Methodology for structured VHDL model development

by

Ram Gummadi

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

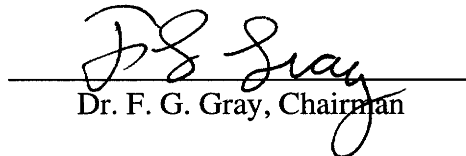
in partial fulfillment of the requirements for the degree of

Master of Science

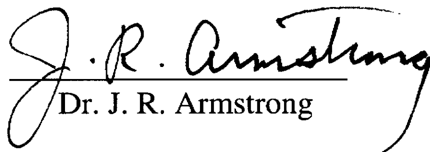
in

Electrical Engineering

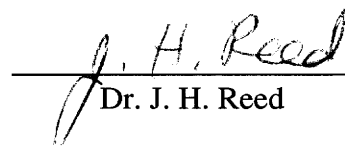
APPROVED:



Dr. F. G. Gray, Chairman



Dr. J. R. Armstrong



Dr. J. H. Reed

April 1995

Blacksburg, Virginia

LD
5655
V855
1995
G8606
c.2

Methodology for structured VHDL model development

by

Ram Gummadi

Dr. F. Gail Gray, Chairman

Electrical Engineering

(ABSTRACT)

The Rapid Prototyping of Application Specific Signal Processors (RASSP) program seeks an improvement in the time required to take a design from concept to fielded prototype or to upgrade an existing design, with similar improvements in design quality and life cycle cost. The term Rapid System Prototyping signifies the need to develop systems in significantly less time or with significantly less effort, and thus provides a solution to the main problem facing the design community. Entire systems are synthesized from models in hardware description languages (HDLs).

The goal of this thesis is to provide a methodology for rapidly creating a database, that can be reused thus decreasing design cost and time for both current and future projects. To demonstrate the methodology, this thesis describes the development of VHDL primitives supporting digital signal processing (DSP) and image processing operations for two of the RASSP specific applications: 1) Synthetic aperture radar image processor (SAR) and 2) Automatic target recognition (ATR) image processing algorithm. Different techniques are investigated to populate these VHDL libraries using commercial tools. The thesis proposes techniques for solving some problems related to the use of commercial tools to generate VHDL code. It includes a full implementation of the SAR processor algorithm developed from DSP primitives.

To my parents

Acknowledgments

I wish to express my thanks to my advisor Dr. F. Gail Gray whose guidance and encouragement was invaluable. It has been an honor and a great pleasure to study and work under him. I would also like to express my gratitude to Dr. J. R. Armstrong for his support and for serving as a member on my committee.

Also, I would like to thank Dr. Jeffrey H. Reed for being on my graduate committee. I would also like to thank Dr. Geoffrey A. Frank and Dr. Bud Clark of Research Triangle Institute, Raleigh for their support.

I appreciate the backing and assistance of all my friends who have been with me. Special thanks to Suri and Ravi. I would like to dedicate this work to my parents who have always encouraged me at all the times.

Table of Contents

Chapter 1. Introduction **1**

1.1 Motivation 1

1.2 Task Description 3

1.3 Contributions 5

1.4 Design cycle for a DSP system 7

1.5 Thesis Organization 11

Chapter 2. Background **13**

2.1 Overview on VHDL 13

2.2 Basic VHDL models 13

2.3 Library Features 17

Chapter 3. Library Primitives **20**

3.1 Overview 20

3.2 Characteristics of the primitives 22

3.3 Development of library primitives 23

3.3.1 Fast fourier transform 24

3.3.2 Convolution 27

3.3.3 Finite impulse response filter 29

3.3.4 Two dimensional convolution 31

3.3.5 Two dimensional fast fourier transform 33

Chapter 4. Synthetic Aperture Radar **35**

4.1 Introduction 35

4.2 Synthetic aperture radar 35

4.2.1 Input data 36

4.2.2 Image processor for SAR 38

4.2.3 Pulse repetition interval detection 38

4.2.4 Video to baseband I/Q conversion 39

4.2.5 Range compression 42

4.2.6 Azimuth compression 42

4.3 Primitives for the SAR image processor 45

4.4 SAR results 46

Chapter 5. Automatic Target Recognition algorithm **49**

5.1 Introduction 49

5.2 Automatic target recognition algorithm 49

5.2.1 Morphological operations 50

5.2.2 Binary template matching 51

5.2.3 Mean square error algorithm 52

5.3 Primitives to support the ATR algorithm 53

Chapter 6. Methodology 55

6.1 Introduction 55

6.2 Overview of Matlab 56

6.3 Integrating Matlab code into VHDL 57

6.3.1 Purpose 57

6.3.2 Modes of operation 58

6.3.3 Advantages/Disadvantages, comparator 63

6.3.4 Results 64

6.4 Introduction to tools 65

6.4.1 Ptolemy 65

6.4.2 Purpose/current status of Ptolemy 67

6.4.3 Generation of VHDL code supporting floating point arithmetic 68

6.5 Techniques to write VHDL code 74

6.5.1 Overloading operators 74

6.5.2 Overloading subprograms 76

6.5.3 Data types 78

Chapter 7. Conclusion and Future Work 81

Bibliography 82

Appendix A: VHDL code for the primitives 84

Appendix B: Ptolemy VHDL model for complex multiplication **92**

Appendix C: SAR VHDL model **94**

Appendix D: VHDL code for the ATR primitives **106**

Vita **110**

Chapter 1. Introduction

1.1 Motivation

The critical importance of the time to market is especially recognized in high technology computer products. Although a three to five year design cycle was commonplace a decade ago, this period now often exceeds the useful lifetime of a microelectronic product. At the same time, the complexity of microelectronic products has reached new heights. Recent microprocessors incorporate two million transistors on a die and some operate at speeds of 300 Megahertz. New products, from global positioning system-based computers for navigation to multimedia educational environments, demand that microelectronic system designers find new approaches to the concept-to-deliverable

system design cycle. In turn, designers require ever-improving capabilities from their computer-aided design (CAD) tools, which frequently require new developments in theory. Advances in theory facilitate such needs as provably correct design methodology and formal languages for specifications.

The term *rapid system prototyping* signifies the need to develop systems in significantly less time or with significantly less effort, and thus provides context for the driving problem in the design community.[1] Development of systems in significantly less time requires a high level language with hardware constructs (HDL), which can be easily mapped onto readily available commercial gate arrays. The significant advantages in using an HDL are less time to market at a lower cost. Field-programmable gate arrays are now routinely used for functional verification of microelectronic systems, and entire systems are synthesized from models in hardware description languages (HDLs) such as VHDL.

HDLs allow designers to create models of chips at various levels in the design hierarchy ranging from the system level to the switch level. HDLs can model control flow, data flow and timing relationships between the various blocks while reducing the quantity and detail of information that a designer needs to manage.

The *VHSIC Hardware Description Language* (VHDL) is becoming very popular in the electronic industry as a modeling tool. VHDL is an IEEE standardized language (IEEE 1076 Standard). It has a large set of constructs for describing circuit behavior and supports both top-down and bottom-up design methodologies through varying levels of abstraction. Once a behavioral model of the circuit is developed in VHDL, test patterns

can be generated for the simulation of the model. The functionality of the circuit can be verified by observing the simulation output.

