
   fscanf(fp, "%d%",&c_word[i]);
                                          /* start execution of program */

   rst_add = c_word[16];                  /*the reset signal for the adder is
                                  in position 16 in the file          */
   preadd_cntrl[0].clk2 = rst_add;

   if (clk != clk_int)              /*check to ensure validity of the data */
   {
    printf( "clocks are not aligned! clk = %d clk_int = %d0,
          clk, clk_int);
    exit();
   }

/*  clock two events
                                          shifting operations*/

   preadd_cntrl[1].clk2 = preadd_cntrl[0].clk1;
   preadd_cntrl[2].clk2 = preadd_cntrl[1].clk1;
   preadd_cntrl[3].clk2 = preadd_cntrl[2].clk1;
   mult_cntrl[0].clk2   = preadd_cntrl[3].clk1;

   for (i = 40; i >= 0; i--)
    mult_cntrl[i+1].clk2 = mult_cntrl[i].clk1;

   postadd_cntrl[0].clk2 = mult_cntrl[41].clk1;
   postadd_cntrl[1].clk2 = postadd_cntrl[0].clk1;
   postadd_cntrl[2].clk2 = postadd_cntrl[1].clk1;
```

```
/* clock one events */

   for (i = 3; i >= 0; i--)
     preadd_cntrl[i].clk1 = preadd_cntrl[i].clk2 ;

   for ( i = 41; i >= 0; i--)
     mult_cntrl[i].clk1 = mult_cntrl[i].clk2;

   for (i = 2; i >= 0; i--)
     postadd_cntrl[i].clk1 = postadd_cntrl[i].clk2 ;

/* the multiplier signals are generated in sets of three  */

   for ( i = 0; i <= 13; i++)
   {
     flags[i].reset_0 = mult_cntrl[3*i + 1].clk2;
     flags[i].reset_1 = !(flags[i].reset_0);
   }

   for(i = 0; i <= 13; i++)
   {
     flags[i].rstdc = mult_cntrl[3*i + 1].clk1;
     flags[i].s_extend = !(mult_cntrl[3*i+1].clk1 & mult_cntrl[3*i+2].clk1);
   }

/* print the output files */

   fprintf(hp," %d", clk);
   fprintf(ip,"%d0, clk_count);
   fprintf(jp," %d", clk_count);

   for (i = 0; i <= 2; i++)
   {
     fprintf(hp," %d ", preadd_cntrl[i].clk2);
     fprintf(jp," %d ",postadd_cntrl[i].clk2);
   }

   flags[13].s_extend = 0;          /*no sign extensions of column thirteen*/
   for (i = 0; i <= 13; i++)
      fprintf(ip,"  %d %d %d %d 0 ,flags[i].reset_0,flags[i].reset_1,
                                  flags[i].rstdc, flags[i].s_extend);

 clk_int +=1;
 }                          /* end while */

}                           /* end main */
```

3.1.

```
/*************************************************************************
**
**
**      DATE: 12 NOV 1985
**
**      AUTHOR: Jim Collins
**      TITLE: Preadd pipeline simulation program
**      FILENAME: pre_wfta.c
**      PROJECT:  WFT16 Simulation
**      OPERATING SYSTEM: UNIX V 4.2
**      LANGUAGE: C
**      USE:    This program is the third in the series which model the
**              16-point winograd pipeline.  It follows the multiply.c
**              program.
**
**
**      FILES READ:
**      master_control:  control word for the processor per simulation
**                       cycle.
**      preadd_cntrl:  reset signal for the carry/borrow of the
**                     postadd column.
**      test_piso:      Problem set to be used to caluculate DFTs
**                        output of bin.c (decimal-binary conversion
**                        program).
**
**      FILES WRITTEN:
**
**      piso_out:         serial output of the piso.
**      zf_out:           output of zero fill cell.
**      preadd1_in:     input to the preadd column 1.
**      preadd2_in:     input to the second preadd column.
**      preadd3_in:     input to the second preadd column.
**      phi1_out:         output of the latch following the last adder.
**      to_mult:          input to the multiplier program.
**
**
**      FILES INCLUDED:
**
**      typedefin: structure declarations for the program.
**      fn_add.c: binary addition function.
**      sr.c:    evaluates the set reset function (SRFF).
**      declare:   type declarations for the program.
**
**
*************************************************************************/
#include <stdio.h>
#include "typedefin"
#include "fn_add.c"
#include "sr.c"
#include "declare"
#define clk_cycle 32  /*  16 point wfta cycle */
main()
{
```

```
qp = fopen("col3_out","w");
sp = fopen("col4_out","w");
rp = fopen("col5_out","w");
tp = fopen("col6_out","w");
up = fopen("col7_out","w");
vp = fopen("col8_out","w");
wp = fopen("col9_out","w");
yp = fopen("cola_out","w");
zp = fopen("colb_out","w");
pz = fopen("colc_out","w");
xp = fopen("cold_out","w");
zzp = fopen("mult_out","w");

fscanf(ap,"%d",&cycles);
while (clk_count <= cycles)
{
 fscanf(ap,"%d",&clk_count);  /* read master control word */
 for (i=0; i<=19; i++)
   fscanf(ap,"%d",&flags[i]);
   clk =(clk_count % clk_cycle);
   clk_int =(clk_int % clk_cycle);
   if (clk !=clk_int)
   { printf("clocks are not synchronized  %d0, clk);
       exit();
   }
    mult_round = flags[7];      /* rounding signal for input
                                   to the first column     */

    fscanf(bp, "%d", &clk_data);

/********************* ASSIGNMENTS TO THE MULTIPLIER **********************/


/********************* TO  COLUMN 1 *************************/

  for (i = 0; i <= 7; i++)
    if ( i <= 4)
    { (p00[i])->ff1clk2 =  phi1_latch[i];
      p00[i]->prod_in = 0; }
    else
    { (p00[i])->ff1clk2 =  phi1_latch[i+5];
      p00[i]->prod_in = 0; }

   p801->ff1clk2 = phi1_latch[5];
   p801->prod_in = mult_round;

   p901->ff1clk2 = phi1_latch[6];
   p901->prod_in = mult_round;

   pa0n->ff1clk2 = phi1_latch[7];
   pa0n->prod_in = mult_round;

   pb00->ff1clk2 = phi1_latch[8];
   pb00->prod_in = mult_round;
```

```
    pc0n->ff1clk2 = phi1_latch[9];
    pc0n->prod_in = mult_round;

    pd0n->ff1clk2 = phi1_latch[13];
    pd0n->prod_in = mult_round;

    pe0n->ff1clk2 = phi1_latch[14];
    pe0n->prod_in = mult_round;

    pf0n->ff1clk2 = phi1_latch[15];
    pf0n->prod_in = mult_round;

    pg0n->ff1clk2 = phi1_latch[16];
    pg0n->prod_in = mult_round;

    ph00->ff1clk2 = phi1_latch[17];
    ph00->prod_in = mult_round;

    fscanf(cp, "%d", &clk_data);
    for (i = 0; i <= 17; i++)
      fscanf(cp, "%d", &phi1_latch[i]);

/********************* TO COLUMN 1 *************************/

    for (i = 0; i <= 7; i++)
    { (p10[i])->ff1clk2 = p00[i]->ff3clk1;
      p10[i]->prod_in = p00[i]->sumffclk1; }

    p81n->ff1clk2 = p801->ff3clk1;
    p81n->prod_in = p801->sumffclk1;

    p91n->ff1clk2 = p901->ff3clk1;
    p91n->prod_in = p901->sumffclk1;

    pa1n->ff1clk2 = pa0n->ff3clk1;
    pa1n->prod_in = pa0n->sumffclk1;

    pb11->ff1clk2 = pb00->ff3clk1;
    pb11->prod_in = pb00->sumffclk1;

    pc10->ff1clk2 = pc0n->ff3clk1;
    pc10->prod_in = pc0n->sumffclk1;

    pd11->ff1clk2 = pd0n->ff3clk1;
    pd11->prod_in = pd0n->sumffclk1;

    pe11->ff1clk2 = pe0n->ff3clk1;
    pe11->prod_in = pe0n->sumffclk1;

    pf12->ff1clk2 = pf0n->ff3clk1;
    pf12->prod_in = pf0n->sumffclk1;

    pg10->ff1clk2 = pg0n->ff3clk1;
    pg10->prod_in = pg0n->sumffclk1;
```

```
    ph1n->ff1clk2 = ph00->ff3clk1;
    ph1n->prod_in = ph00->sumffclk1;

********** TO COLUMN 2 **********/

    for (i = 0; i <= 7; i++)
    {  p20[i]->ff1clk2 = p10[i]->ff3clk1;
       p20[i]->prod_in = p10[i]->sumffclk1; }

    p821->ff1clk2 = p81n->ff3clk1;
    p821->prod_in = p81n->sumffclk1;

    p921->ff1clk2 = p91n->ff3clk1;
    p921->prod_in = p91n->sumffclk1;

    pa2n->ff1clk2 = pa1n->ff3clk1;
    pa2n->prod_in = pa1n->sumffclk1;

    pb2q->ff1clk2 = pb11->ff3clk1;
    pb2q->prod_in = pb11->sumffclk1;

    pc2n->ff1clk2 = pc10->ff3clk1;
    pc2n->prod_in = pc10->sumffclk1;

    pd2n->ff1clk2 = pd11->ff3clk1;
    pd2n->prod_in = pd11->sumffclk1;

    pe2n->ff1clk2 = pe11->ff3clk1;
    pe2n->prod_in = pe11->sumffclk1;

    pf20->ff1clk2 = pf12->ff3clk1;
    pf20->prod_in = pf12->sumffclk1;

    pg21->ff1clk2 = pg10->ff3clk1;
    pg21->prod_in = pg10->sumffclk1;

    ph22->ff1clk2 = ph1n->ff3clk1;
    ph22->prod_in = ph1n->sumffclk1;

********** TO COLUMN 3 **********/

    for (i = 0; i <= 7; i++)
    { p30[i]->ff1clk2 = p20[i]->ff3clk1;
      p30[i]->prod_in = p20[i]->sumffclk1; }

    p83n->ff1clk2 = p821->ff3clk1;
    p83n->prod_in = p821->sumffclk1;

    p93n->ff1clk2 = p921->ff3clk1;
    p93n->prod_in = p921->sumffclk1;

    pa31->ff1clk2 = pa2n->ff3clk1;
    pa31->prod_in = pa2n->sumffclk1;

    pb3n->ff1clk2 = pb2q->ff3clk1;
```

```
pb3n->prod_in = pb2q->sumffclk1;

pc3n->ff1clk2 = pc2n->ff3clk1;
pc3n->prod_in = pc2n->sumffclk1;

pd31->ff1clk2 = pd2n->ff3clk1;
pd31->prod_in = pd2n->sumffclk1;

pe31->ff1clk2 = pe2n->ff3clk1;
pe31->prod_in = pe2n->sumffclk1;

pf3q->ff1clk2 = pf20->ff3clk1;
pf3q->prod_in = pf20->sumffclk1;

pg31->ff1clk2 = pg21->ff3clk1;
pg31->prod_in = pg21->sumffclk1;

ph31->ff1clk2 = ph22->ff3clk1;
ph31->prod_in = ph22->sumffclk1;

