Ada Tools for the Description and Simulation of Digital Signal Processing Systems

1987

Mark D. Happel
University of Central Florida

Find similar works at: http://stars.library.ucf.edu/rtd
University of Central Florida Libraries http://library.ucf.edu

Part of the Systems and Communications Commons

STARS Citation

http://stars.library.ucf.edu/rtd/5031

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact lee.dotson@ucf.edu.
ADA TOOLS FOR THE DESCRIPTION AND SIMULATION
OF DIGITAL SIGNAL PROCESSING SYSTEMS

BY

MARK DAVID HAPPEL
B.S.E.E., United States Naval Academy, 1980

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in the Graduate Studies Program
of the College of Engineering
University of Central Florida
Orlando, Florida

Summer Term
1987
ABSTRACT

While specialized hardware description languages allow for maximum capability and efficiency in a design automation system, the use of a general purpose language in the same role can make the system more available or more practical for a larger set of users. This project demonstrates the use of ADA* for the description and simulation of small digital signal processing systems. Building on conventions and primitives proposed by Denyer and Renshaw, a simple subsystem was described in ADA and then tested with a small simulator also written in ADA.

* ADA is a trademark of the United States Department of Defense - ADA Joint Program Office (AJPO).
# TABLE OF CONTENTS

LIST OF FIGURES .................................................. V

I. INTRODUCTION ................................................... 1
   Hardware Descriptions ........................................ 2
   Toward a Common Language .................................... 3
   Recent Developments .......................................... 4
   An Alternate Approach ........................................ 5
   Statement of Thesis ........................................... 6

II. ALGORITHMIC VLSI DESIGN ................................. 8
    A Building Block Approach ................................ 8
    The Denyer and Renshaw Project ............................ 9

III. A SAMPLE ALGORITHM .......................................... 13

IV. BEHAVIORAL DESCRIPTION IN ADA ...................... 17
    Descriptive Levels and Abstraction ........................ 17
    Simulation Package Level .................................. 19
    Operator Description Level ................................ 19
    The Chip Level .............................................. 23

V. BEHAVIORAL SIMULATION IN ADA ....................... 24
    Simulator Structure ......................................... 24
    Scope of Simulation Variables ............................. 25
    Data Structures ............................................. 26
    Simulation Abstractions ................................... 27
    The Event Queue ............................................. 28
    Advancing Simulation Time ................................ 30
    Control Considerations ................................... 30

VI. CONCLUSIONS ................................................ 32
Appendices

A. SIMULATION PACKAGE .................................... 36
B. SIMULATOR ................................................. 60
C. SAMPLE SYSTEM ............................................ 63
D. PRIMITIVE OPERATORS .................................... 68
E. SUMMARY OF DESCRIPTION AND SIMULATION PROCEDURES ............................................ 69
F. THE EVENT-DRIVEN OPERATOR .......................... 71

REFERENCES .................................................. 73
LIST OF FIGURES

1. Digital Signal Processing Hierarchy .................. 10
2. Complex-to-Magnitude Signal Flow Graph ............ 15
3. Seven-Eighths Signal Flow Graph ..................... 16
CHAPTER I
INTRODUCTION

The last twenty years have seen the emphasis in electronic design shift from a low level, device oriented approach to a more systems oriented, building block style solution. By working with integrated circuits, a designer can view the proposed implementation from a more abstract level, concentrating more on behavioral requirements and less encumbered by low level details like biasing networks and temperature stabilization. Combined with the relatively low cost of mass-produced integrated circuits, this allows an individual designer the capability to design ever more powerful and complex systems.

This added power has not come without cost, however. The use of standard circuits tends to limit the flexibility of the designer (Hill and Peterson 1981), while the limitations on speed, efficiency and throughput of general purpose chips can render them unsuitable for demanding specialized applications (DeMan et al. 1986). Perhaps the most serious problem has been how to most efficiently manage the complexity of the ever more complicated systems being produced. This complexity issue has struck not only systems designers, but also those who develop the integrated circuits for use in these systems.
Specialized custom integrated circuits can eliminate some of the shortcomings of general purpose, off-the-shelf ICs, but the development cost of custom chips, usually somewhere between two and five million dollars, is prohibitive (Dewey and Gadient 1986). Much of this cost is driven by the enormous effort and time required to design and manage networks that can exceed, in the case of very large scale integration (VLSI) chips, 100,000 transistors. For example, 13,000 man-hours were required in the design of the Z8000 microprocessor (Rice 1980).

Hardware Descriptions

In order to deal with the complexity problem facing integrated circuit and system designers, structured design methodology and practices are now being applied to electronic design. Concepts of modularity, block-structuring, system hierarchies and structured descriptive techniques that have been utilized in large programming projects have been successfully applied to VLSI design (Lea 1986).

The manner in which digital logic circuits are represented has been strongly influenced by these structured design tools. In the 1950s, I.S. Reed developed a descriptive notation for bit transfers in sequential digital systems that became known as a register transfer language (Dietmeyer and Duley 1975) and has since been expanded to encompass a number of so-called hardware description languages. The hardware description languages now
available differ widely in syntax, features and capabilities (Aylor, Waxman and Scarratt 1986) and are used primarily to provide structured descriptions of hardware and to allow for the computer simulation of the description in order to confirm the intended behavior of the circuit at various levels (Dietmeyer and Duley 1975).

Toward a Common Language

Unfortunately, utilization of these descriptive languages has been hampered by the lack of a clear standard. Many of the languages have remained in purely academic applications, while the remainder tend to be the property of a particular manufacturer and not available to companies without the resources to develop their own software tools. Worse, the lack of a standard places a severe limitation of the interoperability of various computer aided engineering tools and environments (Dewey and Gadient 1986).

In response to the lack of standards, the Department of Defense (DOD) has undertaken a project to develop a standardized hardware description language as a part of its very high speed integrated circuit (VHSIC) program. This language has been named VHDL, for VHSIC hardware description language (Waxman 1986). The VHDL language is both an extension and a subset of the general purpose programming language ADA* (Wallace 1986), a language

* ADA is a trademark of the United States Department of Defense - ADA Joint Program Office (AJPO).
developed during the 1970s and early 1980s by the Department of Defense as a standard language for the programming of embedded computer systems (Cohen 1986).

One of the most controversial aspects of VHDL is the extent of the similarity between VHDL and ADA. During reviews of the VHDL standard, some have argued for the full inclusion of ADA in VHDL, while an equal number have fought against too much similarity. Those arguing for the inclusion of ADA maintained that "since VHDL will be implemented in ADA and the behavioral component will describe behavior, ADA seems the simplest and most robust means to provide structured hardware design". Those arguing against the inclusion of ADA felt that since VHDL is a hardware description language and not a general programming language, the inclusion of ADA would over-complicate the VHDL language, making learning the language overly difficult (Nash and Saunders 1986).

Recent Developments

Despite the potential standardization offered by VHDL and similar efforts, it is quite likely that the implementation of a single set of standard tools capable of the engineering of chips for all application areas should not be expected anytime in the near future. Efforts at the present time appear to be concentrating on systems that deal with only one application area in particular (DeMan et al. 1986).
Digital signal processing has been a popular applications area among design automation developers. Due to the highly specialized, complex algorithms and high data throughput rates, custom ICs have become a necessity in signal processing applications (DeMan 1986). A notable example of work in this area is the system developed by Denyer and Renshaw (1985) of the University of Edinburgh. This system, a so-called silicon compiler, takes a description of the desired behavior of the circuit and produces as output the various masks required for chip fabrication. The system also provides for verification of the design via simulation of the behavioral description.