Design verification by the simulation of circuits using powerful computer tools is a very widely used technique. Here, the circuit is simulated with a very large amount of test data given by the modeler. Generation of this data set is a very expensive and time-consuming task. Also, the test data generated manually satisfies no formal definition of completeness. To relieve the user of such problems, high level approaches to test generation has become very popular[7].

A major focus of the RASSP program here at Virginia Tech is the rapid development of VHDL models for signal processing algorithms and different architectures. However, development of libraries consisting of digital signal processing, image processing, radar and architectural models is a time consuming task. There is a critical need to develop high level approaches to support this task.

1.2 Task Description

The Rapid Prototyping of Application Specific Signal Processors (RASSP) program is an ARPA/Tri-Service initiative intended to dramatically improve the process by which complex digital systems, particularly embedded digital signal processors, are designed, manufactured, upgraded, and supported[7].

VHDL modeling is an important part of this process. Abstraction can be expressed in two domains: 1) *Structural domain* 2) *Behavioral domain.* A structural description is

an interconnection of primitive components defined for a given level in the hierarchical design process. Behavioral descriptions in hardware description languages frequently are divided into two types: *algorithmic* and *data flow.* Testing of these models involves the use of a *testbench.*

We define these terms formally as follows:

Algorithmic: A behavioral description in which the procedure defining the I/O response is not meant to imply any particular implementation[6].

Data flow: A behavioral description in which the data dependencies in the description match those in a real implementation[6].

The proposed approach to library development as part of the RASSP program has the following features:

1. The libraries consists of primitives which support two of the RASSP program specific applications: the synthetic aperture radar algorithm (SAR) and the Automatic target recognition algorithm (ATR). It uses specific constructs in VHDL that speed up the simulation of the VHDL model.

2. It describes techniques for efficiently using high-level tools so as to relieve the modeler of the complex details of model development. Usage of tools will also offer better readability of the model.

3. Models can be developed either behaviorally or structurally. With behavioral model development, a single high-level description of the model has to be developed while in structural model development, a library of primitives is used. This thesis concentrates on the task of model development which can be either behavioral or structural using various tools.

4. As a result of the necessity for floating point arithmetic in the RASSP program different tools have been studied which can generate VHDL code supporting real data types.

1.3 Contributions

This thesis develops a methodology for model generation for DSP systems. Different tools such as Matlab, Ptolemy and Comdisco SPW are employed to develop high-level models describing the behavior.

Figure 1.1 shows how a VHDL model for a DSP system can be developed and tested using different tools and employing different techniques described below:

New approaches which are investigated as part of this methodology are:

Integrating Matlab code into a VHDL model: This approach avoids writing huge VHDL programs by making use of the DSP functions which are already available in the digital signal processing toolbox in Matlab.

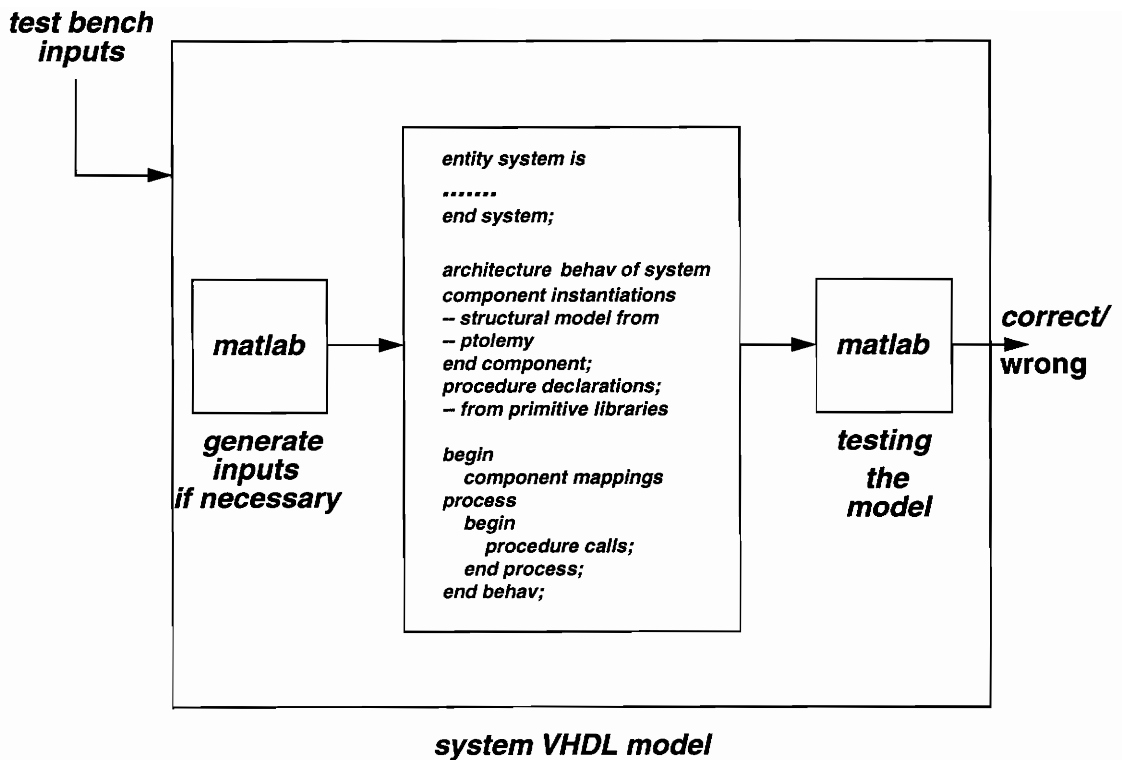


Figure 1.1 methodology developed

This approach is used to verify the functioning of the VHDL primitives by assuming that the matlab results are accurate. It is also used to test a portion of the system before the whole system is developed.

The thesis describes how *Ptolemy*, a new CAD tool being developed by University of California at Berkeley is used to overcome problems faced in generating VHDL code with other tools by supporting the real data type in VHDL.

The thesis describes some of the *coding techniques* which are used not only to increase the simulation efficiency of a VHDL model, but also to make the models developed as part of the primitive libraries easier to read and maintain.

As part of the work done at Virginia Tech towards the RASSP program, a typical *synthetic aperture radar image processor* has been developed in VHDL using a behavioral approach. The model is developed using procedural calls to the primitives which are built as part of the digital signal processing primitives library. This model doesn't use any internal signals in order to increase the simulation efficiency. It consists of different basic blocks: Video to baseband I/Q conversion, Range compression and Azimuthal compression. It is developed on the lines of a typical SAR model described in the MIT Lincoln Labs SAR benchmark. It is tested by comparing the outputs of this VHDL model with the outputs of a SAR image processor written in 'C' at MIT Lincoln Labs.

1.4 Design cycle for a DSP system

Figure 1.2 shows how a person can design and verify a DSP system using the approaches defined in this thesis.