/********************** TO  COLUMN 4 **************************/

for (i = 0; i <= 7; i++)
{ p40[i]->ff1clk2 = p30[i]->ff3clk1;
  p40[i]->prod_in = p30[i]->sumffclk1; }

p840->ff1clk2 = p83n->ff3clk1;
p840->prod_in = p83n->sumffclk1;

p940->ff1clk2 = p93n->ff3clk1;
p940->prod_in = p93n->sumffclk1;

pa4q->ff1clk2 = pa31->ff3clk1;
pa4q->prod_in = pa31->sumffclk1;

pb40->ff1clk2 = pb3n->ff3clk1;
pb40->prod_in = pb3n->sumffclk1;

pc41->ff1clk2 = pc3n->ff3clk1;
pc41->prod_in = pc3n->sumffclk1;

pd40->ff1clk2 = pd31->ff3clk1;
pd40->prod_in = pd31->sumffclk1;

pe40->ff1clk2 = pe31->ff3clk1;
pe40->prod_in = pe31->sumffclk1;

pf4n->ff1clk2 = pf3q->ff3clk1;
pf4n->prod_in = pf3q->sumffclk1;

pg4n->ff1clk2 = pg31->ff3clk1;
pg4n->prod_in = pg31->sumffclk1;

ph40->ff1clk2 = ph31->ff3clk1;
ph40->prod_in = ph31->sumffclk1;
```

```
/********************* TO  COLUMN 5 *************************/

  for (i = 0; i <= 7; i++)
  { p50[i]->ff1clk2 = p40[i]->ff3clk1;
    p50[i]->prod_in = p40[i]->sumffclk1; }

  p851->ff1clk2 = p840->ff3clk1;
  p851->prod_in = p840->sumffclk1;

  p951->ff1clk2 = p940->ff3clk1;
  p951->prod_in = p940->sumffclk1;

  pa50->ff1clk2 = pa4q->ff3clk1;
  pa50->prod_in = pa4q->sumffclk1;

  pb5n->ff1clk2 = pb40->ff3clk1;
  pb5n->prod_in = pb40->sumffclk1;

  pc50->ff1clk2 = pc41->ff3clk1;
  pc50->prod_in = pc41->sumffclk1;

  pd5n->ff1clk2 = pd40->ff3clk1;
  pd5n->prod_in = pd40->sumffclk1;

  pe5n->ff1clk2 = pe40->ff3clk1;
  pe5n->prod_in = pe40->sumffclk1;

  pf51->ff1clk2 = pf4n->ff3clk1;
  pf51->prod_in = pf4n->sumffclk1;

  pg50->ff1clk2 = pg4n->ff3clk1;
  pg50->prod_in = pg4n->sumffclk1;

  ph51->ff1clk2 = ph40->ff3clk1;
  ph51->prod_in = ph40->sumffclk1;

/********************* TO  COLUMN 6 *************************/

  for (i = 0; i <= 7; i++)
  { p60[i]->ff1clk2 = p50[i]->ff3clk1;
    p60[i]->prod_in = p50[i]->sumffclk1; }

  p861->ff1clk2 = p851->ff3clk1;
  p861->prod_in = p851->sumffclk1;

  p961->ff1clk2 = p951->ff3clk1;
  p961->prod_in = p951->sumffclk1;

  pa62->ff1clk2 = pa50->ff3clk1;
  pa62->prod_in = pa50->sumffclk1;

  pb6n->ff1clk2 = pb5n->ff3clk1;
  pb6n->prod_in = pb5n->sumffclk1;

  pc61->ff1clk2 = pc50->ff3clk1;
```

```
pc61->prod_in = pc50->sumffclk1;

pd6n->ff1clk2 = pd5n->ff3clk1;
pd6n->prod_in = pd5n->sumffclk1;

pe6n->ff1clk2 = pe5n->ff3clk1;
pe6n->prod_in = pe5n->sumffclk1;

pf6n->ff1clk2 = pf51->ff3clk1;
pf6n->prod_in = pf51->sumffclk1;

pg6n->ff1clk2 = pg50->ff3clk1;
pg6n->prod_in = pg50->sumffclk1;

ph61->ff1clk2 = ph51->ff3clk1;
ph61->prod_in = ph51->sumffclk1;

/********************** TO  COLUMN 7 **************************/

for (i = 0; i <= 7; i++)
{ p70[i]->ff1clk2 = p60[i]->ff3clk1;
  p70[i]->prod_in = p60[i]->sumffclk1; }

p870->ff1clk2 = p861->ff3clk1;
p870->prod_in = p861->sumffclk1;

p970->ff1clk2 = p961->ff3clk1;
p970->prod_in = p961->sumffclk1;

pa7n->ff1clk2 = pa62->ff3clk1;
pa7n->prod_in = pa62->sumffclk1;

pb70->ff1clk2 = pb6n->ff3clk1;
pb70->prod_in = pb6n->sumffclk1;

pc7n->ff1clk2 = pc61->ff3clk1;
pc7n->prod_in = pc61->sumffclk1;

pd70->ff1clk2 = pd6n->ff3clk1;
pd70->prod_in = pd6n->sumffclk1;

pe70->ff1clk2 = pe6n->ff3clk1;
pe70->prod_in = pe6n->sumffclk1;

pf70->ff1clk2 = pf6n->ff3clk1;
pf70->prod_in = pf6n->sumffclk1;

pg71->ff1clk2 = pg6n->ff3clk1;
pg71->prod_in = pg6n->sumffclk1;

ph70->ff1clk2 = ph61->ff3clk1;
ph70->prod_in = ph61->sumffclk1;

********************** TO  COLUMN 8 **************************
```

```
    for (i = 0; i <= 7; i++)
    { p80[i]->ff1clk2 = p70[i]->ff3clk1;
      p80[i]->prod_in = p70[i]->sumffclk1; }

    p880->ff1clk2 = p870->ff3clk1;
    p880->prod_in = p870->sumffclk1;

    p980->ff1clk2 = p970->ff3clk1;
    p980->prod_in = p970->sumffclk1;

    pa80->ff1clk2 = pa7n->ff3clk1;
    pa80->prod_in = pa7n->sumffclk1;

    pb82->ff1clk2 = pb70->ff3clk1;
    pb82->prod_in = pb70->sumffclk1;

    pc82->ff1clk2 = pc7n->ff3clk1;
    pc82->prod_in = pc7n->sumffclk1;

    pd80->ff1clk2 = pd70->ff3clk1;
    pd80->prod_in = pd70->sumffclk1;

    pe80->ff1clk2 = pe70->ff3clk1;
    pe80->prod_in = pe70->sumffclk1;

    pf82->ff1clk2 = pf70->ff3clk1;
    pf82->prod_in = pf70->sumffclk1;

    pg8q->ff1clk2 = pg71->ff3clk1;
    pg8q->prod_in = pg71->sumffclk1;

    ph8q->ff1clk2 = ph70->ff3clk1;
    ph8q->prod_in = ph70->sumffclk1;

/********************* TO  COLUMN 9 *************************/

    for (i = 0; i <= 7; i++)
    { p90[i]->ff1clk2 = p80[i]->ff3clk1;
      p90[i]->prod_in = p80[i]->sumffclk1; }

    p891->ff1clk2 = p880->ff3clk1;
    p891->prod_in = p880->sumffclk1;

    p991->ff1clk2 = p980->ff3clk1;
    p991->prod_in = p980->sumffclk1;

    pa92->ff1clk2 = pa80->ff3clk1;
    pa92->prod_in = pa80->sumffclk1;

    pb9q->ff1clk2 = pb82->ff3clk1;
    pb9q->prod_in = pb82->sumffclk1;

    pc91->ff1clk2 = pc82->ff3clk1;
    pc91->prod_in = pc82->sumffclk1;
```

```
pd9n->ff1clk2 = pd80->ff3clk1;
pd9n->prod_in = pd80->sumffclk1;

pe9n->ff1clk2 = pe80->ff3clk1;
pe9n->prod_in = pe80->sumffclk1;

pf9n->ff1clk2 = pf82->ff3clk1;
pf9n->prod_in = pf82->sumffclk1;

pg9n->ff1clk2 = pg8q->ff3clk1;
pg9n->prod_in = pg8q->sumffclk1;

ph92->ff1clk2 = ph8q->ff3clk1;
ph92->prod_in = ph8q->sumffclk1;

/********************** TO  COLUMN 10 ************************/

for (i = 0; i <= 7; i++)
{ pa0[i]->ff1clk2 = p90[i]->ff3clk1;
  pa0[i]->prod_in = p90[i]->sumffclk1; }

p8a1->ff1clk2 = p891->ff3clk1;
p8a1->prod_in = p891->sumffclk1;

p9a1->ff1clk2 = p991->ff3clk1;
p9a1->prod_in = p991->sumffclk1;

paa0->ff1clk2 = pa92->ff3clk1;
paa0->prod_in = pa92->sumffclk1;

pba0->ff1clk2 = pb9q->ff3clk1;
pba0->prod_in = pb9q->sumffclk1;

pca1->ff1clk2 = pc91->ff3clk1;
pca1->prod_in = pc91->sumffclk1;

pdan->ff1clk2 = pd9n->ff3clk1;
pdan->prod_in = pd9n->sumffclk1;

pean->ff1clk2 = pe9n->ff3clk1;
pean->prod_in = pe9n->sumffclk1;

pfa1->ff1clk2 = pf9n->ff3clk1;
pfa1->prod_in = pf9n->sumffclk1;

pgan->ff1clk2 = pg9n->ff3clk1;
pgan->prod_in = pg9n->sumffclk1;

pha0->ff1clk2 = ph92->ff3clk1;
pha0->prod_in = ph92->sumffclk1;

********************** TO  COLUMN 11 ************************

for (i = 0; i <= 7; i++)
{ pb0[i]->ff1clk2 = pa0[i]->ff3clk1;
```

```
    pb0[i]->prod_in = pa0[i]->sumffclk1; }

  p8bn->ff1clk2 = p8a1->ff3clk1;
  p8bn->prod_in = p8a1->sumffclk1;

  p9bn->ff1clk2 = p9a1->ff3clk1;
  p9bn->prod_in = p9a1->sumffclk1;

  pabq->ff1clk2 = paa0->ff3clk1;
  pabq->prod_in = paa0->sumffclk1;

  pbb1->ff1clk2 = pba0->ff3clk1;
  pbb1->prod_in = pba0->sumffclk1;

  pcbn->ff1clk2 = pca1->ff3clk1;
  pcbn->prod_in = pca1->sumffclk1;

  pdb1->ff1clk2 = pdan->ff3clk1;
  pdb1->prod_in = pdan->sumffclk1;

  peb1->ff1clk2 = pean->ff3clk1;
  peb1->prod_in = pean->sumffclk1;

  pfb1->ff1clk2 = pfa1->ff3clk1;
  pfb1->prod_in = pfa1->sumffclk1;

  pgb1->ff1clk2 = pgan->ff3clk1;
  pgb1->prod_in = pgan->sumffclk1;

  phbn->ff1clk2 = pha0->ff3clk1;
  phbn->prod_in = pha0->sumffclk1;

/********************** TO  COLUMN 12 ************************/

  for (i = 0; i <= 7; i++)
  { pc0[i]->ff1clk2 = pb0[i]->ff3clk1;
    pc0[i]->prod_in = pb0[i]->sumffclk1; }

  p8cn->ff1clk2 = p8bn->ff3clk1;
  p8cn->prod_in = p8bn->sumffclk1;

  p9cn->ff1clk2 = p9bn->ff3clk1;
  p9cn->prod_in = p9bn->sumffclk1;

  pac2->ff1clk2 = pabq->ff3clk1;
  pac2->prod_in = pabq->sumffclk1;

  pbc1->ff1clk2 = pbb1->ff3clk1;
  pbc1->prod_in = pbb1->sumffclk1;

  pcc2->ff1clk2 = pcbn->ff3clk1;
  pcc2->prod_in = pcbn->sumffclk1;

  pdc1->ff1clk2 = pdb1->ff3clk1;
  pdc1->prod_in = pdb1->sumffclk1;
```

```
    pec1->ff1clk2 = peb1->ff3clk1;
    pec1->prod_in = peb1->sumffclk1;

    pfc0->ff1clk2 = pfb1->ff3clk1;
    pfc0->prod_in = pfb1->sumffclk1;

    pgcq->ff1clk2 = pgb1->ff3clk1;
    pgcq->prod_in = pgb1->sumffclk1;

    phcn->ff1clk2 = phbn->ff3clk1;
    phcn->prod_in = phbn->sumffclk1;

/********************** TO  COLUMN 13 ************************/

    for (i = 0; i <= 6; i++)
    { pd1[i]->ff1clk2 = pc0[i]->ff3clk1;
      pd1[i]->prod_in = pc0[i]->sumffclk1; }

    p7dn->ff1clk2 = pc0[7]->ff3clk1;
    p7dn->prod_in = pc0[7]->sumffclk1;

    p8d1->ff1clk2 = p8cn->ff3clk1;
    p8d1->prod_in = p8cn->sumffclk1;

    p9d1->ff1clk2 = p9cn->ff3clk1;
    p9d1->prod_in = p9cn->sumffclk1;

    pad0->ff1clk2 = pac2->ff3clk1;
    pad0->prod_in = pac2->sumffclk1;

    pbd1->ff1clk2 = pbc1->ff3clk1;
    pbd1->prod_in = pbc1->sumffclk1;

    pcdn->ff1clk2 = pcc2->ff3clk1;
    pcdn->prod_in = pcc2->sumffclk1;

    pddn->ff1clk2 = pdc1->ff3clk1;
    pddn->prod_in = pdc1->sumffclk1;

    pedn->ff1clk2 = pec1->ff3clk1;
    pedn->prod_in = pec1->sumffclk1;

    pfdn->ff1clk2 = pfc0->ff3clk1;
    pfdn->prod_in = pfc0->sumffclk1;

    pgd1->ff1clk2 = pgcq->ff3clk1;
    pgd1->prod_in = pgcq->sumffclk1;

    phdn->ff1clk2 = phcn->ff3clk1;
    phdn->prod_in = phcn->sumffclk1;