The Denyer and Renshaw project utilizes a special purpose language specifically developed for that system. This language, "FIRST," (Denyer and Renshaw 1985) has been developed from the ground up with hardware description in mind. Some designers of hardware description languages feel that the similarity often observed between programming languages and description languages is unfortunate and the result of the lesser evolution of description languages compared with that of programming languages (Dietmeyer and Duley 1975).

An Alternate Approach

While special purpose languages can be optimized for an individual task or set of tasks, they also present inherent disadvantages. In order to use a new description language, one
must obviously become familiar with it. Perhaps not as obvious is the need for proper support for this language -- compilers, debuggers, development tools, etc. This additional required support adds to the total software overhead of any project, and may prevent the use of other tools or software support not written for that special purpose language.

There has been interest in and study of the use of general purpose programming languages to describe and simulate hardware. It has been suggested that strong typing, automatic memory management and polymorphic operators be features of languages to be used in this role (Ayres 1979). Certainly, the power and versatility of recent programming languages could prove useful for hardware description.

Statement of Thesis

The purpose of this research was to examine the potential of the ADA programming language for the description and subsequent simulation of bit-serial digital signal processing architectures. ADA was chosen for examination due to its versatility, standardization and the likelihood that the DOD sponsorship will make ADA a well supported and commonly used language. The selection of digital signal processing as the applications area makes possible a comparison between Denyer and Renshaw's language and ADA.
This investigation was carried out by encoding in ADA several of the fundamental architectural blocks, or "primitives," selected by Denyer and Renshaw. Once encoded, the primitives were collected, along with important simulation routines, into an ADA package. A simple simulator was written to verify a description of a DSP algorithm in terms of DSP primitives and other hierarchical blocks.

The final project step was to create the description of a modest digital signal processing subsystem to exercise the primitive and simulation routines. As an initial example, a complex number to magnitude conversion unit was chosen. This particular example has been demonstrated by Denyer and Renshaw using their specialized hardware description language, and therefore, made an interesting case study for the use of ADA (Denyer and Renshaw 1985).
This chapter briefly describes a standard cell approach to VLSI chip design used in recent silicon compilation projects. It focuses on the work of Denyer and Renshaw that is much of the basis behind this paper.

A Building Block Approach

In 1980, a text was published (Mead and Conway 1980) that has had a significant impact on both the present and future of VLSI design. In this text, Mead and Conway advocate a VLSI design technique of functional blocks interconnected to form larger systems (Lea 1986). Each of these functional blocks can be individually designed as a separate circuit and eventually reduced into integration masks. Once the circuit, or "leaf-cell," corresponding to a functional block has been designed, it can be used as a building block, with more complex integrated circuits being formed by the interconnection of the leaf-cells. Different VLSI chips can be developed by interconnecting various standard leaf-cells, without requiring the redesign of the leaf-cells themselves (Mead and Conway 1980).
This design style for VLSI chips has been labelled the "simplified full-custom leaf-cell design style" (Lea 1986) and has led to the development of standard cell blocks. The approach sacrifices the efficiency (in terms of silicon area and performance) of a fully optimized design in favor of a more structured hierarchical approach. This structured approach, similar to the top-down design methodology popular in large software projects (Fairley 1985) reduces the design complexity of large systems to a manageable level by allowing the designer to concentrate his efforts on one set of problems at a time, omitting complicating details of lower hierarchical levels. The similarity to software engineering methodology has made this approach popular among developers of silicon compilers (Lea 1986).

The Denyer and Renshaw Project

It is this standard cell approach that has been adopted by Denyer and Renshaw for their silicon compiler research. In this case, the standard cell is a generic bit-serial processing element. The use of bit-serial communications between the cells permits simplification of the networks connecting the cells, a potential problem in the chip layout (Denyer and Renshaw 1985).

The hierarchy used in this project is shown in Figure 1. All design levels below the leaf-cell level have been automated into a silicon compiler and are, therefore, not of concern to a designer.
Figure 1. Digital Signal Processing Hierarchy.
specifying a digital signal processing system. The DSP designer can view the leaf-cells as functional "black boxes."

A base set of leaf-cells, called primitives, was developed by Denyer and Renshaw and is included in this report as Appendix D. A description of a DSP system written in terms of these primitives could be input to a silicon compiler which would produce as output the masks for making integrated circuit chips to perform the DSP function. When processing the description, the silicon compiler would call routines corresponding to the individual primitives involved. These routines would draw the mask sections for the leaf-cell in the appropriate area of the chip (Denyer and Renshaw 1985).

In order to check the description before the time and expense of silicon compilation, the description should first be processed by a behavioral simulator. As with the silicon compiler, the simulator calls routines corresponding to the description primitives. However, in this case, the routines simulate the behavior of the primitives by processing sample test data and simulating propagation delays through the system. By comparing the simulation data with the desired output, design flaws can be detected prior to silicon implementation.

This thesis is concerned with the simulation of DSP system descriptions. The DSP descriptions have been developed to be consistent with Denyer and Renshaw's format and timing conventions (Denyer and Renshaw 1985). Specifically:
1. Data is fixed point, bit-serial, LSB first.

2. The primitives are operated synchronously from a master two-phase clock.

3. Control signals are delayed by an integer number of clock cycles via an associated control network to provide primitive synchronization.

4. Each operator possesses a fixed latency (propagation delay from input to output) which is also an integer number of clock cycles. (Since operation is bit-serial, with one bit of data entering or leaving a primitive during each clock cycle, the terms "bit" and "clock cycle" will be used interchangeably when referring to timing considerations.)

5. Each primitive operator has an optional, one bit input pre-delay in order to reduce the number of single bit delay leaf-cells that might otherwise be required to ensure that all necessary input data arrives simultaneously).
CHAPTER III
A SAMPLE ALGORITHM

The algorithm chosen to exercise the ADA behavioral description capabilities, as well as the operation of the accompanying simulator, is a relationship that approximates the magnitude of a complex number. This function, called the "four region approximation," calculates the magnitude, M, of a complex number, \(I + jQ\), by (Filip 1976):

\[
M = \text{MAX} \left( \left( \frac{7}{8} \right) |I| + \left( \frac{1}{2} \right) |Q| \right) \,
\left( \left( \frac{1}{2} \right) |I| + \left( \frac{7}{8} \right) |Q| \right) \,
\left( |Q| \right)
\]

Denyer and Renshaw proposed the following restatement of the four region approximation to improve its structure for implementation as:

\[
M = \text{MAX} \left( \sqrt{G} \right) \,
\left( \left( \frac{7}{8} \right) G + \left( \frac{1}{2} \right) L \right)
\]

where \(G = \text{max} \ (I, Q)\) and \(L = \text{min} \ (I, Q)\) (Denyer and Renshaw 1985).
A flow graph based on the above algorithm is shown in Figure 2. Note that the numbers within circles represent delay times for synchronization of data and control signals. The control signal C1 arrives simultaneously with the LSBs of the real and imaginary input data words. The notation C1-X refers to the C1 control signal delayed by X bits.

The arcs interconnecting the primitives are designated by an ID number in parentheses. Labels are also used where appropriate. The latency associated with each primitive is shown in brackets below the primitive name.