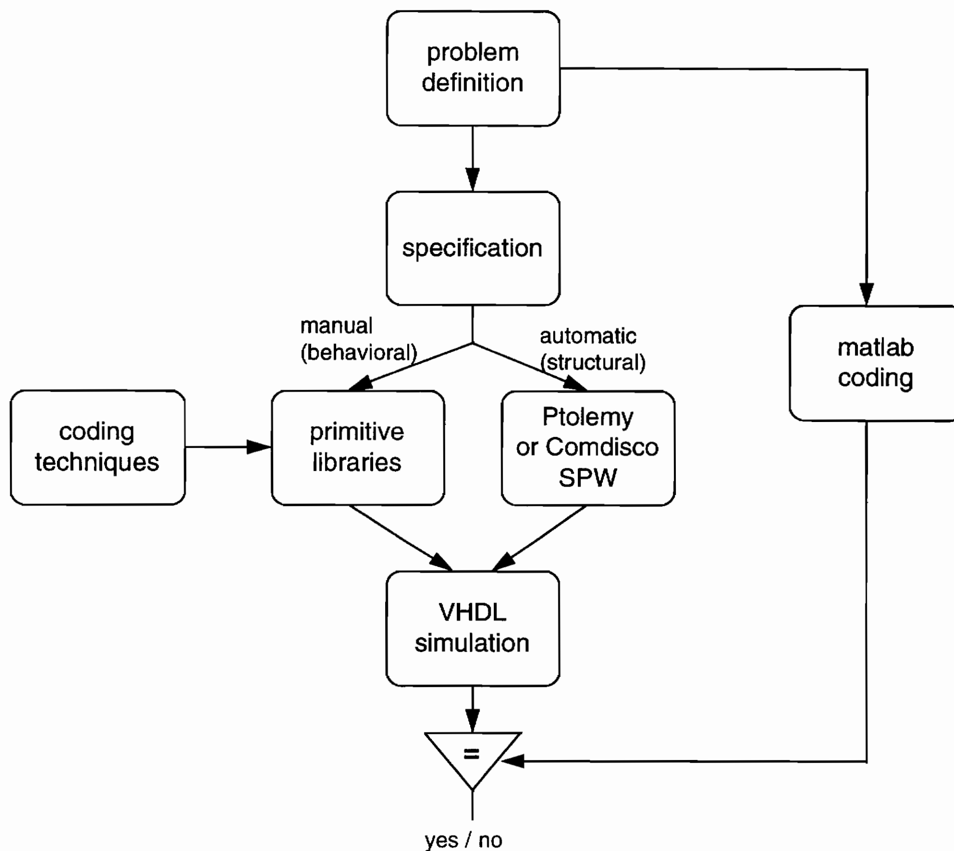


Figure 1.2 Design cycle

The first phase of any design cycle is the problem definition. The user can describe the design according to the specifications in two methods:

1) *structural*

2) *behavioral.*

Structural: Referring to Figure 1.2, the user can generate structural models using CAD tools such as Ptolemy or Comdisco SPW. With these tools the user can select blocks from the palette and interconnect them to form a whole system. The VHDL output option with these tools is used to generate structural VHDL code with component instantiations

for each of the blocks used in the system. One problem with Comdisco SPW is that it doesn't generate VHDL code supporting real data types. This is due to the fact that commercial tools are designed for generating synthesizable VHDL code. Chapter 6 of this thesis explains how this problem can be solved using Ptolemy.

It explains how new stars in Ptolemy were built and interconnected to form a filter. This schematic was used to generate VHDL supporting real data types. The VHDL generated was simulated using the Synopsys VHDL analyzer & simulator. The code, schematics for the application (based upon an example in Ptolemy) and results from the VHDL simulations are presented in Chapter 6.

Behavioral: As shown in Figure 1.2, the user can also develop his own libraries supporting required data types. In this thesis, we developed a library that contains all the primitives needed to support two RASSP specific applications: SAR and ATR. Since these applications were very simulation intensive there was a need to investigate some specific VHDL coding techniques to increase the simulation efficiency.

Some alternatives such as using alias and record data types have been investigated. The improvements obtained using some of the above coding techniques are described in this thesis. For the sake of clarity and readability overloaded primitive functions and procedures should be created. These coding techniques are described in Chapter 6.

Chapter 3 of this thesis describes library primitives which are essential for a digital signal processing application. The primitives relevant to the two main RASSP applications, SAR and ATR are discussed. Some of the primitives are one dimensional fast fourier

transform, one dimensional discrete fourier transform etc.. This chapter explains the DSP functions along with the data structures and the pseudo code that can be used to implement the routines.

Chapter 4 illustrates how a DSP application can be developed using primitives from the DSP primitives library. It elaborates on the above methodology with SAR, one of the RASSP applications as an example. As part of this thesis SAR was implemented using the library primitives. Chapter 4 gives a brief explanation of the SAR algorithm and the primitives needed to support it. This algorithm is based on the SAR benchmark from MIT Lincoln Labs[4]. Appendix C contains code for the SAR (synthetic aperture radar) application. Appendix A contains VHDL code for the library primitives. Matlab can be used to compare the results obtained from the application with the golden results.

Chapter 5 illustrates the above methodology using ATR as an example. It gives a brief explanation of the ATR algorithm and suggests some of the primitives essential to support this application. This algorithm is based on the MSE classifier from Sandia National Labs[5].

Verification: After generating or writing his own VHDL code the user simulates the model using a VHDL simulator. The results need to be compared with results obtained from simulating a gold model. As shown in Figure 1.2, Matlab can be used as a source for golden models. The inputs applied to the gold model and the VHDL model are the same. Chapter 6 of this thesis explains how matlab, which consists of a wide range of DSP functions, is used to verify the VHDL simulation results. This can be accomplished using the simulation control language provided with any VHDL simulator.

It also explains how parts of a model can be verified without the need for a complete design. This can be accomplished using Approach 2 as described in Chapter 6. It runs that part of the application that hasn't been developed in Matlab and then applies these results as inputs to the MUT. This approach uses the monitor command of the simulation control language to accomplish the task.

As a whole this thesis describes the methodology of Figure 1.2 for developing and verifying a DSP application.

1.5 Thesis Organization

The thesis contains an abstract, six chapters, bibliography and appendices.

Chapter 2 provides an overview of models, and various features of libraries.

Chapter 3 discusses the techniques used in developing primitives. It describes the typical primitives needed to support the two RASSP specific applications.

Chapter 4 describes the first RASSP application - Synthetic Aperture Radar (SAR) algorithm. It discusses the algorithm in detail along with the simulation results for the SAR VHDL model.

Chapter 5 describes the second RASSP application - Automatic Target Recognition (ATR) algorithm. It explains the ATR algorithm in detail along with the primitives needed to support this application.

Chapter 6 discusses new techniques which are employed to import models from various sources into VHDL. It explains how matlab functions are integrated with VHDL and how Ptolemy is used to generate VHDL code supporting real data types along with simulation results for both these approaches. It discusses specific VHDL constructs used in developing the primitive libraries

Chapter 7 states the conclusions drawn from the work done and also discusses possible avenues of future work.

Chapter 2. Background

2.1 Overview on VHDL

Hardware description language can be defined as a high level programming language with specialized constructs for modeling hardware. An algorithmic description is an executable VHDL model in which the procedure defining the I/O response is not specific to any particular implementation[6].