*********************** COLUMN 0 OF  THE MULTIPLIERS ********************

    fscanf(bp, "%d%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);
```

```
    fprintf(zp,"   prod_in = %d ", ph00->prod_in); }

    for (i = 0; i <= 7; i++)
      m0(p00[i], sign_ext);

    m1(p801, reset_0, sign_ext);
    m1(p901, reset_0, sign_ext);
    n1(pa0n, reset_1, sign_ext);
    m0(pb00, sign_ext);
    n1(pc0n, reset_1, sign_ext);
    n1(pd0n, reset_1, sign_ext);
    n1(pe0n, reset_1, sign_ext);
    n1(pf0n, reset_1, sign_ext);
    n1(pg0n, reset_1, sign_ext);
    m0(ph00, sign_ext);

/********************** COLUMN 1 0F THE MULTIPLIER ************************/

/* read the input control signals for this column before calling the
   multiplier function                                          */

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p10[i], sign_ext);

    n1(p81n, reset_1, sign_ext);
    n1(p91n, reset_1, sign_ext);
    n1(pa1n, reset_1, sign_ext);
    m1(pb11, reset_0, sign_ext);
    m0(pc10, sign_ext);
    m1(pd11, reset_0, sign_ext);
    m1(pe11, reset_0, sign_ext);
    m2(pf12, reset_0, rstdc, sign_ext);
    m0(pg10, sign_ext);
    n1(ph1n, reset_1, sign_ext);


/********************** COLUMN 2 0F THE MULTIPLIER ************************/

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);



    for (i = 0; i <= 7; i++)
      m0(p20[i], sign_ext);

    m1(p821, reset_0, sign_ext);
    m1(p921, reset_0, sign_ext);
    n1(pa2n, reset_1, sign_ext);
    n2(pb2q, reset_1, rstdc, sign_ext);
    n1(pc2n, reset_1, sign_ext);
    n1(pd2n, reset_1, sign_ext);
    n1(pe2n, reset_1, sign_ext);
    m0(pf20, sign_ext);
```

```
    m1(pg21, reset_0, sign_ext);
    m2(ph22, reset_0, rstdc, sign_ext);

/********************** COLUMN 3 0F THE MULTIPLIER ***************************/

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p30[i], sign_ext);

    n1(p83n, reset_1, sign_ext);
    n1(p93n, reset_1, sign_ext);
    m1(pa31, reset_0, sign_ext);
    n1(pb3n, reset_1, sign_ext);
    n1(pc3n, reset_1, sign_ext);
    m1(pd31, reset_0, sign_ext);
    m1(pe31, reset_0, sign_ext);
    n2(pf3q, reset_1, rstdc, sign_ext);
    m1(pg31, reset_0, sign_ext);
    m1(ph31, reset_0, sign_ext);

    /********************** COLUMN 4 0F THE MULTIPLIER ***************************/

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p40[i], sign_ext);

m0(p840, sign_ext);
    m0(p940, sign_ext);
    n2(pa4q, reset_1, rstdc, sign_ext);
    m0(pb40, sign_ext);
    m1(pc41, reset_0, sign_ext);
    m0(pd40, sign_ext);
    m0(pe40, sign_ext);
    n1(pf4n, reset_1, sign_ext);
    n1(pg4n, reset_1, sign_ext);
    m0(ph40, sign_ext);

    /********************** COLUMN 5 0F THE MULTIPLIER ***************************

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p50[i], sign_ext);

    m1(p851, reset_0, sign_ext);
    m1(p951, reset_0, sign_ext);
    m0(pa50, sign_ext);
    n1(pb5n, reset_1, sign_ext);
```

```
m0(pc50, sign_ext);
n1(pd5n, reset_1, sign_ext);
n1(pe5n, reset_1, sign_ext);
m1(pf51, reset_0, sign_ext);
m0(pg50, sign_ext);
m1(ph51, reset_0, sign_ext);


/********************** COLUMN 6 0F THE MULTIPLIER ***************************/


fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(p60[i], sign_ext);

m1(p861, reset_0, sign_ext);
m1(p961, reset_0, sign_ext);
m2(pa62, reset_0, rstdc, sign_ext);
n1(pb6n, reset_1, sign_ext);
m1(pc61, reset_0, sign_ext);
n1(pd6n, reset_1, sign_ext);
n1(pe6n, reset_1, sign_ext);
n1(pf6n, reset_1, sign_ext);
n1(pg6n, reset_1, sign_ext);
m1(ph61, reset_0, sign_ext);


/********************** COLUMN 7 0F THE MULTIPLIER ***************************/

fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(p70[i], sign_ext);

m0(p870, sign_ext);
m0(p970, sign_ext);
n1(pa7n, reset_1, sign_ext);
m0(pb70, sign_ext);
n1(pc7n, reset_1, sign_ext);
m0(pd70, sign_ext);
m0(pe70, sign_ext);
m0(pf70, sign_ext);
m1(pg71, reset_0, sign_ext);
m0(ph70, sign_ext);


********************** COLUMN 8 0F THE MULTIPLIER ***************************
fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(p80[i], sign_ext);

m0(p880, sign_ext);
m0(p980, sign_ext);
```

```
    pec1->ff1clk2 = peb1->ff3clk1;
    pec1->prod_in = peb1->sumffclk1;

    pfc0->ff1clk2 = pfb1->ff3clk1;
    pfc0->prod_in = pfb1->sumffclk1;

    pgcq->ff1clk2 = pgb1->ff3clk1;
    pgcq->prod_in = pgb1->sumffclk1;

    phcn->ff1clk2 = phbn->ff3clk1;
    phcn->prod_in = phbn->sumffclk1;

/********************** TO  COLUMN 13 ************************/

    for (i = 0; i <= 6; i++)
    { pd1[i]->ff1clk2 = pc0[i]->ff3clk1;
      pd1[i]->prod_in = pc0[i]->sumffclk1; }

    p7dn->ff1clk2 = pc0[7]->ff3clk1;
    p7dn->prod_in = pc0[7]->sumffclk1;

    p8d1->ff1clk2 = p8cn->ff3clk1;
    p8d1->prod_in = p8cn->sumffclk1;

    p9d1->ff1clk2 = p9cn->ff3clk1;
    p9d1->prod_in = p9cn->sumffclk1;

    pad0->ff1clk2 = pac2->ff3clk1;
    pad0->prod_in = pac2->sumffclk1;

    pbd1->ff1clk2 = pbc1->ff3clk1;
    pbd1->prod_in = pbc1->sumffclk1;

    pcdn->ff1clk2 = pcc2->ff3clk1;
    pcdn->prod_in = pcc2->sumffclk1;

    pddn->ff1clk2 = pdc1->ff3clk1;
    pddn->prod_in = pdc1->sumffclk1;

    pedn->ff1clk2 = pec1->ff3clk1;
    pedn->prod_in = pec1->sumffclk1;

    pfdn->ff1clk2 = pfc0->ff3clk1;
    pfdn->prod_in = pfc0->sumffclk1;

    pgd1->ff1clk2 = pgcq->ff3clk1;
    pgd1->prod_in = pgcq->sumffclk1;

    phdn->ff1clk2 = phcn->ff3clk1;
    phdn->prod_in = phcn->sumffclk1;

 ********************** COLUMN 0 OF  THE MULTIPLIERS ********************

    fscanf(bp, "%d%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);
```

```
    fprintf(zp,"   prod_in = %d ", ph00->prod_in); }

    for (i = 0; i <= 7; i++)
      m0(p00[i], sign_ext);

    m1(p801, reset_0, sign_ext);
    m1(p901, reset_0, sign_ext);
    n1(pa0n, reset_1, sign_ext);
    m0(pb00, sign_ext);
    n1(pc0n, reset_1, sign_ext);
    n1(pd0n, reset_1, sign_ext);
    n1(pe0n, reset_1, sign_ext);
    n1(pf0n, reset_1, sign_ext);
    n1(pg0n, reset_1, sign_ext);
    m0(ph00, sign_ext);

/********************** COLUMN 1 0F THE MULTIPLIER ***************************/

/* read the input control signals for this column before calling the
   multiplier function                                          */

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p10[i], sign_ext);

    n1(p81n, reset_1, sign_ext);
    n1(p91n, reset_1, sign_ext);
    n1(pa1n, reset_1, sign_ext);
    m1(pb11, reset_0, sign_ext);
    m0(pc10, sign_ext);
    m1(pd11, reset_0, sign_ext);
    m1(pe11, reset_0, sign_ext);
    m2(pf12, reset_0, rstdc, sign_ext);
    m0(pg10, sign_ext);
    n1(ph1n, reset_1, sign_ext);


/********************** COLUMN 2 0F THE MULTIPLIER ***************************

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);



    for (i = 0; i <= 7; i++)
      m0(p20[i], sign_ext);

    m1(p821, reset_0, sign_ext);
    m1(p921, reset_0, sign_ext);
    n1(pa2n, reset_1, sign_ext);
    n2(pb2q, reset_1, rstdc, sign_ext);
    n1(pc2n, reset_1, sign_ext);
    n1(pd2n, reset_1, sign_ext);
    n1(pe2n, reset_1, sign_ext);
    m0(pf20, sign_ext);
```

```
    m1(pg21, reset_0, sign_ext);
    m2(ph22, reset_0, rstdc, sign_ext);

/********************** COLUMN 3 0F THE MULTIPLIER ***************************/

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p30[i], sign_ext);

    n1(p83n, reset_1, sign_ext);
    n1(p93n, reset_1, sign_ext);
    m1(pa31, reset_0, sign_ext);
    n1(pb3n, reset_1, sign_ext);
    n1(pc3n, reset_1, sign_ext);
    m1(pd31, reset_0, sign_ext);
    m1(pe31, reset_0, sign_ext);
    n2(pf3q, reset_1, rstdc, sign_ext);
    m1(pg31, reset_0, sign_ext);
    m1(ph31, reset_0, sign_ext);

    /********************** COLUMN 4 0F THE MULTIPLIER ***************************/

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p40[i], sign_ext);

m0(p840, sign_ext);
    m0(p940, sign_ext);
    n2(pa4q, reset_1, rstdc, sign_ext);
    m0(pb40, sign_ext);
    m1(pc41, reset_0, sign_ext);
    m0(pd40, sign_ext);
    m0(pe40, sign_ext);
    n1(pf4n, reset_1, sign_ext);
    n1(pg4n, reset_1, sign_ext);
    m0(ph40, sign_ext);

    ********************** COLUMN 5 0F THE MULTIPLIER ***************************

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(p50[i], sign_ext);

    m1(p851, reset_0, sign_ext);
    m1(p951, reset_0, sign_ext);
    m0(pa50, sign_ext);
    n1(pb5n, reset_1, sign_ext);
```

```
m0(pc50, sign_ext);
n1(pd5n, reset_1, sign_ext);
n1(pe5n, reset_1, sign_ext);
m1(pf51, reset_0, sign_ext);
m0(pg50, sign_ext);
m1(ph51, reset_0, sign_ext);


/********************** COLUMN 6 0F THE MULTIPLIER ***************************/


fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(p60[i], sign_ext);

m1(p861, reset_0, sign_ext);
m1(p961, reset_0, sign_ext);
m2(pa62, reset_0, rstdc, sign_ext);
n1(pb6n, reset_1, sign_ext);
m1(pc61, reset_0, sign_ext);
n1(pd6n, reset_1, sign_ext);
n1(pe6n, reset_1, sign_ext);
n1(pf6n, reset_1, sign_ext);
n1(pg6n, reset_1, sign_ext);
m1(ph61, reset_0, sign_ext);