Note that "seven-eighths" is not a primitive as such, but is actually an operator made up of primitives in accordance with the hierarchy shown in Figure 1. The internal architecture of the seven-eighths operator is shown in Figure 3.
Figure 2. Complex-to-Magnitude Signal Flow Graph.
Figure 3. Seven-Eighths Signal Flow Graph.
CHAPTER IV
BEHAVIORAL DESCRIPTION IN ADA

This chapter details the conventions adopted to describe digital signal processing systems in ADA. In order to meet the perceived needs of a VLSI signal processing designer and to best utilize the capabilities of ADA, an abstract behavioral level of description was chosen. Each level of the descriptive hierarchy is discussed with respect to the sample implementation of a complex-to-magnitude algorithm in the appendices.

Descriptive Levels and Abstraction

The intent behind digital hardware description in a high level language is to permit the precise specification of the intended structure, capabilities, and/or functions of the system being described. This description could be an abstract algorithmic specification with no implication of actual hardware structure, a detailed gate-level description that obscures the system behavior, or something in between these two extremes. In addition, different hierarchies are possible, with higher levels using more powerful building blocks like adders and multipliers instead of the lower levels' combinatorial logic gates. Most hardware description languages provide the capability
of specifying systems at several different hierarchical levels and with varying degrees of behavioral abstraction (Aylor, Waxman and Scarratt 1986).

For the purpose of this project, it was decided to limit the description to levels at or above the primitive level of the hierarchy shown in Figure 1, with a fair degree of abstraction. The ultimate intent of a system such as this is to allow a system designer to specify an algorithmic description of the system in terms of primitives like multiply and absolute value, leaving the development of the details of lower levels to the automated silicon compilation process and thus freeing the DSP designer from the unnecessary details of low level digital synthesis. Thus, the designer can concentrate on the intended behavior and need not be a skilled digital designer or MOS circuit designer.

It is also true that the strong data-typing and verbosity that makes ADA and similar structured languages more readable and easily documented than unstructured languages also tends to clarify behavioral descriptions while lengthening and perhaps overly complicating structural descriptions. It has been noted that while the ADA-derived VHDL has been designed for use at various levels of behavioral abstraction, it has proven much more successful at behavioral description than at structural description (Nash and Saunders 1986).
Simulation Package Level

The abstraction of the description is strongest within the primitive themselves, as can be seen by examining the procedure "ABSOLUTE" in the ADA package "SIMPKG" detailed in Appendix A. In the middle of the execution body is the statement:

\[
\text{ARC (OUTPUT).DATA := ABS(ARC(INPUT).DATA)}
\]

which takes the absolute value of the input signal arc and assigns it to the output signal arc. While this could be accomplished in hardware with shift registers and complementing logic (Denyer and Renshaw 1985), the description here implies virtually nothing about the actual structure. The only clue to the nature of the hardware lies in the calculation of latency as the sum of a constant plus the system word length, implying internal temporary storage of the incoming serial bits, as well as an additional processing delay. If it were not for the need to include numerous simulation details within the primitive, the function of the unit would be quite obvious.

Operator Description Level

The level of abstraction is reduced somewhat in the next hierarchical level which involves the description of the complex-to-magnitude operator. This is the lowest level which the designer need deal with, as the primitives have been provided for
him as part of the package "SIMPKG." The level of involvement with the simulator has been minimized to just a requirement to place the actual description within a case statement, allowing conditional execution of primitive routines (see Appendix F).

The description at this level consists primarily of procedure calls to various primitive subroutines. This is similar to the syntax of several descriptive languages such as Denyer and Renshaw's FIRST, where the routines are behavioral primitives (Denyer and Renshaw 1985) and Hill and Peterson's AHPL, where the routines are general purpose MSI integrated circuits (Hill and Peterson 1981). In AHPL, the use of hardware primitives keeps the level of abstraction in a system description low, while the behavioral primitives of this paper maintain a fairly abstract operator description.

Nevertheless, there is more structure implied at the operator level than at the primitive level. While the description can be regarded as a structured verbal specification of the complex to magnitude algorithm flow graph, it also tends to imply a physical communication network between the primitive "black boxes."

Observing the complex-to-magnitude routine in Appendix C, it should be noted that the verbosity of ADA so useful for software readability can be used to improve hardware description. In this case, a feature known as named association, which binds subroutine formal parameters with the calling routine's actual parameters via the syntax (Department of Defense 1983):
Formal Parameter => Actual Parameter

allows the description:

\[
\text{Subtract (minuend} \Rightarrow 9, \text{ subtrahend} \Rightarrow 8, \text{ difference} \Rightarrow 12, \\
\text{Latency} \Rightarrow 1, \text{ Ctrl} \Rightarrow C1(12));
\]

instead of the shorter but less revealing description:

\[
\text{Subtract (9, 8, 12, 1, C1(12))};
\]

which requires examination of the primitive description or documentation to reveal the actual interconnections. The first three parameters (9, 8 and 12) are the numerical designations of the input and output arcs (type ARC_NO) satisfying the formal procedure specification:

\[
\text{Procedure Subtract (MINUEND, SUBTRAHEND, DIFFERENCE: ARC_NO;}
\]
\[
\text{CTRL: NATURAL; LATENCY: POSITIVE: = 1;}
\]
\[
\text{DEL1, DEL2: BIT: = 0});
\]

Significantly, the use of named association is also an optional (but recommended) feature of VHDL (Lipsett, Harschner and Shahdad 1986).

Also, in keeping with another VHDL example, defaults are provided for some of the parameters in the primitive routines.
This frees the designer from specifying connections to seldom used inputs or features, like the optional input pre-delay. While this is provided to hide unnecessary detail and streamline the description, it is recommended that it be used with caution as unintended omission of parameter specifications may result in operation other than that intended, with the root cause difficult to detect.

A useful feature of the description method developed here is that operators can be used as building blocks for more complex operators. Thus, a frequently occurring or useful operator, like complex-to-magnitude, may be used as a building block of an operator at the next higher hierarchical level, once again managing complexity by abstraction -- in this case, the inner workings of the complex-to-magnitude operator. Although it complicates the simulation somewhat, the added power of description was deemed well worth the additional trouble.

It should be noted that the control portion of the operator is separated from the data processing portion in keeping with the popular register transfer sequential machine model (Hill and Peterson 1981). This separation is emphasized by use of a block declaration for the control and a separate block declaration for the data unit. These block structures are provided solely to clarify the description and serve no useful purpose as far as simulation is concerned, though they may prove useful to leaf-cell or floorplan compilers operating on the same description.
The Chip Level

The highest level of description in the present project is the chip level, representing a single VLSI chip. This chip consists of the associated operators, as well as a master control generator for overall signal flow coordination and the input and output pads and buffers necessary to route data to and from the chip. While the Denyer and Renshaw system has a higher, multi-chip system hierarchy (Denyer and Renshaw 1985). As shown in Figure 1, limiting this project's description to single chips only was done to limit the scope of this work and is not perceived to be a limitation of ADA descriptions. It is felt that only minor changes would be necessary to the present simulator and primitives to allow for multiple chip systems.
CHAPTER V
BEHAVIORAL SIMULATION IN ADA

This chapter considers the simulation of the complex to magnitude description referred to in the previous chapter. The simulator was constrained by conventions adopted for hardware description, necessitating some programming practices normally avoided by software engineers and some compromises with simulation fidelity. The data structures, control signal implementation and basic operation of the simulator are discussed.

Simulator Structure

Compared to the effort required to establish conventions for hardware description in ADA, the construction of a simulator for the ADA description proved a much more formidable and time-consuming task. The problem was to develop a simulator, part of which would be a descriptive routine (the "chip" description) that ideally would contain little or no direct reference to simulation variables or requirements in order not to burden the designer with simulation details, and yet still be efficient enough to perform its functions in a reasonable time with reasonable resources. Although the simulator finally developed is rather limited and perhaps somewhat simplistic, the limitations were a result of the
scope of the project and, it is believed, could be expanded to a more sophisticated system without changing its basic structure or that of the hardware description discussed previously.