2.2 Basic VHDL models

Figure 1 shows a typical VHDL modeling environment. The stimulus generator drives the model under test with test vectors. The model response is compared with the

expected response by a comparator and Go/No Go signals are generated.

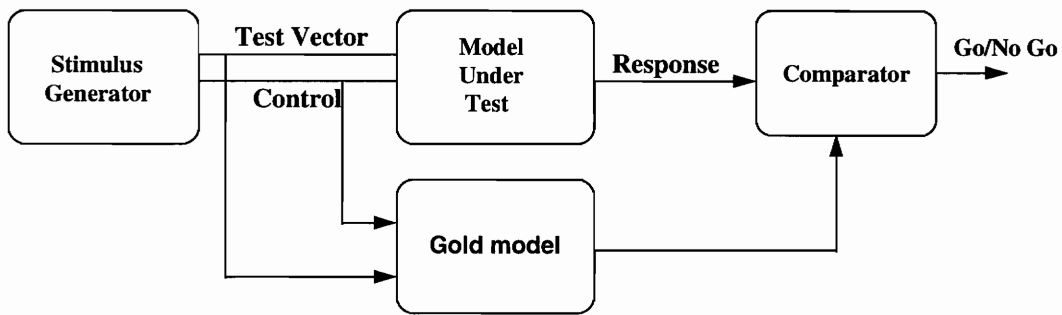


Figure 2.1. VHDL modeling environment

The following structural and behavioral definitions are taken from [6].

"The model generation abstraction can be expressed in two domains, which are defined as follows:

1. Structural domain

A domain in which a component is described in terms of an interconnection of more primitive components. The structural form of the design hierarchy implies a design decomposition process. This is because at any level we choose, the system model is composed by interconnection of the primitives defined for that level. A question that

should be answered at this point is: How are these primitives defined ? The answer is that they are frequently defined in terms of primitives at a lower level. Figure 2. shows that

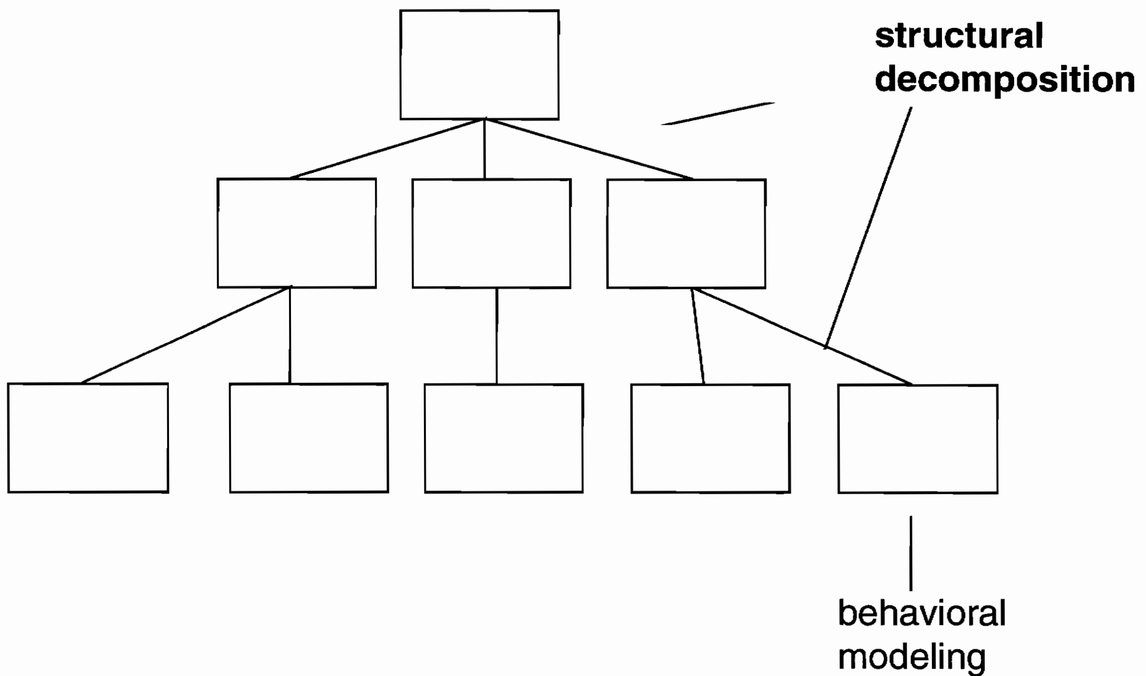


Figure 2.2. Structural decomposition

a design can be represented as a tree, with the different levels of the tree corresponding to levels in the abstraction hierarchy. The lowest level component is always a behavioral description.

Two concepts related to the design tree are those of *top-down and bottom-up design.* The word "top" refers to the root of the tree, while "bottom" refers to the leaves. In the *top-down design,* the designer begins with knowledge of only the function of the root. The root should be partitioned into a set of lower level primitives. The design then proceeds to the next lower level, where partitioning using primitives at this new level takes place. This process continues until the leaf nodes of the design are reached. Partitioning at each level is optimized according to some objective criterion such as cost, speed and chip area. The main point in top-down design is that the partitioning is not constrained by "what's available".

The term *bottom-up design* refers to partitioning conditioned by what is available. The designer must partition based on what parts will be available i.e.. lower parts of the tree will have been designed previously, perhaps on another project, and the designer is constrained to use them. The disadvantage of the top-down design is that it may produce components which are not "standard" thus increasing the cost of design. Bottom-up designs are more economical, but they do not meet the objective performance criterion as well as top-down designs. Most of the real designs currently are a mixture of top-down and bottom-up techniques.

2. Behavioral domain

Behavioral domain is a domain in which a component is described by defining its input/output response. There are two types of behavioral descriptions: *algorithmic and data flow.*

An algorithmic description is merely a procedure or program written to model the behavior of a device, to check that it is performing the function correctly without being concerned about how it is going to be built. General algorithmic model development in the behavioral domain involves the following steps:

1) Map sentence groups in the requirements to VHDL processes
2) Assign an activity list to each process
3) Develop VHDL code that implements the activity

The factors that should be taken into consideration for trade-off's between using more or fewer processes in an algorithmic description are:

1) *Number of signals:* Using more processes requires more signals. More signals will require more queue processing which effects the simulation efficiency adversely.

2) *Individual process complexity:* The process code may not account for all possible combinations of input changes. Using simpler processes can solve this problem.

3) *Ease of mapping:* Usage of more processes will allow for a very natural mapping between specification requirements and processes.

Data flow descriptions show how data flows between registers. Both the algorithmic and data flow descriptions can be hardware description language implementations of behavior at any level"[6].

2.3 Library features

The purpose of the VHDL library development is to allow a system designer to develop high-level VHDL models of a signal processing or an image processing system using the primitives from the library. The components of the VHDL library are:

1) digital signal processing primitives
2) image processing primitives

which are used to support the two RASSP specific applications:

1) Synthetic aperture radar image processing algorithm
2) Automatic target recognition algorithm

1. *Digital signal processing primitives library:*

A *digital* signal is defined as a function in which both time and amplitude are discrete. A digital signal can always be represented by a sequence of numbers in which each number has a finite number of digits. Mathematical operations on digital signals can be called as *digital signal processing.*

"A *digital filter* is a computational process in which the sequence of input numbers is converted into a sequence of output numbers representing the alteration of the data in some prescribed manner. A common example is the process of retaining a certain range of frequencies in a signal while rejecting all other frequencies, which is one of the foremost classical approaches to analog filter design. In the digital case this type of filtering can be achieved completely by the process of digital addition, multiplication by constants, and delay"[3].