/********************** COLUMN 7 0F THE MULTIPLIER ***************************/

fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(p70[i], sign_ext);

m0(p870, sign_ext);
m0(p970, sign_ext);
n1(pa7n, reset_1, sign_ext);
m0(pb70, sign_ext);
n1(pc7n, reset_1, sign_ext);
m0(pd70, sign_ext);
m0(pe70, sign_ext);
m0(pf70, sign_ext);
m1(pg71, reset_0, sign_ext);
m0(ph70, sign_ext);


/********************** COLUMN 8 0F THE MULTIPLIER ***************************/
fscanf(bp, "%d%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(p80[i], sign_ext);

m0(p880, sign_ext);
m0(p980, sign_ext);
```

```
m0(pa80, sign_ext);
m2(pb82, reset_0, rstdc, sign_ext);
m2(pc82, reset_0, rstdc, sign_ext);
m0(pd80, sign_ext);
m0(pe80, sign_ext);
m2(pf82, reset_0, rstdc, sign_ext);
n2(pg8q, reset_1, rstdc, sign_ext);
n2(ph8q, reset_1, rstdc, sign_ext);


/********************** COLUMN 9 0F THE MULTIPLIER **************************/


fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(p90[i], sign_ext);

m1(p891, reset_0, sign_ext);
m1(p991, reset_0, sign_ext);
m2(pa92, reset_0, rstdc, sign_ext);
n2(pb9q, reset_1, rstdc, sign_ext);
m1(pc91, reset_0, sign_ext);
n1(pd9n, reset_1, sign_ext);
n1(pe9n, reset_1, sign_ext);
n1(pf9n, reset_1, sign_ext);
n1(pg9n, reset_1, sign_ext);
m2(ph92, reset_0, rstdc, sign_ext);


/********************** COLUMN 10 0F THE MULTIPLIER **************************/

fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(pa0[i], sign_ext);

m1(p8a1, reset_0, sign_ext);
m1(p9a1, reset_0, sign_ext);
m0(paa0, sign_ext);
m0(pba0, sign_ext);
m1(pca1, reset_0, sign_ext);
n1(pdan, reset_1, sign_ext);
n1(pean, reset_1, sign_ext);
m1(pfa1, reset_0, sign_ext);
n1(pgan, reset_1, sign_ext);
m0(pha0, sign_ext);


 ********************** COLUMN 11 0F THE MULTIPLIER ***************************

fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

for (i = 0; i <= 7; i++)
  m0(pb0[i], sign_ext);
```

```
    n1(p8bn, reset_1, sign_ext);
    n1(p9bn, reset_1, sign_ext);
    n2(pabq, reset_1, rstdc, sign_ext);
    m1(pbb1, reset_0, sign_ext);
    n1(pcbn, reset_1, sign_ext);
    m1(pdb1, reset_0, sign_ext);
    m1(peb1, reset_0, sign_ext);
    m1(pfb1, reset_0, sign_ext);
    m1(pgb1, reset_0, sign_ext);
    n1(phbn, reset_1, sign_ext);


    /********************** COLUMN 12 0F THE MULTIPLIER ************************/

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 7; i++)
      m0(pc0[i], sign_ext);

    n1(p8cn, reset_1, sign_ext);
n1(p9cn, reset_1, sign_ext);
    m2(pac2, reset_0, rstdc, sign_ext);
    m1(pbc1, reset_0, sign_ext);
    m2(pcc2, reset_0, rstdc, sign_ext);
    m1(pdc1, reset_0, sign_ext);
    m1(pec1, reset_0, sign_ext);
    m0(pfc0, sign_ext);
    n2(pgcq, reset_1, rstdc, sign_ext);
n1(phcn, reset_1, sign_ext);


    /********************** COLUMN 13 0F THE MULTIPLIER ************************/

    fscanf(bp, "%d%d%d%d", &reset_0,&reset_1,&rstdc, &sign_ext);

    for (i = 0; i <= 6; i++)
      m1(pd1[i], reset_0, sign_ext);

    n1(p7dn, reset_1, sign_ext);
    m1(p8d1, reset_0, sign_ext);
    m1(p9d1, reset_0, sign_ext);
    m0(pad0, sign_ext);
    m1(pbd1, reset_0, sign_ext);
    n1(pcdn, reset_1, sign_ext);
    n1(pddn, reset_1, sign_ext);
    n1(pedn, reset_1, sign_ext);
    n1(pfdn, reset_1, sign_ext);
    m1(pgd1, reset_0, sign_ext);
    n1(phdn, reset_1, sign_ext);


            ********************** PRINT RESULTS ****************************

    if ( clk_count >= 39)
```

```
{ fprintf(np," %d ",clk_int);
  for(i =0; i <= 7; i++)
    fprintf(np,"%d ",(p00[i])->sumffclk1);
    fprintf(np," %d %d %d %d %d %d %d %d %d %d ", p801->sumffclk1,
        p901->sumffclk1, pa0n->sumffclk1, pb00->sumffclk1,
        pc0n->sumffclk1, pd0n->sumffclk1, pe0n->sumffclk1,
        pf0n->sumffclk1, pg0n->sumffclk1, ph00->sumffclk1); }

if ( clk_count >= 42)
{ fprintf(op," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(op,"%d ",(p10[i])->sumffclk1);
fprintf(op," %d %d %d %d %d %d %d %d %d %d ",  p81n->sumffclk1,
        p91n->sumffclk1, pa1n->sumffclk1, pb11->sumffclk1,
        pc10->sumffclk1, pd11->sumffclk1, pe11->sumffclk1,
        pf12->sumffclk1, pg10->sumffclk1, ph1n->sumffclk1); }

if ( clk_count >= 45)
{ fprintf(pp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(pp,"%d ",(p20[i])->sumffclk1);
fprintf(pp," %d %d %d %d %d %d %d %d %d %d ",  p821->sumffclk1,
        p921->sumffclk1, pa2n->sumffclk1, pb2q->sumffclk1,
        pc2n->sumffclk1, pd2n->sumffclk1, pe2n->sumffclk1,
        pf20->sumffclk1, pg21->sumffclk1, ph22->sumffclk1); }


if ( clk_count >= 48)
{ fprintf(qp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(qp,"%d ",(p30[i])->sumffclk1);
  fprintf(qp," %d %d %d %d %d %d %d %d %d %d ",  p83n->sumffclk1,
        p93n->sumffclk1, pa31->sumffclk1, pb3n->sumffclk1,
        pc3n->sumffclk1, pd31->sumffclk1, pe31->sumffclk1,
        pf3q->sumffclk1, pg31->sumffclk1, ph31->sumffclk1); }

if ( clk_count >= 51)
{ fprintf(sp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(sp,"%d ",(p40[i])->sumffclk1);
fprintf(sp," %d %d %d %d %d %d %d %d %d %d %d ",  p840->sumffclk1,
        p940->sumffclk1, pa4q->sumffclk1, pb40->sumffclk1,
        pc41->sumffclk1, pd40->sumffclk1, pe40->sumffclk1,
        pf4n->sumffclk1, pg4n->sumffclk1, ph40->sumffclk1); }

if ( clk_count >= 54)
{ fprintf(rp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(rp,"%d ",(p50[i])->sumffclk1);
  fprintf(rp," %d %d %d %d %d %d %d %d %d %d %d ",  p851->sumffclk1,
        p951->sumffclk1, pa50->sumffclk1, pb5n->sumffclk1,
        pc50->sumffclk1, pd5n->sumffclk1, pe5n->sumffclk1,
        pf51->sumffclk1, pg50->sumffclk1, ph51->sumffclk1); }
```

```
if ( clk_count >= 57)
{ fprintf(tp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(tp,"%d ",(p60[i])->sumffclk1);
fprintf(tp," %d %d %d %d %d %d %d %d %d %d ",  p861->sumffclk1,
        p961->sumffclk1, pa62->sumffclk1, pb6n->sumffclk1,
        pc61->sumffclk1, pd6n->sumffclk1, pe6n->sumffclk1,
        pf6n->sumffclk1, pg6n->sumffclk1, ph61->sumffclk1); }

if ( clk_count >= 60)
{ fprintf(up," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(up,"%d ",(p70[i])->sumffclk1);
fprintf(up," %d %d %d %d %d %d %d %d %d %d ",  p870->sumffclk1,
        p970->sumffclk1, pa7n->sumffclk1, pb70->sumffclk1,
        pc7n->sumffclk1, pd70->sumffclk1, pe70->sumffclk1,
        pf70->sumffclk1, pg71->sumffclk1, ph70->sumffclk1); }

if ( clk_count >= 63)
{ fprintf(vp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(vp,"%d ",(p80[i])->sumffclk1);
fprintf(vp," %d %d %d %d %d %d %d %d %d %d ",  p880->sumffclk1,
        p980->sumffclk1, pa80->sumffclk1, pb82->sumffclk1,
        pc82->sumffclk1, pd80->sumffclk1, pe80->sumffclk1,
        pf82->sumffclk1, pg8q->sumffclk1, ph8q->sumffclk1); }

if ( clk_count >= 66)
{ fprintf(wp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(wp,"%d ",(p90[i])->sumffclk1);
fprintf(wp," %d %d %d %d %d %d %d %d %d %d ",  p891->sumffclk1,
        p991->sumffclk1, pa92->sumffclk1, pb9q->sumffclk1,
        pc91->sumffclk1, pd9n->sumffclk1, pe9n->sumffclk1,
        pf9n->sumffclk1, pg9n->sumffclk1, ph92->sumffclk1); }

if ( clk_count >= 69)
{ fprintf(yp," %d ",clk_int);
  for(i =0; i <= 7; i++)
    fprintf(yp,"%d ",(pa0[i])->sumffclk1);
  fprintf(yp," %d %d %d %d %d %d %d %d %d %d ",  p8a1->sumffclk1,
        p9a1->sumffclk1, paa0->sumffclk1, pba0->sumffclk1,
        pca1->sumffclk1, pdan->sumffclk1, pean->sumffclk1,
        pfa1->sumffclk1, pgan->sumffclk1, pha0->sumffclk1); }

if ( clk_count >= 72)
{ fprintf(zp," %d ",clk_int);
for(i =0; i <= 7; i++)
  fprintf(zp,"%d ",(pb0[i])->sumffclk1);
  fprintf(zp," %d %d %d %d %d %d %d %d %d %d ",  p8bn->sumffclk1,
        p9bn->sumffclk1, pabq->sumffclk1, pbb1->sumffclk1,
        pcbn->sumffclk1, pdb1->sumffclk1, peb1->sumffclk1,
        pfb1->sumffclk1, pgb1->sumffclk1, phbn->sumffclk1); }

if ( clk_count >= 75)
```

```
        { fprintf(pz," %d ",clk_count);
        for(i =0; i <= 7; i++)
          fprintf(pz,"%d ",(pc0[i])->sumffclk1);
        fprintf(pz," %d %d %d %d %d %d %d %d %d %d ",  p8cn->sumffclk1,
                p9cn->sumffclk1, pac2->sumffclk1, pbc1->sumffclk1,
                pcc2->sumffclk1, pdc1->sumffclk1, pec1->sumffclk1,
                pfc0->sumffclk1, pgcq->sumffclk1, phcn->sumffclk1); }

        if ( clk_count >= 78)
        { fprintf(xp," %d ",clk_count);
          for(i =0; i <= 6; i++)
            fprintf(xp,"%d ",(pd1[i])->sumffclk1);
          fprintf(xp," %d %d %d %d %d %d %d %d %d %d %d %d ",p7dn->sumffclk1,
              p8d1->sumffclk1, p9d1->sumffclk1, pad0->sumffclk1,
                pbd1->sumffclk1, pcdn->sumffclk1, pddn->sumffclk1,
                pedn->sumffclk1, pfdn->sumffclk1, pgd1->sumffclk1,
                phdn->sumffclk1); }

          fprintf(zzp," %d ",clk_count);
          for(i =0; i <= 6; i++)
            fprintf(zzp,"%d ",(pd1[i])->sumffclk1);
          fprintf(zzp," %d %d %d %d %d %d %d %d %d %d %d %d ", p7dn->sumffclk1,
                  p8d1->sumffclk1, p9d1->sumffclk1, pad0->sumffclk1,
                  pbd1->sumffclk1, pcdn->sumffclk1, pddn->sumffclk1,
                  pedn->sumffclk1, pfdn->sumffclk1, pgd1->sumffclk1,
                  phdn->sumffclk1);

      clk_int +=1;


      }
      fclose(ap);
      fclose(bp);
      fclose(cp);
      fclose(np);
      fclose(op);
      fclose(pp);
      fclose(qp);
      fclose(pz);
      fclose(sp);
      fclose(tp);
      fclose(up);
      fclose(vp);
      fclose(wp);
      fclose(xp);
      fclose(yp);
      fclose(zp);
      fclose(zzp);
      }
```

_" . ds |n Functions"

5.1.1.