The simulator consists of a main routine "SIMULATE" (Appendix B), which calls the description "CHIP" (Appendix C) as a subroutine. The "SIMULATE" routine is actually little more than a user I/O interface, an initialization section and a loop containing the system description and a simulation time advancement routine. The "CHIP" routine is compiled separately and linked at run time, permitting the compiling and linking of different "chips" at run time without requiring recompilation of the main simulator routine. This maintains the general nature of the simulator.

Scope of Simulation Variables

As mentioned previously, it was considered desirable that as many simulation tasks to be performed by the system as possible. This led to many details being buried within the primitive routines that are called by the description routine. As these primitives are provided to the user via the package "SIMPKG," he is spared from simulation technicalities. Unfortunately, the placement of the description routine on a software level above the primitive routine level but below the simulator level made it impractical to pass simulation variables, such as simulation time and data structures, via specified formal parameters since the
simulation variables would have to be explicitly named in the parameter lists of the "system" call in the "simulate" main routine and the primitive calls in the system description. This limitation could only be overcome by extensive use of global parameters, a practice discouraged by software engineers due to susceptibility to accidental modification of variables or unforeseen side effects (Fairley 1985). ADA practitioners warn of obscured information passing and software maintenance difficulties (Cohen 1986). Despite their warnings, the use of global variables in this case appears warranted by the benefit of allowing the system designer to ignore the communication of simulation information between the simulator and the primitives. It is interesting to note that VHDL has been criticized for forcing the current simulator time to be passed via port (parametric) interconnections, hampering high level behavioral models that would profit from internal access to a global simulation time (Nash and Saunders 1986).

**Data Structures**

The fundamental data type in this project is a type defined in "SIMPKG" as type "signal." This is a record consisting of a floating-point data variable and an integer time variable. Thus, a signal becomes a two-dimensional vector tying sample data and time of occurrence. This is similar to the basic construction of another digital signal processing descriptive language, SILAGE,
utilized in the Cathedral-II Silicon Compiler Project (DeMan et al. 1986).

As can be seen in the listing of package "SIMPKG" in Appendix A, the type signal is used to define a global array of signal-type variables that is called the arc array. This is a listing of signals passing between primitives (nodes of a signal flow graph). By passing the index numbers of different elements of the arc array to the primitives as parameters of the primitive calls (in the hardware description), the arc array can be used as a source of input data for primitives as well as storage for output data. By specifying the same index to one primitive as output and another primitive as input, the two primitives are effectively joined together just as the nodes of a signal flow graph are joined by arcs.

**Simulation Abstractions**

It must be noted that some abstractions have been introduced into the simulation to improve throughput. Most significantly, the simulation primitives operate on parallel words (not bit-serial as in the actual hardware) that occur at the same time as the earliest (least significant) bit of the serial data word. By using parallel data in this fashion instead of one serial bit at each clock time, the primitive can process the entire word during the same clock (simulation) time and eliminate the need to call the primitive routine at every click of the simulation clock.
By adding adjustments to the primitive's internal algorithmic description, the same bit-fidelity as the actual bit-serial approach can be maintained while providing a significant increase in simulator throughput (Denyer and Renshaw 1985).

Another deviation from the actual hardware convention was the use of floating-point variables for the system data instead of the fixed point data in the actual hardware. Here, floating-point variables were used due to the ease with which ADA operates on floating-point variables compared to fixed-point quantities. Further work along the lines of this project should use fixed-point data to more accurately model the limitations of the hardware, though for this project it is not believed that the use of floating-point data significantly contaminated or invalidated the results.

The Event Queue

Due to the pipelined architectures of the signal processing systems under study, where new input data enters the system before the previous data has been completely processed in order to improve throughput, the arc array proved unsatisfactory as the sole means of storing the data on the arcs. Evaluation of timing diagrams for the system revealed that output data could be overwritten by later outputs before being input to the following primitive if the preceding primitive had a latency greater than
one word length. The use of multiple arc arrays and status bits was discarded as too cumbersome.

In order to solve the data overwriting problem, it was decided to load the output data, the output time (equal to input time + pre-delay + latency) and the arc index (identifying the arc) onto a queue whenever a primitive completes its operations on a signal. The arc index-output data-output time vector can now be referred to as an event. Events are stored in the queue in order of increasing event time with the earliest occurring event at the head of the queue. When it is time for a primitive to process a data sample, it locates the data in the queue by locating the desired input arc index and event time and then loading the corresponding data into the proper location in the arc array. Thus, each primitive is assured of valid input data and the overwriting of data of the arc array is immaterial. Indeed, the arc array now becomes more of a communications path and less of a storage facility, with the latter role now being performed by the queue. Overwriting in the queue is avoided by defining a new record for every new event, discarding it only after loading it back into the arc array on the request of a primitive routine.

The queue is implemented as a global recursive-style linked list. Event records are dynamically allocated during run-time and linked via access types (similar to pointers in C or PASCAL) to other event records already in existence. Events are loaded onto the queue by the SIMPKG procedure "ENQUEUE," returned to the arc
array by the procedure "RETRIEVE," and deleted from the queue by the procedure "DELETE."

**Advancing Simulation Time**

The storage of data on the queue serves a much more useful purpose than just ensuring the validity of primitive input data. Since the events in the queue are ordered by increasing time, the event at the head of the queue will be the next event to occur in the simulation. If the time of occurrence of the next event is greater than the current simulation time (i.e., the next event occurs at some time in the future), it would be pointless to invoke any new primitives, as no new data would be available for processing. The same would hold true for all simulation times earlier than the next event time. Obviously, the simulation clock should skip ahead to the time of the next event, ignoring intermediate clock cycles. This results in an event-driven simulation and a significant improvement in simulator throughput. It is similar, in fact, to Denyer and Renshaw's event-driven behavioral simulator (Denyer and Renshaw (1985)).

**Control Considerations**

Control signals are generated in a chip description by delaying the outputs of the chip's "control generator" primitive. The delays are accomplished by the "CBITDELAY" primitive, which delays the original control signal by a user-specified latency.
There are three possible control signals from the control generator, coinciding with the first bit of a word, the first word of a group of words, and the first group of a block of groups of words, respectively, and labelled C1, C2 and C3. Simulation of these control signals is accomplished by loading the first element of three global arrays named C1, C2 and C3 with the simulation time of the most recent occurrence of C1, C2 and C3. The "CBITDELAY" primitives are now used to fill out the rest of the arrays with the corresponding delayed signal occurrence time.

When invoked by the occurrence of an event on a data arc, a primitive will compare the present (simulation) time to the time of its control signal. If the times are not identical, the simulator will issue a warning message and commence a diagnostics routine to assist the designer in locating the missing control signal. A weakness of this arrangement is that extra control signals will not be detected by the simulator but would cause erroneous operation in the actual hardware system. This feature should be added in future versions of this simulation.
CHAPTER VI
CONCLUSIONS

The purpose of this research was to investigate the possibility of using ADA as a means of describing digital hardware and performing subsequent simulation of the description. That goal was accomplished with the behavioral description and simulation of a complex-to-magnitude digital signal processing algorithm implementation.