"The techniques of spectral analysis employing *fourier transforms* and series have long represented an important area of applications in continuous-time signal processing. The spectrum of signals containing many thousands of sample points can be achieved in a matter of milliseconds. Some of the varied scientific disciplines in which the fourier

transform has opened new potential applications are oceanographic analysis, communications signal analysis, and statistical analysis"[3].

2. *Image processing primitives library:*

"Modern technology utilizes all types of pictures, or images, as sources of information for interpretation and analysis. These may be portions of the earth's surface viewed from an orbiting satellite, or schematic line drawings of electronic circuitry. *Image processing* combines computer applications with modern scanning techniques to perform various forms of image enhancement, distortion correction, pattern recognition, and objective measurements. The two important operations associated with image processing are: Neighborhood operations and Point operations.

"*Neighborhood* operations, as the name implies, are operations that modify the value of a picture element in a way that depends on the value of neighboring picture elements. The most important and general neighborhood operation is convolution. For sampled data images encountered in digital image processing, convolution is given by

$$P(x,y)=\sum_{\varepsilon=-J}^{J}\sum_{\eta=-K}^{K} I(\varepsilon,\eta)\,S(x-\varepsilon,\,y-\eta)$$

where I is the image function, S is the convolution function, P is the convolved image, ε and η are dummy variables, K and J are the horizontal and vertical spatial dimensions, respectively.

The importance of convolution is that it offers a method of performing spatial filtering of images in real time or nearly real time.

Point operations are carried out the same way for each picture element in the image, independently of its position or the value of its neighboring elements. Examples of point operations are arithmetic combinations of two images (e.g., addition, subtraction, multiplication and division); and histogram equalization"[3].

Chapter 3. Library Primitives

3.1 Overview

This chapter describes in detail the methodology followed for the development of behavioral and structural models. The tools that have been used for the development of the primitives are Comdisco SPW, Ptolemy and Matlab. Comdisco SPW can generate both behavioral and structural models for the schematics developed using blocks from the HDS library. Ptolemy and SPW operate under the X11 windowing environment and use both textual and graphical windows which can be manipulated with the keyboard and the mouse of a workstation. These tools replace the tedium and complexity of writing VHDL with concise and easily read graphic languages.

The disadvantages with these tools are:

1) Ptolemy generates structural VHDL models which use signals as interconnects between components and signals require queue processing for execution. This has a

serious effect on the execution times of the models. The future versions of Ptolemy have a VHDL domain which is capable of generating procedure call based behavioral VHDL models with just the input/output ports as signals.

2) The models should support real data types in VHDL. But Comdisco SPW is capable of generating VHDL code supporting bit vector and integer data types, which can be read into any synthesis tool to provide a gate level layout or even layouts further down the design cycle.

3.2 Characteristics of the primitives

Any model should also have the following two features in order to help in rapid development of models and parametrization :

1. The models should support reuse - models developed for a particular application should be able to be used for other similar applications.
2. The models should support the *model year concept* - The model year concept is based on a successive refinement approach, where the features of the models are improved from year to year. The main requirements as a part of the model year concept are:

- Model development should allow for quick development of changes in the application areas from year to year.
- VHDL modules should allow for easy modifications to reflect the model year changes in the model.
- Use of realistic data files as inputs to the VHDL model.

3.3 Development of library primitives

There are two main RASSP specific applications for which the primitive libraries have been built:

1) Synthetic aperture radar image processor

2) Automatic target recognition algorithm

The primitive libraries developed to support these two applications are:

1) *digital signal processing* primitives library to support the synthetic aperture radar image processor.

2) *image processing* primitives library to support the automatic target recognition algorithm.

The SAR VHDL model has many generics such as the number of pulses, size of the input array, the number of filter coefficients used in the FIR filter etc. These are given as generics to the VHDL model so that the same VHDL model can be used for different types of test benches. The VHDL model for the synthetic aperture radar image processor is a behavioral model developed using procedure calls. These procedures are the primitives in the digital signal processing library for the synthetic aperture radar image processor. In the case of the automatic target recognition algorithm the procedures are primitives in the image processing primitives library. These primitives should be modeled in such a way that the input array can be of any length. The output array should be properly mapped to the data structure used in the procedure call. This supports the model year concept i.e.. the same model can be used in different applications.

3.3.1 Fast Fourier transform

The fast Fourier transform is a high speed algorithm for computing the fourier transform of a discrete-time signal. In order to fully evaluate a Fourier transform or inverse transform with digital operations, it is necessary that both the time and frequency functions be samples in one form or other. Fourier and inverse transforms can be considered in four combinations obtained by successively assuming that the time and frequency variables are either continuous or discrete. The one that is of primary interest in digital computation is the fourier pair with both the time and frequency variables being discrete[3]. The discrete fourier transform pair can be stated as:

$$X(m)=\sum_{n=0}^{N-1} x(n) W_n^{mn} \quad \textit{forward}$$

$$x(n)=\frac{1}{N}\sum_{m=0}^{N-1} X(m) W_n^{-mn} \quad \textit{inverse}$$

$$\textit{where} \quad W_n = e^{-j(2*\pi)/N}$$

The time domain signal is denoted as x(n) and the frequency domain function as X(m). In this sense n represents a time integer and m represents a frequency integer. The fast fourier transform is not a different transform from the discrete fourier transform but it represents a means for computing the discrete fourier transform with a considerable reduction in the number of computations. The convolution process in the azimuth compression stage of the SAR algorithm can be done using the overlap-save method. The fastest way to do convolution is to fourier transform the inputs, multiply the results of the fourier transforms and then perform an inverse fourier transform on the multiplication

result. Hence the fourier transform primitive developed as part of the digital signal processing library should support both the fourier transforms in the range compression and azimuth compression stages. The number of discrete samples in the range compression stage is equal to 1024 and the number of discrete samples in the azimuth compression stage is equal to 2048. Hence the data structure of the input array to the fast fourier transform can be an array of POSITIVE range of numbers.

Comdisco SPW has blocks for the FFT in its HDS library. The VHDL link can be used to generate VHDL code for fast fourier transform with any data type except for real. The block is shown in figure 3.1.

The "fft" VHDL primitive incorporates both the forward fourier transform and the inverse fourier transform.

type ARR1_RE_TYP is array(POSITIVE range <>) of REAL;
procedure fft(DIN: in ARR1_RE_TYP; ISIGN: in INTEGER; DOUT: out ARR1_CMPLX_TYP);

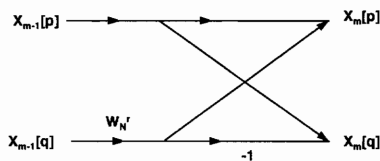
If ISIGN = -1 it's a forward transform and if it is equal to 1 it's an inverse transform. The inverse fourier transform has the same operations as the forward transform except that the data obtained at the end of these operations is divided by the total number of samples, and the sign is changed in the exponential term.