```
     /*************************************************************************
**      DATE:  29 AUG 1985
**
**      TITLE: multiplier functions
**      FILENAME: subm0.c, subm1.c, subm2.c subn1.c, subn2.c
**      COORDINATOR: Jim Collins.
**      PROJECT: WFT16 SIMULATION
**      USE: These are the functions called by the multiply.c program
**          to evaluate the bits in the data structure pointed to
**          by the position in the array in the same
**          fashion the hardware multipliers will do.
**
*************************************************************************/

/* zero multiplier cell */

        m0 (ptr, sign_ext)
        struct multX0 *ptr;
        int sign_ext;
        {
        if (sign_ext == 0)
          ptr->sumffclk2 = ptr->prod_in;
        else
          ptr->sumffclk2 = ptr->sumffclk2;

          ptr->ff2clk2  = ptr->ff1clk1;
          ptr->ff3clk2  = ptr->ff2clk1;

          ptr->ff1clk1  = ptr->ff1clk2;
          ptr->ff2clk1  = ptr->ff2clk2;
          ptr->ff3clk1  = ptr->ff3clk2;
          ptr->sumffclk1 = ptr->sumffclk2;

      return;
}
```

```
/* plus one multiply function */

        m1 (ptr, reset_0, sign_ext)
        struct multX1 *ptr;
        int reset_0, sign_ext;

        {
         int add = 0;

         if (reset_0 == 0)
         {
          ptr->carry_ffclk2 = 0;        /* reset the carry flip-flop to 0 */
          ptr->carry_ffclk1 = 0;
         }

        /* things that happen on clock two */

         add = ptr->ff2clk1 + ptr->prod_in + ptr->carry_ffclk1;

         switch(add)
         {
          case 0:
           ptr->tmpsum = 0;
           ptr->carry_ffclk2 = 0;
          break;

          case 1:
           ptr->tmpsum = 1;
           ptr->carry_ffclk2 = 0;
          break;

          case 2:
           ptr->tmpsum = 0;
           ptr->carry_ffclk2 = 1;
          break;

          case 3:
           ptr->tmpsum = 1;
           ptr->carry_ffclk2 = 1;
          break;

        }   /* end case */

         if (sign_ext != 1)
          ptr->sumffclk2 = ptr->tmpsum;

         ptr->ff2clk2 = ptr->ff1clk1;
         ptr->ff3clk2 = ptr->ff2clk1;

        * things that happen on clk1 */

         ptr->ff1clk1 = ptr->ff1clk2;
         ptr->ff2clk1 = ptr->ff2clk2;
         ptr->ff3clk1 = ptr->ff3clk2;
```

```
  ptr->carry_ffclk1 = ptr->carry_ffclk2;
  ptr->sumffclk1 = ptr->sumffclk2;

return;
}
```

```
/* plus two multiplier */

        m2(ptr, reset_0, rstdc, s_extend)

        struct multX2 *ptr;
        int  reset_0, rstdc, s_extend;
        {
        int add = 0;
        int addin = 0;

        if (reset_0 == 0)
        {
        ptr->carry_ffclk1 = 0;  /* reset the carry flip-flop to 0 */
        ptr->carry_ffclk2 = 0;  /* reset the carry flip-flop to 0 */
        }
        /* things that happen on clock two */

        addin = (ptr->ff3clk1 &&  rstdc);
        add = addin + ptr->prod_in + ptr->carry_ffclk1;

        switch(add)
        {
          case 0:
            ptr->tmpsum = 0;
            ptr->carry_ffclk2 = 0;
          break;

          case 1:
            ptr->tmpsum = 1;
            ptr->carry_ffclk2 = 0;
          break;

          case 2:
            ptr->tmpsum = 0;
            ptr->carry_ffclk2 = 1;
          break;

          case 3:
            ptr->tmpsum = 1;
            ptr->carry_ffclk2 = 1;
          break;

        }    /* end case */

          if (s_extend != 1)
            ptr->sumffclk2 = ptr->tmpsum;

          ptr->ff2clk2 = ptr->ff1clk1;
          ptr->ff3clk2 = ptr->ff2clk1;

         * things that happen on clk1 */

          ptr->ff1clk1 = ptr->ff1clk2;
          ptr->ff2clk1 = ptr->ff2clk2;
          ptr->ff3clk1 = ptr->ff3clk2;
```

```
  ptr->carry_ffclk1 = ptr->carry_ffclk2;
  ptr->sumffclk1 = ptr->sumffclk2;

return;
}
```

```
/* minus one multiplier function */

n1(ptr, reset_1, s_extend)
struct multN1 *ptr;
int  reset_1, s_extend;
{
 int add = 0;
 /* things that happen on clock two */

 if (reset_1 == 1)
 {
  ptr->carry_ffclk1 = 1;        /* reset the carry flip-flop to 0*/
  ptr->carry_ffclk2 = 1;
 }

 add = !(ptr->ff2clk1) + ptr->prod_in + ptr->carry_ffclk1;

  switch(add)
  {
   case 0:
    ptr->tmpsum = 0;
    ptr->carry_ffclk2 = 0;
   break;

   case 1:
    ptr->tmpsum = 1;
    ptr->carry_ffclk2 = 0;
   break;

   case 2:
    ptr->tmpsum = 0;
    ptr->carry_ffclk2 = 1;
   break;

   case 3:
    ptr->tmpsum = 1;
    ptr->carry_ffclk2 = 1;
   break;


 }   /* end case */

 if (s_extend != 1)
  ptr->sumffclk2 = ptr->tmpsum;

 ptr->ff2clk2 = ptr->ff1clk1;
 ptr->ff3clk2 = ptr->ff2clk1;

  * things that happen on clk1 *


 ptr->ff1clk1 = ptr->ff1clk2;
 ptr->ff2clk1 = ptr->ff2clk2;
 ptr->ff3clk1 = ptr->ff3clk2;
```

```
    ptr->carry_ffclk1 = ptr->carry_ffclk2;
    ptr->sumffclk1 = ptr->sumffclk2;


   return;
  }
```

```
/*  minus two multiplier */

n2(ptr,reset_1, rstdc, s_extend)
struct multN2 *ptr;
int  reset_1, rstdc, s_extend;

{

int add = 0;
int addin = 0;

 if (reset_1 == 1)
 {
  ptr->carry_ffclk1 = 1;        /* reset the carry flip-flop to 0*/
  ptr->carry_ffclk2 = 1;        /* reset the carry flip-flop to 0*/
 }

 addin = !(ptr->ff3clk1 && rstdc);
 add = addin + ptr->prod_in + ptr->carry_ffclk1;

  switch(add)
  {
   case 0:
    ptr->tmpsum = 0;
    ptr->carry_ffclk2 = 0;
   break;

   case 1:
    ptr->tmpsum = 1;
    ptr->carry_ffclk2 = 0;
   break;

   case 2:
    ptr->tmpsum = 0;
    ptr->carry_ffclk2 = 1;
   break;

   case 3:
    ptr->tmpsum = 1;
    ptr->carry_ffclk2 = 1;
   break;

  }    * end case */

 if (s_extend != 1)
  ptr->sumffclk2 = ptr->tmpsum;

 ptr->ff2clk2 = ptr->ff1clk1;
 ptr->ff3clk2 = ptr->ff2clk1;

  * things that happen on clk1 */

 ptr->ff1clk1 = ptr->ff1clk2;
 ptr->ff2clk1 = ptr->ff2clk2;
 ptr->ff3clk1 = ptr->ff3clk2;
```

```
  ptr->carry_ffclk1 = ptr->carry_ffclk2;
  ptr->sumffclk1 = ptr->sumffclk2;

return;

}
```

6.1.

```
*************************************************************************/
**
**
**      DATE: 12 NOV 1985
**
**      AUTHOR: Jim Collins
**      FILENAME: post_wfta.c
**      PROJECT:  WFT16 Simulation
**      OPERATING SYSTEM: UNIX V 4.2
**      LANGUAGE: C
**      USE:    This program is the third in the series which model the
**              16-point winograd pipeline.  It follows the multiply.c
**              program.
**
**
**      FILES READ:
**      master_control:  control word for the processor per simulation
**                       cycle.
**      postadd_cntrl:  reset signal for the carry/borrow of the
**                      postadd column.
**      rmult_out:       output of the real pass through the serial
**                       multiplier, input to the real postadders.
**      imult_out:       output of the imaginary pass through the serial
**                       multiplier, input to the imaginary postadders.
**
**
**      FILES WRITTEN:
**
**      rpostadd1_in: input to the first columns of the real postadders.
**      rpostadd2_in: input to the second column of the real postadders.
**      rpostadd3_in: input to the third column of the real postadders.
**      rprcell_in:   input to the real parity round cell.
**      rprcell_out:  output of the real parity round cell.
**      rsipo_out:    output of real results.
**      ipostadd1_in: input to the second column of the imaginary postadders.
**      ipostadd2_in: input to the second column of the imaginary postadders.
**      ipostadd3_in: input to the third column of the  imaginary postadders.
**      iprcell_in:   input to the imaginary parity round cell.
**      isipo_out:    output of imaginary results.
**
**
**      FILES INCLUDED:
**                  fn_add.c: addition function
**                      postdec:  type and structure declarations for
**                                the program.
**
*************************************************************************
#include <stdio.h>
#include "fn_add.c"
#include "postdec"

#define clk_cycle 32  /*  16 point wfta cycle */
 typedef struct add_cell
```

```
 {
  int x_1;
  int y_1;
  int c_1;
  int b_1;
 };

 typedef struct add2_cell
 {
  int sum;
  int diff;
  int co;
  int bo;
 };

main()
{
 FILE *ap,*bp,*cp,*dp,*ep,*gp,*hp,*ip,*jp,*kp,*lp,*mp,
      *np, *op, *pp, *qp, *fopen();

 /* initialize the pointers for the adders*/

 for (i = 0; i <= 8; i++)
 { p_r_1i[i] = &r_col_1in[i];
   p_r_1o[i] = &r_col_1out[i];
   p_i_1i[i] = &i_col_1in[i];
   p_i_1o[i] = &i_col_1out[i]; }

 for (i = 0; i <= 3; i++)
 { p_r_2i[i] = &r_col_2in[i];
   p_r_2o[i] = &r_col_2out[i];
   p_i_2i[i] = &i_col_2in[i];
   p_i_2o[i] = &i_col_2out[i]; }

 for (i = 0; i <= 6; i++)
 { p_r_3i[i] = &r_col_3in[i];
   p_r_3o[i] = &r_col_3out[i];
   p_i_3i[i] = &i_col_3in[i];
   p_i_3o[i] = &i_col_3out[i]; }

 * open the files for the control words, and input data and output */

 ap = fopen("master_control", "r");
 bp = fopen("postadd_cntrl","r");
 cp = fopen("rmult_out", "r");
 dp = fopen("imult_out","r");
 hp = fopen("rpostadd1_in","w");
 jp = fopen("rpostadd2_in","w");
 ip = fopen("rpostadd3_in","w");
 kp = fopen("rprcell_in","w");
 gp = fopen("rprcell_out","w");
 lp = fopen("rsipo_out","w");
 np = fopen("ipostadd1_in","w");
 op = fopen("ipostadd2_in","w");
 pp = fopen("ipostadd3_in","w");
```

```
qp = fopen("iprcell_in","w");
mp = fopen("isipo_out","w");


fscanf(ap,"%d",&cycles);        /* number of full cycles which the program
                                   will simulate                          */
while (clk_count <= cycles)
{

 fscanf(ap,"%d",&clk_count);  /* read master control clock cycle */

 clk =(clk_count % clk_cycle);
 clk_int =(clk_int % clk_cycle);

/* check to see if internal and master clocks are sychronized */
 if (clk !=clk_int)
 {  printf(" clocks are not synchronized  %d0, clk);
    exit(); }

/* read all the control signals */

for (i=0; i<=19; i++)
 fscanf(ap,"%d",&flags[i]);