In theory, almost any task that is within the capabilities of one programming language can be accomplished by another language as well, given sufficient time and resources. How well the second language accomplishes the task compared to the first is to a large extent subjective. Programming languages and styles tend to vary in popularity as a matter of personal preference as much as with any true measure of their capabilities (Nash and Saunders 1986). Nevertheless, some additional conclusions were reached as a result of this project:

1. General purpose languages are useful for hardware description. As previously noted, Dietmeyer and Duley questioned the wisdom of allowing hardware description languages to be strongly influenced by general purpose programming languages (Dietmeyer and Duley 1975). It should be noted that this was
written prior to the current emphasis on top-down design of
digital systems. That same ten years has seen programming
languages become ever more powerful and versatile, with an
emphasis on strong type checking, readability, maintainability,
structuring and standardization that was the exception rather than
the rule in 1975. If present hardware description languages are
only now evolving to the level of the 1950s programming languages,
it seems reasonable to assume that a powerful language like ADA
would be far enough advanced to perform adequately in the same
role as lesser developed description languages.

2. ADA is not difficult to learn and well worth the effort. It appears that anyone familiar with C or PASCAL can adapt to ADA
with little additional effort. Despite concerns of those who
argued against full ADA inclusion in VHDL on the grounds that
training designers in ADA and VHDL would be too difficult (Nash
and Saunders 1986), it is believed that the added power of a full
ADA implementation in VHDL would justify any additional training
effort.

3. Although not originally intended for the role, ADA is a
good choice for hardware description due to its ready availability
and support. By 1990, software costs are expected to amount to
90% of all computing costs (Fairley 1986). Organizations without
a presently available design automation system might profit more
from ADA development with its base of existing software support than from starting from scratch with a special purpose language.

4. Further research is justified. Future work should concentrate on silicon compilation from ADA behavioral descriptions, automatic test vector generation for the simulation, improvement in the simulation tools and additional operators for use by large systems.
APPENDICES
APPENDIX A

SIMULATION PACKAGE
with text_io, sequential_io;
use text_io, sequential_io;

Package SimPkg is

    package int_io is new integer_io(integer);
    use int_io;
    package flt_io is new float_io(float);
    use flt_io;

    -- simulation type

    Type simulation_type is
    record
        time: natural;
        last_time: integer;
    end record;

    sim: simulation_type;

    -- data variables

    Type signal is
    record
        data: float;
        time: natural;
    end record;

    package seq_io is new sequential_io(signal);
    use seq_io;

    subtype arc_no is integer
    range -1 .. integer'last;

    NC: constant := -1;

    type signal_list is array (arc_no range <>)
    of signal;

    arc: signal_list (-1..100);
    indata: signal_list (1..10);
    outdata: signal_list (1..10);

    -- control variables
type carc is array(natural range 0..100)
of natural;
c1,c2,c3:carc;

-- queue variables

type queue_type;
type queue_ptr is access queue_type;
type queue_type is
  record
    arc_number:arc_no;
data:float;
time:natural;
link:queue_ptr;
end record;

head:queue_ptr:=null;

-- misc global variables

subtype bit is integer range 0..1;
event:arc_no;
count:positive;
input_file,output_file:seq_io.file_type;
data_ready:natural;
trace_enable,name_enable:boolean:=false;

-- simulation procedures

procedure ADVANCE_TIME;
procedure CLOSE_FILE;
procedure DATA_IN(number,ctrl:natural);
procedure DATA_OUT(number,ctrl:natural);
procedure DELETE(arc_number:arc_no);
procedure ENQUEUE(arc_number:arc_no);
procedure DIAGNOSTICS;
procedure KEY_IN(number,ctrl:natural);
procedure OPEN_FILE;
procedure PRINT_OUT(number,ctrl:natural);
procedure RETRIEVE(arc_number:arc_no;
    del:bit:=0);
procedure TRACE;

-- primitive procedures

procedure ABSOLUTE(input,output:arc_no;
    ctrl:natural;
    swl:positive:=16;
    del:bit:=0);

procedure ADD(addend1,addend2,sum:arc_no;
    ctrl:natural;latency:positive:=1;
    del1,del2:bit:=0);

procedure BITDELAY(input,output:arc_no;
    latency:positive:=1);

procedure CBITDELAY(cin:natural;
    cout:in out natural;
    latency:positive:=1);

procedure CONTROL_GENERATOR(c1_cnt,c2_cnt,
    c3_cnt:positive:=1);

procedure DSHIFT(input,output:arc_no;
    ctrl:natural;
    power2:positive:=1;
    del:bit:=0);

procedure ORDER(in1,in2,max,min:arc_no;
    ctrl:natural;
    swl:positive:=16;del1:bit:=0);

procedure PADIN(ext,int:arc_no;ctrl:natural);
procedure PADOUT(int, ext: arc_no; ctrl: natural);

procedure SUBTRACT(minuend, subtrahend, difference: arc_no;
                   ctrl: natural;
                   latency: positive := 1;
                   del1, del2: bit := 0);

end SIMPKG;

package body SIMPKG is

procedure ADVANCE_TIME is

   temp_time: natural;

begin

   if name_enable = true then
      put_line("ADVANCE_TIME");
   end if;

   while head.all.arc_number = -1 loop
      head := head.all.link;
   end loop;

   temp_time := head.all.time;

   if temp_time > cl(0) + count then
      sim.time := cl(0) + count;
   else sim.time := temp_time;
      event := head.all.arc_number;
   end if;

end ADVANCE_TIME;

procedure CLOSE_FILE is
begin
close(input_file);
close(output_file);
end CLOSE_FILE;

procedure DATA_IN(number,ctrl:natural) is
begin
  if name_enable = true then
    put_line("DATA_IN");
  end if;
  if (sim.time=ctrl) and
    (sim.last_time /= sim.time) then
    for i in 1..number loop
      read(input_file,indata(i));
    end loop;
    sim.last_time:=sim.time;
  end if;
end DATA_IN;

procedure DATA_OUT(number,ctrl:natural) is
begin
  if name_enable = true then
    put_line("DATA_OUT");
  end if;
  if sim.time=ctrl then
    for i in 1..number loop
      write(output_file,outdata(i));
    end loop;
  end if;
end DATA_OUT;
procedure DELETE(arc_number: arc_no) is

    search_pos: queue_ptr := head;
    next_node: queue_ptr renames
        search_pos.all.link;
    event_time: natural renames
        arc(arc_number).time;

begin

    if name_enable = true then
        put_line("DELETE");
    end if;

    if head=null then
        return;
    elsif (head.all.time = event_time) and
        (head.all.arc_number = arc_number) then

        head := head.all.link;
        return;
    else

        while search_pos.all.link /= null loop

            if (search_pos.all.link.all.time
                = event_time) and
                (search_pos.all.link.all.arc_number
                = arc_number) then

                search_pos.all.link :=
                    search_pos.all.link.all.link;
                return;

            else search_pos := search_pos.all.link;

            end if;

        end loop;

    end if;

end DELETE;
procedure ENQUEUE(arc_number:arc_no) is

  prior_pos,search_pos:queue_ptr;
  event_time:natural renames
    arc(arc_number).time;
  event_data:float renames
    arc(arc_number).data;

begin

  if name_enable = true then
    put_line("ENQUEUE");
  end if;

  if head=null then
    head:=new
      queue_type'(arc_number,
      event_data,event_time,null);
  elsif head.all.time > event_time then
    head:=new queue_type'(arc_number,
      event_data,event_time,head);
  else