The exponential term in the primitive is represented as shown below:

```
type COMPLX_TYP is record
    RE: REAL;
   IMAG: REAL;
end record;
variable TEMP: COMPLX_TYP;
TEMP:=(
cos(REAL(m*n*ISIGN)*2.0*π / REAL(N), sin(REAL(m*n*ISIGN)*2.0*π / REAL(N)
)
```

$$W_N^{-mn} = \cos(2*\pi*m*n/N) - j*\sin(2*\pi*m*n/N)$$

The above expansion is used to incorporate both the exponential term and the sign change for the forward and inverse transforms. The fast fourier transform is decomposed into butterfly computations. The basic butterfly computation and the associated VHDL code is shown below:



TEMP := W*DATA1(J);-- representing W_N^r

DATA1(J) := DATA1(X)-TEMP;-- representing $X_m[q]$

DATA1(X) := DATA1(X)+TEMP;--representing $X_m[p]$

The full source code for the Fast Fourier Transform, named FFT_SIG, is given in Appendix A.

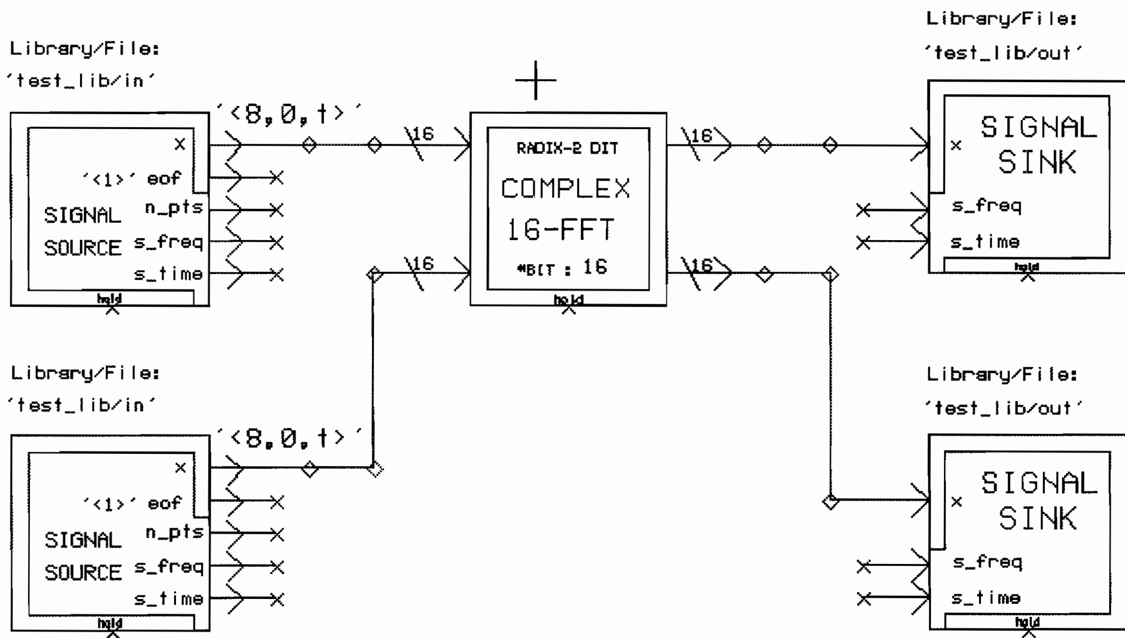


Figure 3.1. SPW schematic

3.3.2 Convolution

One of the most important applications of the FFT is that of *high speed convolution*, a process made possible by the simplicity of the corresponding relationships in the transform domain. The convolution of two signals x(n) and h(n) is denoted by x(n)*h(n) and is defined as

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

By the use of the fourier transform operation this expression is equivalent to:

$$x(n) * h(n) \longleftrightarrow X(m)H(m)$$

where X(m) and H(m) are the fourier transforms of x(n) and h(n) respectively. If x(n) and h(n) are relatively long functions, the number of computations required to perform a direct convolution can become excessively large. However according to the previous equation this operation is equivalent in the transform domain to simply multiplying the fourier transforms of the two signals. The three basic steps used to perform high speed convolution are:

1) The fourier transforms of the two signals are computed using the fast fourier transform algorithm

2) The transforms of the signals are multiplied together at all pertinent frequency points

3) The inverse transform of the product is computed, again using a fast fourier transform algorithm

The convolution operation is equivalent to an FIR filter. Hence this primitive uses the filter primitive. If the inputs to this primitive are X and Y arrays the pseudo code which represents the convolution operation is shown below:

```
if (X'high > Y'high) then
    TEMP := X; if index <= X'length
    else TEMP := 0.0 until Index = X'length+Y'length-1
   FILT_VAR(TEMP, Zeros, Y, Z);
else
    TEMP := Y; if index <= Y'length
    else TEMP := 0.0 until index = X'length+Y'length-1
   FILT_VAR(TEMP, Zeros, X, Z);
end if;
```

The "Zeros" variable is an array of zeros to derive the FIR filtering operation from the filter primitive because the denominator coefficients for an FIR filter are zeros. The full source code for one dimensional convolution, named CONV_SIG, is given in Appendix A.

3.3.3 Finite impulse response filter

A *digital filter* may be defined as a computational process or algorithm that converts one sequence of numbers representing an input signal into another sequence of numbers representing an output signal , and in which the conversion changes the character of the signal in some prescribed fashion. The associated transfer function will often be referred to as a digital transfer function[3]. In the z-domain the discrete transfer function can be represented as:

$$H(z)=\frac{\sum_{i=0}^{k} a_i z^{-i}}{1+\sum_{i=1}^{k} b_i z^{-i}} \quad \textit{equation } 3.1$$

where a's and b's determine the desired behavior. A *finite impulse response* filter is one in which the impulse response h(n) is limited to a finite number of samples defined over the range

$n_1 \leq n \leq n_2$

where n_1 and n_2 are both finite. FIR filters have both advantages and disadvantages as compared with *infinite impulse response* (IIR) filters. Among the advantages of FIR filters are: they can achieve ideal linear phase characteristics and they have less susceptibility to quantization effects. Possible disadvantages are they lack simple design procedures that would permit direct attainment of filter specifications[3]. The transfer function of an FIR filter can be expressed in the form of

$$H(z)=a_0+a_1 z^{-1}+a_2 z^{-2} \ldots .+a_k z^{-k}$$

where k represents the order of the function.

The transfer function of an IIR filter can be expressed in the form of

$$H(z)=\frac{1}{1+b_1 z^{-1}+b_2 z^{-2} \ldots .+b_k z^{-k}}$$

where k represents the order of the function.

From the three equations it is inferred that:

if $b_1, b_2, b_3 ... = 0$ in equation 3.1 then it's an FIR filter

or

if $a_1 = 1, a_2 = 0, a_3 = 0 ... a_k = 0$ in equation 3.1 then it's an IIR filter.