/* assign control variable to be used in this program */

 sr_sipo  = flags[14];
 l_sipo   = flags[13];
 sd_sipo  = flags[12];
 p_calc   = flags[5];
 p_append = flags[6];
 r_calc   = flags[4];

fscanf(bp,"%d",&clk_add);              /* read adder reset signals */
for (i = 0; i <= 2; i++)
 fscanf(bp,"%d",&rst_bit[i]);


********************** POST ADD MODULE 1 **********************************/


                              /*  call fn_add.c to add bit stream*/
for(i = 0; i <= 8; i++)
{
 add(p_r_1i[i], p_r_1o[i]);
 add(p_i_1i[i], p_i_1o[i]);
}

* The MSFFs in the postadder are reversed, data enters through phi_1 and
 leaves through phi_2, assign to output latch of the MSFF        *

for (i = 0; i <= 1; i++)
  { rpostadd[i].fclk2 = rpostadd[i].fclk1;
    ipostadd[i].fclk2 = ipostadd[i].fclk1;  }
```

```
/************************ POST ADD MODULE 2 ********************************/

 for(i = 0; i <= 3; i++)
 {
  add(p_r_2i[i], p_r_2o[i]);
  add(p_i_2i[i], p_i_2o[i]);
  }

                                /* seven MSFFs in the second column */
 for(i = 0; i <= 7; i++)
 {  rpostadd2[i].fclk2 = rpostadd2[i].fclk1;
    ipostadd2[i].fclk2 = ipostadd2[i].fclk1;  }

/************************ POST ADD MODULE 3 ********************************/

 for(i = 0; i <= 6; i++)        /* same as first column  */
 {
  add(p_r_3i[i], p_r_3o[i]);
  add(p_i_3i[i], p_i_3o[i]);
  }

 for(  i = 0; i <= 1; i++)
 {
  rpostadd3[i].fclk2 = rpostadd3[i].fclk1;
  ipostadd3[i].fclk2 = ipostadd3[i].fclk1;
 }

/************************ PARITY ROUND CELL ********************************/

 /* move bits through the parity round cell, both real and imaginary */

 for ( i = 0; i <= 15; i++)
 {
  (r_cell[i]).and_out = (r_cell[i]).clk1 & (r_cell[i]).r.fclk1;
  (r_cell[i]).r_or = (r_cell[i]).and_out | !r_calc;
  (r_cell[i]).in_xor = (r_cell[i]).clk1 ^ (r_cell[i]).r.fclk1;
  (r_cell[i]).p_xor = (r_cell[i]).in_xor ^ (r_cell[i]).p.fclk1;
  (r_cell[i]).p_or = (r_cell[i]).p_xor | !p_calc;
  (r_cell[i]).p.fclk2 = (r_cell[i]).p_or;
  (r_cell[i]).r.fclk2 = (r_cell[i]).r_or;

                                /* check control signals for parity cell*/
  if (p_append == 0)
    (r_cell[i]).clk2 = (r_cell[i]).in_xor;
  else
   (r_cell[i]).clk2 = (r_cell[i]).p.fclk1;

  (r_cell[i]).p.fclk1 = (r_cell[i]).p.fclk2;
  (r_cell[i]).r.fclk1 = (r_cell[i]).r.fclk2;

  (i_cell[i]).and_out = (i_cell[i]).clk1 & (i_cell[i]).r.fclk1;
  (i_cell[i]).r_or = (i_cell[i]).and_out | !r_calc;
  (i_cell[i]).in_xor = (i_cell[i]).clk1 ^ (i_cell[i]).r.fclk1;
  (i_cell[i]).p_xor = (i_cell[i]).in_xor ^ (i_cell[i]).p.fclk1;
  (i_cell[i]).p_or = (i_cell[i]).p_xor | !p_calc;
```

```
    (i_cell[i]).p.fclk2 = (i_cell[i]).p_or;
    (i_cell[i]).r.fclk2 = (i_cell[i]).r_or;

    if (p_append == 0)
      (i_cell[i]).clk2 = (i_cell[i]).in_xor;
    else
     (i_cell[i]).clk2 = (i_cell[i]).p.fclk1;

    (i_cell[i]).p.fclk1 = (i_cell[i]).p.fclk2;
    (i_cell[i]).r.fclk1 = (i_cell[i]).r.fclk2;
  }

/****************** SIPO CELL ****************************************/

/* shift the data right in the serial path */

     if (sr_sipo == 1)
     {
      for (i = 15; i >= 0; i--)
        for (j = 23; j >= 0; j--)
            if (j == 23)
            {  r_sipo[i][j].s_clk2 = r_phi1_latch[i];
               i_sipo[i][j].s_clk2 = i_phi1_latch[i];  }
            else
            {  r_sipo[i][j].s_clk2 = r_sipo[i][j+1].s_clk1;
               i_sipo[i][j].s_clk2 = i_sipo[i][j+1].s_clk1; }
            }

/*****************************************************************/

/* latch data from the serial path into the parallel path */

     if (l_sipo == 1)
     {
      for (i = 15; i >= 0; i--)
        for (j = 23; j >= 0; j--)
          {  r_sipo[i][j].p_clk2 = r_sipo[i][j].s_clk1;
             i_sipo[i][j].p_clk2 = i_sipo[i][j].s_clk1;  }
          }

 ******************************************************************

 * shift the data down in the parallel path */

     if (sd_sipo  == 1)
     {
      for (i = 15; i >= 0; i--)
        for (j = 23; j >= 0; j--)
          if (i == 15);
            else
            {  r_sipo[i][j].p_clk2 = r_sipo[i+1][j].p_clk1;
               i_sipo[i][j].p_clk2 = i_sipo[i+1][j].p_clk1;  }
     }

 ************* CLOCK ONE OCCURENCES ADD COLUMN ONE ***********
```

```
/* Assign input multiplier results to the x and y variables within
   the adder data structures indices on the left are inputs and outputs
   using the Taylor numbering system  */
   (p_r_1i[0])->x_1 = rmult[0];                  /* t00 h00 */
   (p_r_1i[0])->y_1 = rmult[1];                  /* t01 h08 */

   (p_r_1i[1])->x_1 = rmult[3];                  /* t03 t100 */
   (p_r_1i[1])->y_1 = rmult[8];                  /* t05 t101 */

   (p_r_1i[2])->x_1 = rmult[13];                 /* t13 t102 */
   (p_r_1i[2])->y_1 = rmult[6];                  /* t11 t103 */

   (p_r_1i[3])->x_1 = rmult[4];                  /* t04 t104 */
   (p_r_1i[3])->y_1 = rmult[9];                  /* t06 t105 */

   (p_r_1i[4])->x_1 = rmult[11];                 /* t08 t106 */
   (p_r_1i[4])->y_1 = rmult[10];                 /* t07 */

   (p_r_1i[5])->x_1 = rmult[12];                 /* t09 */
   (p_r_1i[5])->y_1 = rmult[10];                 /* t07 t107 */

   (p_r_1i[6])->x_1 = rmult[7];                  /* t12 t108 */
   (p_r_1i[6])->y_1 = rmult[14];                 /* t14 t109 */

   (p_r_1i[7])->x_1 = rmult[15];                 /* t15 t110 */
   (p_r_1i[7])->y_1 = rmult[16];                 /* t15  */

   (p_r_1i[8])->x_1 = rmult[15];                 /* t15 */
   (p_r_1i[8])->y_1 = rmult[17];                 /* t17  t110 */

   rpostadd[0].fclk1 = rmult[2];                 /* t02 */
   rpostadd[1].fclk1 = rmult[5];          /* t10 */

/********************** IMAGINARY SECTION ************************/


   (p_i_1i[0])->x_1 = imult[0];          /* u00 h00 */
   (p_i_1i[0])->y_1 = imult[1] ;                 /* u01 h08 */

   (p_i_1i[1])->x_1 = imult[3];          /* u03 u100 */
   (p_i_1i[1])->y_1 = imult[8];          /* u05 u101 */

   (p_i_1i[2])->x_1 = imult[13];                  * u13 u102 *
   (p_i_1i[2])->y_1 = imult[6];          /* u11 u103 *

   (p_i_1i[3])->x_1 = imult[4];          /* u04 u104 *
   (p_i_1i[3])->y_1 = imult[9];           * u06 u105 *

   (p_i_1i[4])->x_1 = imult[11];                  * u08 u106 *
   (p_i_1i[4])->y_1 = imult[10];                  * u07 *

   (p_i_1i[5])->x_1 = imult[12];                  * u12 *
   (p_i_1i[5])->y_1 = imult[10];                  * u07 u107 *

   (p_i_1i[6])->x_1 = imult[7];           * u12 u108 *
```

```
(p_j_1i[6])->y_1 = imult[14];                    /* u14 u109 */

(p_j_1i[7])->x_1 = imult[15];                    /* u15 u110 */
(p_j_1i[7])->y_1 = imult[16];                    /* u15  */

(p_j_1i[8])->x_1 = imult[15];                    /* u15 */
(p_j_1i[8])->y_1 = imult[17];                    /* u17  u110 */

ipostadd[0].fclk1 = imult[2];                    /* u02 */
ipostadd[1].fclk1 = imult[5];          /* u10 */

for (i = 0; i <= 8; i++)     /* move carry, borrow to CLK1 latches */
{ (p_r_1i[i])->c_1 = p_r_1o[i]->co;
  (p_r_1i[i])->b_1 = p_r_1o[i]->bo;
  (p_j_1i[i])->c_1 = p_j_1o[i]->co;
  (p_j_1i[i])->b_1 = p_j_1o[i]->bo; }

                                 /* if reset high reset the carry and borrow */
if (rst_bit[0] == 0)
   for (i = 0; i <= 8; i++)
   {(p_r_1o[i])->co  = 0;
    (p_r_1o[i])->bo  = 0;
    (p_r_1i[i])->c_1 = 0;
    (p_r_1i[i])->b_1 = 0;
    (p_j_1o[i])->co  = 0;
    (p_j_1o[i])->bo  = 0;
    (p_j_1i[i])->c_1 = 0;
    (p_j_1i[i])->b_1 = 0;}



fscanf(cp, "%d", &clk_real);
fscanf(dp, "%d", &clk_add);
for (i = 0; i <= 17; i++)              /* read input data from multiplier */
{  fscanf(cp,"%d",&rmult[i]);
   fscanf(dp,"%d",&imult[i]);
}

if (clk_count >= 79)                   /* first input not expected until
                                          clock 79                      */
{
 fprintf(hp," %d ",clk_count);  /* print real and imaginary inputs
                                       to the output files        */
 for ( i = 0; i <= 8; i++)
   fprintf(hp,"%d %d ",(p_r_1i[i])->x_1, (p_r_1i[i])->y_1);
fprintf(hp,"%d %d ", rpostadd[0].fclk1, rpostadd[1].fclk1);

fprintf(np," %d ",clk_count);
for ( i = 0; i <= 8; i++)
  fprintf(np,"%d %d ",(p_j_1i[i])->x_1, (p_j_1i[i])->y_1);
  fprintf(np,"%d %d ", ipostadd[0].fclk1, ipostadd[1].fclk1); }

************ CLOCK ONE OCCURENCES ADD COLUMN TWO ************

(p_r_2i[0])->x_1 = (p_r_1o[3])->sum;          /* t104 t200 */
```

```
(p_r_2i[0])->y_1 = (p_r_1o[4])->diff;        /* t106  t201 */
(p_r_2i[1])->x_1 = (p_r_1o[3])->diff;        /* t105 t202 */
(p_r_2i[1])->y_1 = (p_r_1o[5])->diff;        /* t107  t203 */
(p_r_2i[2])->x_1 = (p_r_1o[6])->sum;         /* t108 t204 */
(p_r_2i[2])->y_1 = (p_r_1o[7])->sum;         /* t110 t205 */
(p_r_2i[3])->x_1 = (p_r_1o[6])->diff;        /* t109 t206 */
(p_r_2i[3])->y_1 = (p_r_1o[8])->diff;        /* t111 t207 */

rpostadd2[0].fclk1 = rpostadd[0].fclk2;      /* t02 */
rpostadd2[1].fclk1 = (p_r_1o[0])->sum;       /* h0  */
rpostadd2[2].fclk1 = (p_r_1o[0])->diff;      /* h8  */
rpostadd2[3].fclk1 = (p_r_1o[1])->sum;       /* t100 */
rpostadd2[4].fclk1 = (p_r_1o[1])->diff;      /* t101 */
rpostadd2[5].fclk1 = rpostadd[1].fclk2;      /* t10 */
rpostadd2[6].fclk1 = (p_r_1o[2])->sum;       /* t102 */
rpostadd2[7].fclk1 = (p_r_1o[2])->diff;      /* t103 */
```

/********** IMAGINARY SECTION **********/