    prior_pos:=head;
    search_pos:=head;
    while (search_pos.all.time < event_time)
      and (search_pos.all.link /=null) loop
      prior_pos:=search_pos;
      search_pos:=search_pos.all.link;
    end loop;

    if search_pos.all.link = null then
      search_pos.all.link:=new
        queue_type'(arc_number,event_data,
        event_time,null);
    else

      prior_pos.all.link:=new
        queue_type'(arc_number,event_data,

event_time,prior_pos.all.link);

end if;

end if;
trace;
end ENQUEUE;

procedure DIAGNOSTICS is
    halt:exception;
begin
    trace_enable:=true;
    put_line("DIAGNOSTICS");
    trace;
    put_line("Execution terminated.");
    raise halt;
end DIAGNOSTICS;

procedure KEY_IN(number,ctrl:natural) is
begin
    if name_enable = true then
        put_line("KEY_IN");
    end if;
    if (sim.time = ctrl) and (sim.time /= sim.last_time) then
        data_ready:=0;
        for i in 1..number loop
            put("Time = "); put(sim.time);
            new_line;
            put("Enter next data point for input ");
            put(i);
            put_line(";");
            get(indata(i).data);
            indata(i).time:=sim.time;
            new_line;
            data_ready:=data_ready + 1;
        end loop;
    end if;
end KEY_IN;
end loop;
    sim_last_time:=sim_time;

end if;
end KEY_IN;

procedure OPEN_FILE is
begin
    open(input_file,in_file,"simdata ada");
    open(output_file,out_file,"results ada");
end OPEN_FILE;

procedure PRINT_OUT(number,ctrl:natural) is
begin
    if name_enable = true then
        put_line("PRINT_OUT");
    end if;
    if sim_time=ctrl then
        for i in 1..number loop
            new_line;
            put("Output data is: ");
            put(outdata(i).data);
            new_line;
            put("Output time is: ");
            put(outdata(i).time);
            new_line;
        end loop;
    end if;
end PRINT_OUT;

procedure RETRIEVE(arc_number:arc_no;
    del:bit:=0) is
    search_pos:queue_ptr:=head;
next_node : queue_ptr renames search_pos.all.link;

begin

if name_enable = true then
    put_line("RETRIEVE");
end if;

if head=null then return;
elsif (head.all.time + del = sim.time) and
    (head.all.arc_number = arc_number) then

    event_time:= head.all.time;
    event_data:= head.all.data;
    trace;
    return;

else

    search_pos:= head;
    while search_pos.all.link /= null loop

        if (search_pos.all.link.all.time
            + del = sim.time) and
            (search_pos.all.link.all.arc_number
            = arc_number) then

            event_time:= search_pos.all.link.all.time;
            event_data:= search_pos.all.link.all.data;
            trace;
            return;

        else search_pos:= search_pos.all.link;

        end if;

    end loop;

    trace;
end if;
end RETRIEVE;

procedure TRACE is
    search_pos:queue_ptr:=head;
begin
    if trace_enable = false then return;
    end if;
    put_line("Trace for ");
    put("Simulation time = ");
    put(sim.time);
    new_line;
    put_line("Queue contents: ");
    new_line;
    while search_pos /= null loop
        put(search_pos.all.arc_number);
        put(search_pos.all.data);
        put(search_pos.all.time);
        new_line;
        search_pos:=search_pos.all.link;
    end loop;
    new_line;
    put_line("Arc array contents: ");
    new_line;
    for i in 1..20 loop
        put(i);
        put(arc(i).data);
        put(arc(i).time);
        new_line;
    end loop;
    new_line;
    put_line("Control array contents: ");
    new_line;
for i in 1..20 loop
    put(i);
    put(c1(i));
    new_line;
end loop;
end TRACE;

-- primitive procedures

procedure ABSOLUTE(input,output:arc_no;
    ctrl:natural;
    swl:positive:=16;
    del:bit:=0) is

    latency:positive;
    data_timing,ctrl_timing:exception;
    temp:natural;

begin
    if name_enable = true then
        put_line("ABSOLUTE");
    end if;
    temp:=sim.time;
    sim.time:=sim.time+del;
    if ctrl=sim.time then
        retrieve(input,del);
        latency:=sw1+3;
        if arc(input).time+del=sim.time then
            delete(input);
            arc(output).data:=
                abs(arc(input).data);
            arc(output).time:=
                arc(input).time+latency+del;
            enqueue(output);
        else raise data_timing;
    end if;
end if;
else raise ctrl_timing;
end if;
sim.time:=temp;

exception

when data_timing=>
    put("Data not available for 'absolute' at time ");
    put(sim.time);
    new_line;
diagnostics;

when ctrl_timing=>
    put("no control signal for 'absolute' at time ");
    put(sim.time);
    new_line;
diagnostics;

end ABSOLUTE;

procedure ADD(addend1,addend2,sum:arc_no;
    ctrl:natural;
    latency:positive:=1;
    del1,del2:bit:=0) is

    data_timing,ctrl_timing:exception;
    temp:natural;

begin

    if name_enable = true then put_line("ADD");
    end if;

    temp:=sim.time;
    if del1=1 or del2=1 then
        sim.time:=sim.time+1;
    end if;
    if ctrl=sim.time then
        retrieve(addend1,del1);
        retrieve(addend2,del2);
    end if;

end procedure ADD;
if (sim.time=arc(addend1).time+del1) and (sim.time=arc(addend2).time+del2) then

  delete(addend1);
  delete(addend2);
  arc(sum).data:=
    arc(addend1).data+arc(addend2).data;
  arc(sum).time:=
    arc(addend1).time+latency+del1;
  enqueue(sum);

else raise data_timing;
end if;

else raise ctrl_timing;
end if;
sim.time:=temp;

exception

  when data_timing=>
    put("Data not available for 'add' at time ");
    put(sim.time);
    new_line;
    diagnostics;

  when ctrl_timing=>
    put("no control signal for 'add' at time ");
    put(sim.time);
    new_line;
    diagnostics;

end ADD;

procedure BITDELAY(input,output:arc_no;
  latency:positive:=1) is

  data_timing:exception;

begin

  if name_enable = true then
    put_line("BITDELAY");
end if;

retrieve(input);
if sim.time=arc(input).time then
  delete(input);
  arc(output).data:=arc(input).data;
  arc(output).time:=
    arc(input).time+latency;
  enqueue(output);
else raise data_timing;
end if;

exception

when data_timing=>
  put("Data not available for 'bitdelay' at time ");
  put(sim.time);
  new_line;
  diagnostics;

end BITDELAY;

procedure CBIDTDELAY(cin:natural;
  cout:in out natural;
  latency:positive:=1) is
begin
  if name_enable = true then
    put_line("CBIDTDELAY");
  end if;

  if (sim.time>cout) or (sim.time=0) then
    cout:=cin+latency;
  end if;
end CBIDTDELAY;

procedure CONTROL_GENERATOR(c1_cnt,c2_cnt,
begin

if name_enable = true then
    put_line("CONTROL GENERATOR");
end if;

count:=c1_cnt;
if sim.time mod c1_cnt = 0 then
    c1(0):=sim.time;
end if;
if sim.time mod (c1_cnt*c2_cnt)=0 then
    c2(0):=sim.time;
end if;
if sim.time mod (c1_cnt*c2_cnt*c3_cnt)=0 then
    c3(0):=sim.time;
end if;
end CONTROL GENERATOR;

procedure DSHIFT(input,output:arc_no;
ctrl:natural;
power2:positive:=1;
de1:bit:=0) is

latency,temp:natural;
data_timing,ctrl_timing:exception;