The filter transfer function is represented mathematically in terms of adders, multipliers and delay elements as:

$$y(n)= a_0 x(n)+ a_1 x(n-1)+ a_2 x(n-2)...+a_k x(n-k)-b_1 y(n-1)-b_2 y(n-2)...b_k y(n-k)$$

where x(n) is the input signal, a's represents the numerator coefficients and b's represent the denominator coefficients.

The above equation is coded in VHDL for the filter primitive with the input signal, numerator coefficients, denominator coefficients as input ports and the output signal as an output port. Hence the same filter primitive is used for an IIR or an FIR filter. The full source code for a generic filter, named FILT_SIG, is given in Appendix A.

3.3.4 Two dimensional convolution

The most common class of processes on images is convolution which is equivalent to two dimensional linear filtering. Convolution is used for image sharpening, noise removal, edge detection and image processing. It can also be used for feature detection and object matching and registration[2]. The equation below employs a two dimensional impulse response, h(m, n) and can be described as

$$g(p,q)=\sum_{m=0}^{N-1}\sum_{n=0}^{N-1}h(m,n)f(p-m,q-n)$$
,

where f(m,n) is the input image. h(m,n) is overlaid on the rotated, translated input image and a point to point multiplication is performed. Each point in the output image is generated by the summation of the multiplications. Convolution can be thought of as a process of sliding a kernel over an image. For fast two dimensional convolution the two dimensional fast fourier transform can be used to speed up this convolution process. The steps in a two dimensional convolution operation using FFT's are:

1) perform a 2D real FFT on the input image
2) multiply the image FFT result with the FFT of the filter kernel
3) perform a 2D inverse FFT on the complex product

But in practical applications this method is not used because the hardware complexity and cost required for the FFT computations is usually much greater than the direct convolution hardware[2]. Therefore, our two dimensional convolution primitive doesn't use the above fourier transform method. The pseudo code for the two dimensional convolution operation is given below:

h(m,n) is overlaid on the rotated, translated input image (f) and a point to point multiplication is performed. Each point in the output image (CONV_OUT) is generated by the summation of the multiplications. The full VHDL source code for two dimensional convolution, named CONV_VAR, is given in Appendix A.

```
for J in h'HIGH(2) loop
  for I in h'HIGH(1) loop
    for K in 0 to f'HIGH(1) loop
      for L in 0 to f'HIGH(2) loop
        CONV_OUT(I+K, J+L) := CONV_OUT(I+K,J+L) + (f*h(I,J))(K,L);
      end loop;
    end loop;
  end loop;
end loop;
```

3.3.5 Two dimensional fast Fourier transform

As in the case of a one dimensional Fourier transform the two dimensional Fourier transform would not be very practical if there were not a method to decrease the number of computations required in its calculation. The two dimensional fast Fourier transform is based on the one dimensional FFT and is separated into two sums. It is described by the following equation:

$$F(u,v)=\frac{1}{MN}\sum_{m=0}^{M-1}\exp(\frac{-j2\pi um}{M})\left\{\sum_{n=0}^{N-1}f(m,n)\exp[\frac{-j2\pi vn}{N}]\right\}$$

If the value of m is considered a parameter, the summation inside the brackets { } in this equation can be considered as a one dimensional fast fourier transform. This technique is used equally well for the inverse 2D FFT. Hence the pseudo code for the two dimensional fast fourier transform is represented as:

```
loop
   TEMP := DATA_IN(row);
   FFT(TEMP, ISIGN, TEMP1);
   INTER_DATA(row) := TEMP1;
end loop;
loop
   TEMP2 := INTER_DATA(column);
   FFT(TEMP2, ISIGN, TEMP3);
   DATA_OUT(column) := TEMP3;
end loop;
```

ISIGN is equal to -1 for forward transform or it is equal to 1 for inverse transform. If it is equal to 1 DATA_OUT is divided by M*N where MxN is the size of the input image. The full VHDL source code for two dimensional fast fourier transform, named FFT_VAR, is given in Appendix A.

Chapter 4. Synthetic Aperture Radar

4.1 Introduction

This chapter describes the first of the two RASSP applications for which primitives have been developed. This first application is the Synthetic Aperture Radar (SAR). SAR is an important tool for the collection of high-resolution, all-weather image data. The SAR algorithm is used to provide a visual representation of the target and the surrounding environment in the form of a high resolution spatial map. The spatial map can be used for navigation and targeting and may also be recorded for post-flight analysis and intelligence gathering. This visual representation is generated based on the radar returns from the target. As part of this thesis a working model of a synthetic aperture radar image processor has been developed in VHDL. The full source code
of the SAR model is given in Appendix C.

4.2 SAR

4.2.1 Input data

The SAR input generation problem is taken from [8].

"Figure 4.1 shows the basic method that is used to generate input data to the SAR image processor model. The radar is fully polarimetric with interleaved H-pol and V-pol transmission and simultaneous H-pol and V-pol reception. The transmitted signal is a sequence of "chirp" pulses generated by modulating a high-frequency RF carrier signal with a ramp signal. The received signal returned from the target is a complex signal and is basically the transmitted signal with a suitable delay. The main components of the SAR sensor model are the down converters, deramp section and A/D converter.

The inputs of the SAR sensor model are the technical parameters of the radar such as squint angle, carrier frequency, swath width, nominal range, pulse repetition frequency, bandwidth and pulse width of the transmitted signal. Some of these parameters are used for producing the transmitted signal while other parameters are used in the various components.

The functions of the various components are: :

Down-converter: The down-converter is used to remove the high-frequency carrier signal component from the transmitted and the received signals, thereby converting them to their baseband frequencies.

Deramp section: The deramp section is basically a mixer whose two inputs are the complex-conjugate of the down-converted transmitted signal and the down-converted received signal.

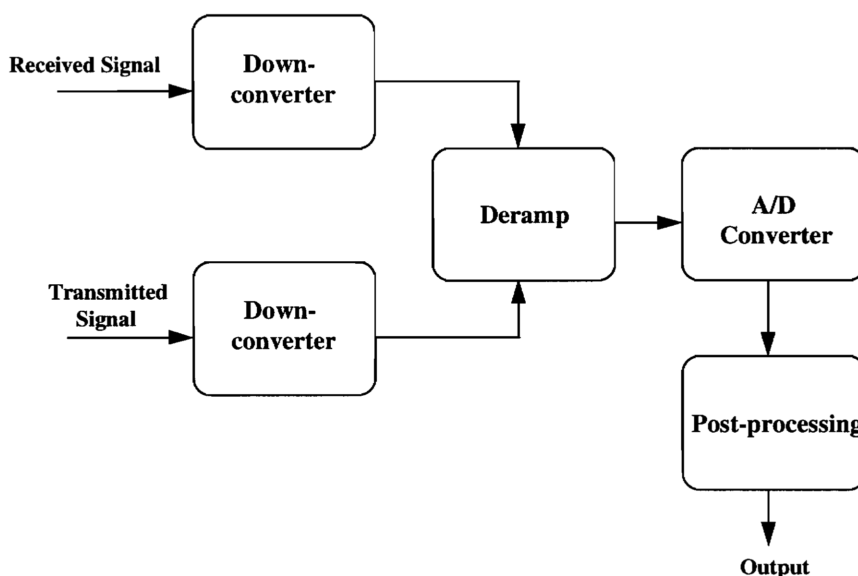


Figure 4.1. SAR Input generation[8]

A/D Converter and Post-processing: The real data is passed through the A/D converter where it is digitized. The post-processing block is responsible for gathering the output data and producing displays using MATLAB or any other similar tool"[8].