```
(p_i_2i[0])->x_1 = (p_i_1o[3])->sum;         /* t104 t200 */
(p_i_2i[0])->y_1 = (p_i_1o[4])->diff;        /* t106  t201 */
(p_i_2i[1])->x_1 = (p_i_1o[3])->diff;        /* t105 t202 */
(p_i_2i[1])->y_1 = (p_i_1o[5])->diff;        /* t107  t203 */
(p_i_2i[2])->x_1 = (p_i_1o[6])->sum;         /* t108 t204 */
(p_i_2i[2])->y_1 = (p_i_1o[7])->sum;         /* t110 t205 */
(p_i_2i[3])->x_1 = (p_i_1o[6])->diff;        /* t109 t206 */
(p_i_2i[3])->y_1 = (p_i_1o[8])->diff;        /* t111 t207 */

ipostadd2[0].fclk1 = ipostadd[0].fclk2;      /* t02 */
ipostadd2[1].fclk1 = (p_i_1o[0])->sum;       /* h0  */
ipostadd2[2].fclk1 = (p_i_1o[0])->diff;      /* h8  */
ipostadd2[3].fclk1 = (p_i_1o[1])->sum;       /* t100 */
ipostadd2[4].fclk1 = (p_i_1o[1])->diff;      /* t101 */
ipostadd2[5].fclk1 = ipostadd[1].fclk2;      /* t10 */
ipostadd2[6].fclk1 = (p_i_1o[2])->sum;       /* t102 */
ipostadd2[7].fclk1 = (p_i_1o[2])->diff;      /* t103 */

for (i = 0; i <= 3; i++)                     /* shift carry, borrow on
                                                phi_1 pulse          */
  { (p_r_2i[i])->c_1 = (p_r_2o[i])->co;
    (p_r_2i[i])->b_1 = (p_r_2o[i])->bo,
    (p_i_2i[i])->c_1 = (p_i_2o[i])->co;
    (p_i_2i[i])->b_1 = (p_i_2o[i])->bo; }

if (rst_bit[1] == 0) /* if reset high reset the carry and borrow *
  for (i = 0; i <= 3; i++)
  {(p_r_2o[i])->co  = 0;
   (p_r_2o[i])->bo  = 0;
   (p_r_2i[i])->c_1 = 0;
   (p_r_2i[i])->b_1 = 0;
   (p_i_2o[i])->co  = 0;
   (p_i_2o[i])->bo  = 0;
   (p_i_2i[i])->c_1 = 0;
```

```
    (p_i_2i[i])->b_1 = 0;}


  if (clk_count >= 80)          /* print real and imaginary inputs
                                   which are the outputs of columns two */
  {
   fprintf(jp," %d ",clk_count);
   for ( i = 0; i <= 3; i++)
    fprintf(jp,"%d %d ",(p_r_2i[i])->x_1, (p_r_2i[i])->y_1);
   for (i = 0; i <= 7; i++)
    fprintf(jp," %d", rpostadd2[i].fclk1);

   fprintf(op," %d ",clk_count);
   for ( i = 0; i <= 3; i++)
    fprintf(op,"%d %d ",(p_i_2i[i])->x_1, (p_i_2i[i])->y_1);
   for (i = 0; i <= 7; i++)
    fprintf(op," %d", ipostadd2[i].fclk1);
  }

/************ CLOCK ONE OCCURENCES ADD COLUMN THREE ******************/

/* assign output of column two adders both real and imaginary
   to the input of column 3 */

  (p_r_3i[0])->x_1 = (p_r_2o[0])->sum;      /* t200 */
  (p_r_3i[1])->x_1 = rpostadd2[3].fclk2;    /* t100 */
  (p_r_3i[2])->x_1 = (p_r_2o[1])->diff;     /* t203 */
  (p_r_3i[3])->x_1 = rpostadd2[0].fclk2;    /* t02 */
  (p_r_3i[4])->x_1 = (p_r_2o[1])->sum;      /* t202 */
  (p_r_3i[5])->x_1 = rpostadd2[4].fclk2;    /* t101 */
  (p_r_3i[6])->x_1 = (p_r_2o[0])->diff;     /* t201 */

 * in the real case, the imaginary term gets assigned to the y variable */

  (p_r_3i[0])->y_1 = (p_i_2o[2])->sum;      /* u204 */
  (p_r_3i[1])->y_1 =  ipostadd2[6].fclk2;   /* u102 */
  (p_r_3i[2])->y_1 = (p_i_2o[3])->diff;     /* u207 */
  (p_r_3i[3])->y_1 =  ipostadd2[5].fclk2;   /* u10 */
  (p_r_3i[4])->y_1 = (p_i_2o[3])->sum;      /* u206 */
  (p_r_3i[5])->y_1 =  ipostadd2[7].fclk2;   /* u103 */
  (p_r_3i[6])->y_1 = (p_i_2o[2])->diff;     /* u205 */

 ********************** IMAGINARY SECTION **********************


  (p_i_3i[0])->x_1 = (p_i_2o[0])->sum;
  (p_i_3i[1])->x_1 = ipostadd2[3].fclk2;
  (p_i_3i[2])->x_1 = (p_i_2o[1])->diff;
  (p_i_3i[3])->x_1 = ipostadd2[0].fclk2;
  (p_i_3i[4])->x_1 = (p_i_2o[1])->sum;
  (p_i_3i[5])->x_1 = ipostadd2[4].fclk2;
  (p_i_3i[6])->x_1 = (p_i_2o[0])->diff;

 * in the imaginary case, the real term is assigned to the x term */
```

```
(p_i_3i[0])->y_1 = (p_r_2o[2])->sum;
(p_i_3i[1])->y_1 =  rpostadd2[6].fclk2;
(p_i_3i[2])->y_1 = (p_r_2o[3])->diff;
(p_i_3i[3])->y_1 =  rpostadd2[5].fclk2;
(p_i_3i[4])->y_1 = (p_r_2o[3])->sum;
(p_i_3i[5])->y_1 =  rpostadd2[7].fclk2;
(p_i_3i[6])->y_1 = (p_r_2o[2])->diff;

rpostadd3[0].fclk1 = rpostadd2[1].fclk2;
rpostadd3[1].fclk1 = rpostadd2[2].fclk2;

ipostadd3[0].fclk1 = ipostadd2[1].fclk2;
ipostadd3[1].fclk1 = ipostadd2[2].fclk2;

for (i = 0; i <= 6; i++)
{ (p_r_3i[i])->c_1 = (p_r_3o[i])->co;
  (p_r_3i[i])->b_1 = (p_r_3o[i])->bo;
  (p_i_3i[i])->c_1 = (p_i_3o[i])->co;
  (p_i_3i[i])->b_1 = (p_i_3o[i])->bo; }


if (rst_bit[2] == 0)          /* reset carry/borrow */
{ for (i = 0; i <= 6; i++)
  { (p_r_3o[i])->co = 0;
    (p_r_3o[i])->bo = 0;
    (p_r_3i[i])->c_1 = 0;
    (p_r_3i[i])->b_1 = 0;
    (p_i_3o[i])->co = 0;
    (p_i_3o[i])->bo = 0;
    (p_i_3i[i])->c_1 = 0;
    (p_i_3i[i])->b_1 = 0; }}


if (clk_count >= 81)                  /* print results */
{
  fprintf(ip," %d ",clk_count);
  for ( i = 0; i <= 6; i++)
    fprintf(ip,"%d %d ",p_r_3i[i]->x_1, p_r_3i[i]->y_1);
  for (i = 0; i <= 1; i++)
    fprintf(ip,"%d ", rpostadd3[i].fclk1);

  fprintf(pp," %d ",clk_count);
  for ( i = 0; i <= 6; i++)
    fprintf(pp,"%d %d ",p_i_3i[i]->x_1, p_i_3i[i]->y_1);
  for (i = 0; i <= 1; i++)
    fprintf(pp,"%d ", ipostadd3[i].fclk1);
}

************************* PARITY ROUND CELL INPUT *************************

* assignments to the phi1_latch in the pr_cell */

r_cell[0].clk1 =  rpostadd3[0].fclk2;
r_cell[1].clk1 = (p_r_3o[0])->diff;
r_cell[2].clk1 = (p_r_3o[1])->diff;
```

```
    r_cell[3].clk1 = (p_r_3o[2])->sum;
    r_cell[4].clk1 = (p_r_3o[3])->diff;
    r_cell[5].clk1 = (p_r_3o[4])->diff;
    r_cell[6].clk1 = (p_r_3o[5])->diff;
    r_cell[7].clk1 = (p_r_3o[6])->sum;
    r_cell[8].clk1 =  rpostadd3[1].fclk2;
    r_cell[9].clk1 = (p_r_3o[6])->diff;
    r_cell[10].clk1 = (p_r_3o[5])->sum;
    r_cell[11].clk1 = (p_r_3o[4])->sum;
    r_cell[12].clk1 = (p_r_3o[3])->sum;
    r_cell[13].clk1 = (p_r_3o[2])->diff;
    r_cell[14].clk1 = (p_r_3o[1])->sum;
    r_cell[15].clk1 = (p_r_3o[0])->sum;

    i_cell[0].clk1 = ipostadd3[0].fclk2;
    i_cell[1].clk1 = (p_i_3o[0])->sum;
    i_cell[2].clk1 = (p_i_3o[1])->sum;
    i_cell[3].clk1 = (p_i_3o[2])->diff;
    i_cell[4].clk1 = (p_i_3o[3])->sum;
    i_cell[5].clk1 = (p_i_3o[4])->sum;
    i_cell[6].clk1 = (p_i_3o[5])->sum;
    i_cell[7].clk1 = (p_i_3o[6])->diff;
    i_cell[8].clk1 = ipostadd3[1].fclk2;
    i_cell[9].clk1 = (p_i_3o[6])->sum;
    i_cell[10].clk1 = (p_i_3o[5])->diff;
    i_cell[11].clk1 = (p_i_3o[4])->diff;
    i_cell[12].clk1 = (p_i_3o[3])->diff;
    i_cell[13].clk1 = (p_i_3o[2])->sum;
    i_cell[14].clk1 = (p_i_3o[1])->diff;
    i_cell[15].clk1 = (p_i_3o[0])->diff;

if (clk_count >= 82)
  {
   fprintf(qp, " %d", clk_count);
   fprintf(kp, " %d", clk_count);
   for (i = 0; i <= 15; i++)
   {
    fprintf (kp," %d ", (r_cell[i]).clk1);
    fprintf (qp," %d ", (i_cell[i]).clk1);
   }
  }

/********* PHI ONE LATCH BETWEEN PARITY AND SIPO CELL **********/

/* This latch returns the pipeline to its normal configuration.
  phi_2 leading, phi_1 trailing */

    for ( i = 0; i <= 15; i++)
    {  r_phi1_latch[i] = r_cell[i].clk2;
       i_phi1_latch[i] = i_cell[i].clk2;  }

    if (clk_count >= 83)
    {
     fprintf(gp," %d ", clk_count);
     for (i= 0; i <= 15; i++)
```

```
        fprintf(gp," %d ", r_phi1_latch[i]);
    }


/************************ SIPO CLOCK ONE OCCURENCES ********************/

/* shifting from phi_2 to phi_1 */

        for (i = 15; i >= 0; i--)
          for (j = 23; j >= 0; j--)
        { r_sipo[i][j].s_clk1 = r_sipo[i][j].s_clk2;
            r_sipo[i][j].p_clk1 = r_sipo[i][j].p_clk2;
          i_sipo[i][j].s_clk1 = i_sipo[i][j].s_clk2;
            i_sipo[i][j].p_clk1 = i_sipo[i][j].p_clk2; }


/*************************************************************************/

    if (clk_count >= 117)        /* write data out of SIPO */
    {
         fprintf(lp,"  ") ;
         fprintf(mp,"  ") ;
          for (j = 23; j >= 0; j--)
          { fprintf(lp," %d ",r_sipo[0][j].p_clk1);
            fprintf(mp," %d ",i_sipo[0][j].p_clk1); }
    }


  clk_int +=1;                   /* increment the internal counter */
 }
                /* end loop */
                /* close all files */
 fclose(ap);
 fclose(bp);
 fclose(cp);
 fclose(dp);
 fclose(ep);
 fclose(gp);
 fclose(hp);
 fclose(ep);
 fclose(kp);
 fclose(ip);
 fclose(lp);
 fclose(mp);
 fclose(np);
 fclose(op);
 fclose(pp);
 fclose(qp);
 }
```

+1 to Binary Conversion Program" . nr $0 2 " +1 to Binary Conversion Program

6.1.
Decimal