begin

if name_enable = true then
    put_line("DSHIFT");
end if;

temp:=sim.time;
sim.time:=sim.time+del;
if ctrl=sim.time then
  retrieve(input,del);
  latency:=power2 + 3;
  if arc(input).time+del=sim.time then
    delete(input);
    arc(output).data:=
      arc(input).data/(2.0**power2);
    arc(output).time:=
      arc(input).time+latency+del;
    enqueue(output);
  else raise data_timing;
  end if;
  else raise ctrl_timing;
  end if;
sim.time:=temp;

exception
  when data_timing=>
    put("Data not available for 'dshift'
         at time ");
    put(sim.time);
    new_line;
    diagnostics;
  when ctrl_timing=>
    put("no control signal for 'dshift'
         at time ");
    put(sim.time);
    new_line;
    diagnostics;
end DSHIFT;

procedure MULTIPLY(data,coeff,delayeddata, 
                  product:arc_no; 
                  ctrl:natural; 
                  coeffbits:natural:=16; 
                  round:boolean:=false; 
                  del1,del2:bit:=0) is
begin
  if name_enable = true then
    put_line("MULTIPLY");
  end if;

  temp:=sim.time;
  if del1=1 or del2=1 then
    sim.time:=sim.time+1;
  end if;

  if ctrl=.sim.time then
    retrieve(data,del1);
    retrieve(coeff,del2);
    if (sim.time=arc(data).time+del1) and (sim.time=arc(coeff).time+del2) then
      delete(data);
      delete(coeff);
      latency:=2*arc(coeff).data+2);
      arc(product).data:=
        arc(data).data*arc(coeff).data;
      arc(product).time:=
        arc(data).time+latency+del1;
      enqueue(product);
      arc(delayeddata).data:=
        arc(data).data;
      arc(delayeddata).time:=
        arc(data).time+del1+latency;
      enqueue(delayeddata);
      else raise data_timing;
    end if;

    else raise ctrl_timing;
  end if;
  sim.time:=temp;
exception
when data_timing=>
    put("Data not available for 'multiply' at time ");
    put(sim.time);
    new_line;
    diagnostics;

when ctrl_timing=>
    put("no control signal for 'multiply' at time ");
    put(sim.time);
    new_line;
    diagnostics;

end MULTIPLY;

procedure ORDER(in1,in2,max,min:arc_no;
    ctrl:natural;
    swl:positive:=16;
    del1: bit:=0) is

    latency,temp:natural;
    data_timing,ctrl_timing:exception;

begin

    if name_enable = true then
        put_line("ORDER");
    end if;

    temp:=sim.time;
    sim.time:=sim.time+del1;
    if ctrl=sim.time then

        retrieve(in1,del1);
        retrieve(in2);
        latency:=swl+3;
        if (sim.time=arc(in1).time+del1) and
            (sim.time=arc(in2).time) then

            delete(in1);
            delete(in2);
            if arc(in1).data>arc(in2).data then

                arc(max).data:=arc(in1).data;

            else

                arc(max).data:=arc(in2).data;

            end if;

        else

            delete(in1);
            delete(in2);

        end if;

    end if;

end ORDER;
arc(min).data:=arc(in2).data;
else arc(max).data:=arc(in2).data;
arc(min).data:=arc(in1).data;
end if;
arc(max).time:=arc(in2).time+latency;
arc(min).time:=arc(in2).time+latency;
enqueue(max);
enqueue(min);
else raise data_timing;
end if;
else raise ctrl_timing;
end if;
sim.time:=temp;

exception

when data_timing=>
    put("Data not available for 'order' at time ");
    put(sim.time);
    new_line;
    diagnostics;

when ctrl_timing=>
    put("No control signal for 'order' at time ");
    put(sim.time);
    new_line;
    diagnostics;
end ORDER;

procedure PADIN(ext,int:arc_no;
ctrl:natural) is
    data_timing,halt:exception;
begin
    if name_enable = true then
        put_line("PADIN");
end if;

if (sim.time=ctrl) and 
(data_ready > 0) then

if indata(ext).time=sim.time then

    arc(int).data:=indata(ext).data;
    arc(int).time:=indata(ext).time;
    event:=int;
    data_ready:=data_ready-1;
    enqueue(int);

else raise data_timing;
end if;

end if;

exception

when data_timing=>
    put("Input data time faulty at time ");
    put(sim.time);
    new_line;
    put("Terminating execution.");
    raise halt;

end PADIN;

procedure PADOUT(int,ext:arc_no;
    ctrl:natural) is
begin

    if name_enable = true then
        put_line("PADOUT");
    end if;

    if sim.time=ctrl then

        retrieve(int);
        delete(int);
        outdata(ext).data:=arc(int).data;
        outdata(ext).time:=arc(int).time;

    end if;

end PADOUT;
end if;
end PADOUT;

procedure SUBTRACT(minuend, subtrahend, difference: arc_no;
ctrl1: natural;
latency: positive := 1;
del1, del2: bit := 0) is

data_timing, ctrl1_timing: exception;
temp: natural

begin

if name_enable = true then
  put_line("SUBTRACT");
end if;

temp := sim.time;
if del1 = 1 or del2 = 1 then
  sim.time := sim.time + 1;
end if;

if ctrl1 = sim.time then
  retrieve(minuend, del1);
  retrieve(minuend, del2);
  if (sim.time = arc(minuend).time + del1) and
      (sim.time = arc(subtrahend).time + del2)
  then
    delete(minuend);
    delete(subtrahend);
    arc(difference).data :=
      arc(minuend).data -
      arc(subtrahend).data;
    arc(difference).time :=
      arc(minuend).time + latency + del1;
    enqueue(difference);
  else raise data_timing;
end if;
else raise ctrl_timing;
end if;
sim.time:=temp;

exception

when data_timing=>
  put("Data not available for 'subtract'
       at time ");
  put(sim.time);
  new_line;
  diagnostics;

when ctrl_timing=>
  put("no control signal for 'subtract'
       at time ");
  put(sim.time);
  new_line;
  diagnostics;

end SUBTRACT;

end SIMPKG;

-- end of package SIMPKG.
APPENDIX B

SIMULATOR
with simpkg, text_io;
use simpkg, text_io;

procedure SIMULATE is

   package int_io is new integer_io(integer);
   use int_io;
   package flt_io is new float_io(float);
   use flt_io;

   maxclk: natural := 1;
   response: character := 'n';
   one_trace_enable: boolean := false;

   procedure SYSTEM is separate;

begin

   put("Enter number of clock times to simulate: ");
   get(maxclk);
   new_line;

   put("Print routine names (y/n)? ");
   get(response);
   if response = 'y' or response = 'Y' then
      name_enable:=true;
   end if;
   new_line;

   put("Print traces (y/n)? ");
   get(response);
   if response = 'Y' or response = 'y' then
      trace_enable:=true;
   end if;
   new_line;

   if trace_enable = false then
      put("Print trace for time advance (y/n)? ");
      get(response);
      if response = 'Y' or response = 'y' then
         one_trace_enable:=true;
      end if;
   end if;
new_line;
end if;
sim.time:=0;
sim.last_time:=-1;

while sim.time <= maxclk loop

SYSTEM;
ADVANCE_TIME;
if name_enable = true then
    put("New sim.time = "); put(sim.time);
    new_line;
end if;
if (one_trace_enable = true) then
trace_enable:=true;
TRACE;
    trace_enable:=false;
else
    TRACE;
end if;
end loop;

put_line("Simulation complete.");

end SIMULATE;--
APPENDIX C

SAMPLE SYSTEM
-- a system description of the Complex to Magnitude algorithm.

separate(SIMULATE)

procedure SYSTEM is

procedure CONTROL_NETWORK is

begin

    CBITDELAY(c1(0), c1(3), 3);
    CBITDELAY(c1(3), c1(6), 3);
    CBITDELAY(c1(6), c1(12), 6);
    CBITDELAY(c1(12), c1(13), 1);
    CBITDELAY(c1(13), c1(14), 1);
    CBITDELAY(c1(14), c1(17), 3);

end CONTROL_NETWORK;

procedure SEVEN_EIGHTHS(input_arc: arc_no;
ctrl1, ctrl2: natural) is

    event_1, x: arc_no;

begin

    x := input_arc;

    event_1 := (event - input_arc) + 1;

    case event_1 is

    when 1 =>
        DSHIFT(input=>x, output=>x+2, power2=>3,
ctrl=>ctrl1);
        BITDELAY(input=>x, output=>x+3,
lacnety=>6);

    when 2 =>
        BITDELAY(input=>x+1, output=>x+5,
latency=>3);

    when 3|4 =>
SUBTRACT(minuend=>x+3, subtrahend=>x+2, difference=>x+6, latency=>1, ctrl=>ctrl2);
BITDELAY(input=>x+3, output=>x+4, latency=>1);

when others =>
  put_line("Seven_eighths failed.");
DIAGNOSTICS;
end case;
end SEVEN_EIGHTHS;

procedure COMPLEX_TO_MAGNITUDE is
begin
  case event is
    when 1=>
      ABSOLUTE(input=>1, output=>3, swl=>16, ctrl=>c1(0));
    when 2=>
      ABSOLUTE(input=>2, output=>4, swl=>16, ctrl=>c1(0));
    when 3..4=>
      ORDER(in1=>3, in2=>4, max=>6, min=>5, swl=>16, ctrl=>c1(3));
    when 5=>
      DSHIFT(input=>5, output=>7, power2=>1, ctrl=>c1(6));
    when 6..9=>
      SEVEN_EIGHTHS(6, c1(6), c1(12));
    when 10..13=>
      ORDER(in1=>10, in2=>13, max=>14, min=>NC, swl=>16, ctrl=>c1(14), del1=>1);
    when 11..12=>
      ADD(addend1=>11, addend2=>12, sum=>13,
latency=>1,ctrl=>c1(13));

when others=>
  put("Event = ");
  put(event);
  new_line;
  diagnostics;

end case;

end COMPLEX_TO_MAGNITUDE;

begin
  CONTROL: declare
  begin
    CONTROL_GENERATOR(16);
    CONTROL_NETWORK;
  end CONTROL;

  DATA: declare
  begin
    KEY_IN(2,c1(0));
    PADIN(1,1,c1(0));
    PADIN(2,2,c1(0));

    case EVENT is
      when 1..13 =>
        COMPLEX_TO_MAGNITUDE;
      when 14 =>
        PADOUT(14,1,c1(17));
        PRINT_OUT(1,c1(17));
      when others =>
        DIAGNOSTICS;
    end case;

  end DATA;

end SYSTEM;
-- end of system description.
APPENDIX D
PRIMITIVE OPERATORS

DSP Simulation Primitives

<table>
<thead>
<tr>
<th>Arithmetic</th>
<th>Storage</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>absolute</td>
<td>bitdelay</td>
<td>cbidelay</td>
</tr>
<tr>
<td>add</td>
<td>worddelay</td>
<td>controlgenerator</td>
</tr>
<tr>
<td>constgen</td>
<td></td>
<td>cworddelay</td>
</tr>
<tr>
<td>dpmultiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dshift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mshift</td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiplex</td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>order</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subtract</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Format</th>
<th>Pads</th>
</tr>
</thead>
<tbody>
<tr>
<td>fformat1to1</td>
<td>padin</td>
</tr>
<tr>
<td>fformat2to1</td>
<td>padout</td>
</tr>
<tr>
<td>fformat3to1</td>
<td></td>
</tr>
<tr>
<td>format1to2</td>
<td></td>
</tr>
<tr>
<td>flimit</td>
<td></td>
</tr>
</tbody>
</table>

68
APPENDIX E

SUMMARY OF DESCRIPTION AND SIMULATION PROCEDURES

I. DESCRIPTION

A. Draw a signal flow graph of the desired architecture.
   1. Number arcs using consecutive integers starting with 1.
      a. If using operators as well as primitives, reserve a block of consecutive arc ID numbers consistent with the number of internal arcs in the operator. For example, seven-eighths has 2 internal arcs. These arc ID numbers are reserved for seven­eighths and are not used by this higher level complex-to-magnitude operator.
      b. Output arcs that are not utilized in a particular application (like the "minimum" output of the last "order" primitive in complex-to-magnitude) should be assigned an arc ID of "NC" (no connection). Data on "NC" arcs will be removed automatically by the simulator.
   2. Connect a control delay network to the nodes (primitives and operators) of the signal flow graph above. Assign delays consistent with the primitive's latencies.

B. Encode the description of the chip.
   1. The description must start with the statement:
      SEPARATE (SIMULATE)
      This will allow the linker to link the compiled description object code with the previously compiled simulator object code. (The simulator, Appendix B, must be compiled before the chip description, but once compiled, does not have to be recompiled for each new chip description.)
2. Include a control-network description as a list of "cbitdelay" statements. Consult the SIMPKG declarative part (Appendix A) for the proper cbitdelay syntax.

3. Include descriptions of the operators used in your chip description. These descriptions can be included in the declarative portion of the chip procedure (like complex-to-magnitude and seven-eighths in Appendix C) or separately compiled with a reference in the declarative part of the chip procedure (i.e., procedure complex-to-magnitude is separate). Detailed instructions on operator description are contained in Appendix F.

4. The executable portion of the description is made up of two block statements, control and data.
   a. The control block contains calls to the control-generator primitive, which generates C1, C2 and C3, and the control-network, which produces delayed signals C1-X, C2-X and C3-X.
   b. The data portion consists of an input section (key-in and padin calls) and an event-driven operator section. For more information on event-driven operators, see Appendix F.
APPENDIX F
THE EVENT-DRIVEN OPERATOR

The key to understanding the operator description lies in the use of a "case" statement. The global variable "event" is loaded with the arc ID number of the next event to be simulated by the subroutine ADVANCE-TIME. "Event" will now be used via a case statement to select which primitives will be executed.

Compare the listing of procedure complex-to-magnitude in Appendix C with the signal flow graph shown in Figure 2. Assume that this next event occurs on arc 2 which is the input arc to an "absolute" primitive. Examination of the complex-to-magnitude listing shows:

Case Event is

...  

When 2 => absolute (input => 2, output => 4, swl => 16, ctrl => Cl(0));

Thus, when an event occurs on arc 2, the absolute primitive will be called.

The use of the case statement above unfortunately ties the description to certain arc ID numbers. In order to create a more
general description, a local variable can be loaded with a modified arc number.

Examine the listing of procedure seven-eighths in Appendix C. Note that the input arc number (the number assigned to a higher level description; in this case, the complex-to-magnitude description) is sent to the seven-eighths procedure as a parameter. This arc is treated within seven-eighths as arc 1. Instead of this global variable "event," a local variable event-1 that has been adjusted by the input arc parameter is used to drive the seven-eighths case statement. Thus, an event on arc 6 is seen by seven-eighths as an event on its arc 1.
REFERENCES