```
/*****************************************************************
**      DATE:  15 SEP 1985
**
**      TITLE: decimal to binary conversion program.
**      FILENAME: bin.c
**      COORDINATOR: Jim Collins.
**      PROJECT: WFT16 SIMULATION
**      USE: Converts decimal input numbers into their binary
**           representation.  Reads the input file wfta_in and
**           produces the binary output in file test_piso (the name
**           of the input file for the simulation program.
**
*****************************************************************/
main ()
{
FILE *fp, *gp, *fopen();
  int j, bit[24], mask;
  long num, k;
  int sum = 0;
  int x;

  fp = fopen("wfta_in","r");
  gp = fopen("test_piso","w");

  fscanf (fp,"%ld", &k);

  while (k != -1)
    {
      num = k;
      for (j = 23; j >= 0; j--)
          {
/* The numbers are converted using a shift and add function.  The masks,
   i.e. MASK00, are not included in this file for reasons of space.  */

          switch (j)
            {
              case 0:  bit[j] = (k & MASK00) >> j;  break;
              case 1:  bit[j] = (k & MASK01) >> j;  break;
              case 2:  bit[j] = (k & MASK02) >> j;  break;
              case 3:  bit[j] = (k & MASK03) >> j;  break;
              case 4:  bit[j] = (k & MASK04) >> j;  break;
              case 5:  bit[j] = (k & MASK05) >> j;  break;
              case 6:  bit[j] = (k & MASK06) >> j;  break;
              case 7:  bit[j] = (k & MASK07) >> j;  break;
              case 8:  bit[j] = (k & MASK08) >> j;  break;
              case 9:  bit[j] = (k & MASK09) >> j;  break;
              case 10:  bit[j] = (k & MASK10) >> j;  break;
              case 11:  bit[j] = (k & MASK11) >> j;  break;
              case 12:  bit[j] = (k & MASK12) >> j;  break;
              case 13:  bit[j] = (k & MASK13) >> j;  break;
              case 14:  bit[j] = (k & MASK14) >> j;  break;
```

```
                case 15:  bit[j] = (k & MASK15) >> j;  break;
                case 16:  bit[j] = (k & MASK16) >> j;  break;
                case 17:  bit[j] = (k & MASK17) >> j;  break;
                case 18:  bit[j] = (k & MASK18) >> j;  break;
                case 19:  bit[j] = (k & MASK19) >> j;  break;
                case 20:  bit[j] = (k & MASK20) >> j;  break;
                case 21:  bit[j] = (k & MASK21) >> j;  break;
                case 22:  bit[j] = (k & MASK22) >> j;  break;
                case 23:  bit[j] = (k & MASK23) >> j;  break;
                case 24:  bit[j] = (k & MASK24) >> j;  break;
                case 25:  bit[j] = (k & MASK25) >> j;  break;
                case 26:  bit[j] = (k & MASK26) >> j;  break;
                case 27:  bit[j] = (k & MASK27) >> j;  break;
                case 28:  bit[j] = (k & MASK28) >> j;  break;
                case 29:  bit[j] = (k & MASK29) >> j;  break;
                case 30:  bit[j] = (k & MASK30) >> j;  break;
                case 31:  bit[j] = (k & MASK31) >> j;  break;
            }     /* end switch */
        }         /* end for loop */

    sum = 0;
    for(j = 0; j <= 22; j++)
        sum = sum + bit[j];

 * odd parity requires that the number of ones in the data word be odd.
   In this case, if the number is even a one is appendend in the MSB
   position, zero  otherwise.                                    */

    if (sum%2 == 0)
      bit[23] = 1;
    else
        bit[23] = 0;

    for (j = 23; j >= 0; j--)
        fprintf (gp," %d", bit[j]);
    fprintf (gp,"0);
    fscanf (fp,"%ld", &k);        /* get next number */

  }             /* end while */
}               /* end main  */
```

7.1.
to

```
/*************************************************************************
**      DATE:  2 AUG 1985
**
**      TITLE: Binary to Decimal Conversion Program
**      FILENAME: form16.c
**      COORDINATOR: Jim Collins.
**      PROJECT: WFT16 SIMULATION
**      USE: This program takes the 16 serial outputs per clock
**          cycle, for any column, and converts it from a veritcal
**          format to a horizontal display.  It also converts the
**          binary output stream into a decimal number.  There is a
**          family of these programs, one for all possible number
**          of outputs for each column
**
*************************************************************************/
#include <stdio.h>
#define clk_cycles 32 /* number used within one file*/
main ()
{
 FILE *ap, *bp, *cp,  *fopen();

 int i, j, flag, cycles;
 int reformat[17][32];
 unsigned long sum;
 int count;

 ap = fopen("master_control", "r");
 bp = fopen("in_format", "r");
 cp = fopen("out_format", "w");

 fscanf(ap,"%d", &cycles);
 count = cycles / clk_cycles;
 while (count > 0)
 {
  for (j = 0; j <= 31; j++)
   for (i = 0; i <= 16 ; i++)
    fscanf(bp,"%d", &reformat[i][j]);

  for (i = 0; i <= 16; i++)
  {
   for (j = 0; j <= 31; j++)
    fprintf(cp," %d ", reformat[i][j]);
    fprintf(cp,"0);
  }

* convert the bit streams into decimal, if the MSB is a 1 the
 result is negative. the program handles this in the same
 manner as normal two's complement conversion.                *

for (i = 0; i <= 16; i++)
{ sum = 0;
  for (j = 0; j <= 31; j++)
  {              * the first entry is the clock tag  *
```

```
      if (i == 0);
      else if (reformat[i][31] == 1)
      {
        sum = sum + (!(reformat[i][j]) << j );
          flag = 1;}
     else
       {sum = sum + (reformat[i][j] << j );
       flag = 0;}
    }
    if (flag == 1)
    { sum = sum +1;
      printf(" [%d] = -%d", (i-1), sum);}
    else
      printf( " [%d] = %d 0, (i-1), sum);
  }
count = count - 1;
}                              /* end while */
}
```

Bibliography

1. Blahut, Richard E. *Fast Algorithms for Digital Signal Processing.* Reading, MA: Addison-Wesley Publishing Company, 1985.

2. Booch, Grady. *Software Engineering with Ada.* Menlo Park, CA: Benjamin Cummins Publishing Company, 1982.

3. Chu, Yaohan. "Why Do we need Hardware Description Languages?", *Computer*, pp 14-17 (1974).

4. Coutee, Paul W. *Arithmetic Circuitry for High Speed VLSI Winograd Fourier Transform Processors.* MS Thesis, GE ENG 85D-11. School of Engineering, Air Force Institute of Technology (AU) Wright-Patterson AFB OH, December 1985.

5. Intermetrics Inc. *VHDL User's Manual, Volume I,II* 1 August 1985, ASD (AFSC), Wright-Patterson AFB, OH. Contract F33615-83-C_1003.

6. Intermetrics, Inc. *VHDL Design, Analysis, and Justification. Version 7.2* 1 August 1985, ASD (AFSC), Wright-Patterson AFB OH. Contract F33615-83-C_1003.

7. Kernighan, B. W. and Lin, S. "An Efficient Heuristic Procedure for Partitioning Graphs." *Bell Systems Technical Journal* Vol 49 (2): pp 291 - 307 (1970).

8. Linderman, Richard W. *High Performance VLSI Technologies, Integrated Circuits, and Architectures for Digital Signal Processing.* PhD Thesis, Cornell University, Ithaca, NY, August 1984.

9. Lyons, R. F. "Two's Complement Pipeline Multiplier," *IEEE Transactions on Communications.* pp 418-425 (April 1977).

10. Liposki, J. H. "Hardware Description Languages, Voices from the Tower of Babel," *Computer* pp 14-17 (June 1977).

11. Moore, David H. and William J. Towle. "VHSIC's Industry Impact", *1981 University, Industry, Government Microelectronics Symposium* Starkville, Mississippi, pp I10 - I22 (1981).

12. Piloty, Robert and Dominique Borrione. "The Conlan Project: Concepts, Implementations, and Applications", *Computer*, Vol 18 (2): pp 81-92 (February 1985).

13. Rossbach, Paul C. *Control Circuitry for High Speed VLSI Winograd Fourier Transform Processors.* MS Thesis,GE/ENG/85D-35 School of Engineering, Air Force Institute of Technology (AU) Wright-Patterson AFB OH, December 1985.

14. Shadad, Moe, et al. "VHSIC Hardware Description Language",

Computer Vol 18 (2): pp 94-102 (February 1985).

15. Stremler, Ferrel G. *Introduction to Communications Systems (Second Edition).* Addsion-Welsey Publishing Company: Reading, MA, 1982.

16. Sumney, Larry W. "VHSIC, Universities, and Industry - Some Issues", *1981 University, Industry, Government Microelectronics Symposium* Starkville, Mississippi, pp I1 - I9 (1981).

17. Taylor, Kent. *Architecture and Numerical Accuracy of High-Speed DFT Processors.* MS Thesis, GE/ENG/85D-47. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH December 1985.

18. Winograd, Shmuel. "On Computing the DFT", *Proceedings, National Academy of Science.* Vol 73: pp 1005-1006 (1976).

Vita

Captain James M. Collins was born on 16 December, 1959 at Fort Monmouth, New Jersey. He attended Teheran American School in Teheran, Iran, and graduated from Norwich University with a Bachelor of Science Degree in Electrical Engineering in May 1981. He entered active duty in June, 1981 and was assigned to the Air Force Armament Laboratory, Eglin AFB, FL where he served as deputy program manager for the Global Positioning System tactical guidance program. He attended the University of West Florida and was awarded a Masters of Business Administration degree in April 1984. In May 1984 he entered the Air Force Institute of Technology to pursue a Masters degree in Electrical Engineering. He is a married to the former Ave Maria Bordenave and is a member of Eta Kappa Nu.

Permanent Address: 258 Tennessee Ave.
Valpariso Fl, 32580

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/CE/ENG/85D-9	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, OH 45433-6583	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFWAL 8b. OFFICE SYMBOL (If applicable) AADE	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code) Wright-Patterson AFB, OH 45433-6583	10. SOURCE OF FUNDING NOS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) See Box 19	
12. PERSONAL AUTHOR(S) James M. Collins, Capt, USAF	
13a. TYPE OF REPORT MS Thesis 13b. TIME COVERED FROM _____ TO _____	14. DATE OF REPORT (Yr., Mo., Day) 1985 December 15. PAGE COUNT 190
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES		
FIELD	GROUP	SUB GR
09	02	

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
Computerized Simulation Digital Simulation
Computer Architecture Signal Processing

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: Modeling and Simulation of a Signal Processor Implementing the Winograd Fourier Transform

Thesis Chairman: Richard W. Linderman, Captain, USAF
Assistant Professor of Electrical Engineering

Approved for public release IAW AFR 190-1.
Lynn Wolaver 16 JAN 86
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFSC)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☒ DTIC USERS ☐	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Richard W. Linderman	22b. TELEPHONE NUMBER (Include Area Code) 513-255-6913 22c. OFFICE SYMBOL AFIT/ENG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

Continuing advances in the state of the art silicon fabriaction technology have allowed tremendous increases in the performance which may be achieved by a single integrated circuit. The natural counterpart of this increased functionality is, of course, increased design complexity. A growing problem is how to concisely and accurately communicate design information on VLSI and VHSIC class circuits.

The VHSIC program office has sponsored the development of a hardware description language designed to address this problem. The VHSIC Hardware Description Language (VHDL) was applied to the problem of modeling a custom signal processor employing the Winograd Fourier Transform. A methodology was developed which decomposes the architecture into subcomponents, and then models the behavior and structure of the macrocells which which comprise those subcomponents. Additionaly, a custom simulator was developed to verify the timing, control, and hardware macrocells used in the implementation of the signal processor. The simulation modeled the circuit at the bit level and validated the architecture and expected numerical performance. (This)

END

FILMED

3 -86

DTIC