The problem of SAR input generation is described in Ms. Zhen Xu's thesis [8].

4.2.2 Image processor for SAR

The SAR algorithm explained below along with the figures is reproduced from MIT Lincoln Labs SAR benchmark[4]. As part of the RASSP project at Virginia Tech the SAR model is implemented on the lines of the algorithm described in the benchmark[4]. The model is tested using 4x4 and 16x16 arrays provided by the MIT Lincoln Labs.

In SAR operation, images are formed continuously from three of the four available polarizations. Figure 4.2 illustrates the processing flow. The image processing flow is composed of three main blocks which are: 1) Video to I/Q baseband conversion which includes the blocks "convert to baseband", "upper sideband filter", "form I/Q samples" blocks from figure 4.2, 2) Range compression which includes the blocks "taylor weighting", "dft" blocks from figure 4.2, 3) Azimuth compression which includes the blocks "collect range pulses", "shift frame into processing array", "dft range gates", "kernel multiply", "idft range gates" blocks from figure 4.2. Figure 4.3 illustrates the processor model.

4.2.3 Pulse Repetition Interval Detection

"Channels of polarized pulse data, header data, and Aux data are presented to the RASSP processor. A 20 word sequence consisting of a 13-word Barker code with 5 leading zeros and 2 trailing don't care words indicate the start of pulse data. This preamble is followed by 2032 words of 11-bit even samples and 11-bit odd samples, sign extended to 12 bits. Included with the pulse samples are header data and Aux data recorded in bit-serial fashion and duplicated in bit positions 3,16,19, and 32 of the 40-bit data word. Header

data describes the polarization of the pulse data, and Aux data contains ancillary navigation and radar data. Pulse data for the four polarizations are output in a repeated sequence (i.e., ..., HH, HV, VH, HH, ...), but Aux data are written for the leading pulse of the sequence (i.e., HH). There are filler data between the end of data for one pulse and the start sequence for the next pulse. Because the HV and VH polarizations contain the same information, no more than three of the four polarizations are used to form images.

4.2.4 Video to Baseband I/Q conversion

Prior to pulse compression, 4064 real video samples of each of the three polarizations to be imaged must be converted to complex (in-phase and quadrature, or I/Q) data at baseband. The approach for performing the I/Q demodulation is to form sequences of even and odd samples and modulate each sequence by $(-1)^n$. This yields two real valued sequences for each pulse: an even sample sequence and an odd sample sequence. The even and odd sequences are passed through an 8-coefficient FIR filter. The FIR filter output sequence should be 8 samples short because the filter must be initialized before valid data samples are obtained. According to the model-year concept this should be able to accommodate up to 48 coefficients. These sequences are combined to form the sequence of complex samples. The FIR filter is given by

$$y_n = \sum_{m=0}^{7} a_m x_{n+m} ,$$

where y_n is the nth output sample, x_n is the nth input sample, and a_m is the mth FIR coefficient.

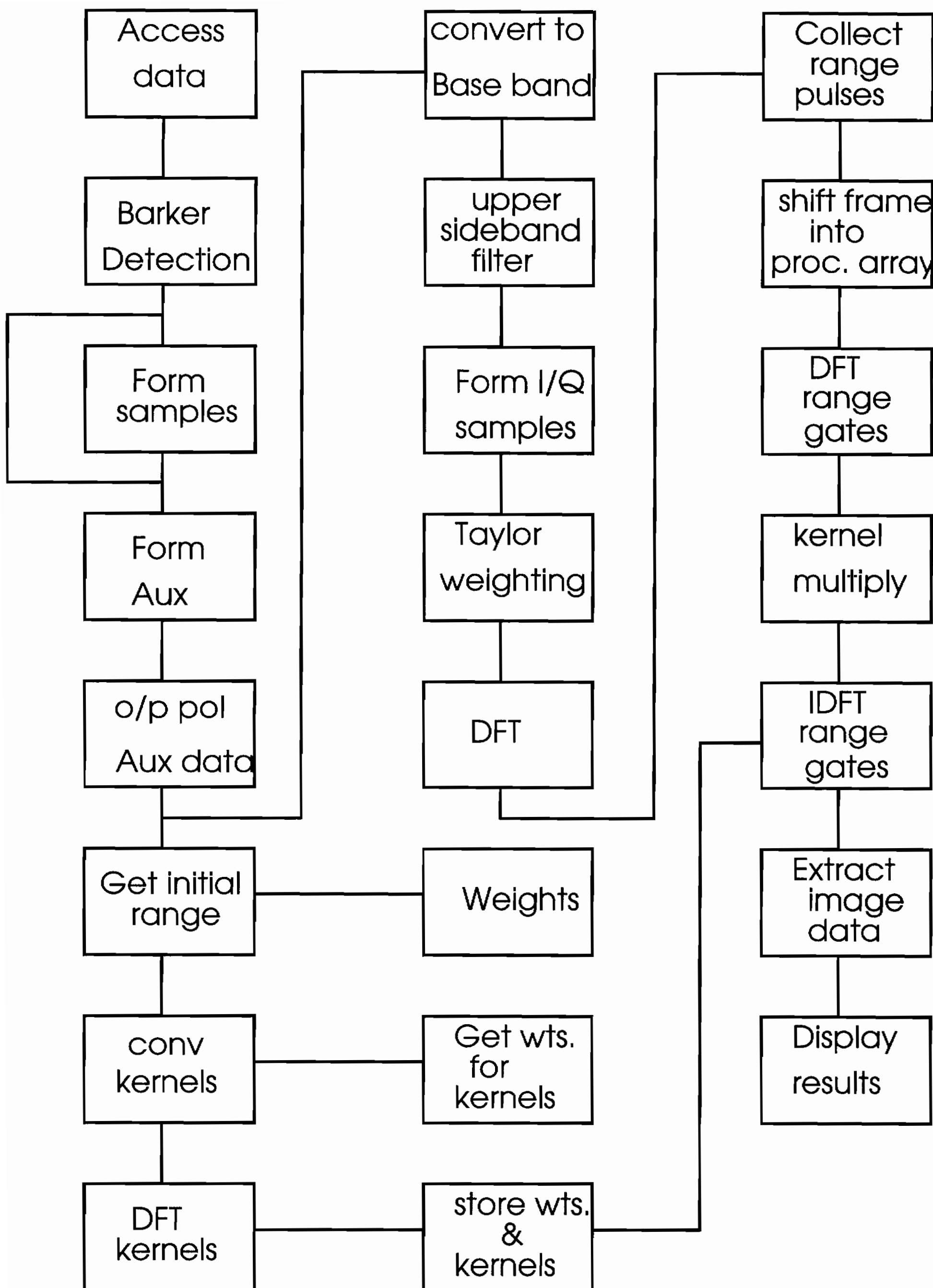


Figure 4.2. SAR image processing flow[4]

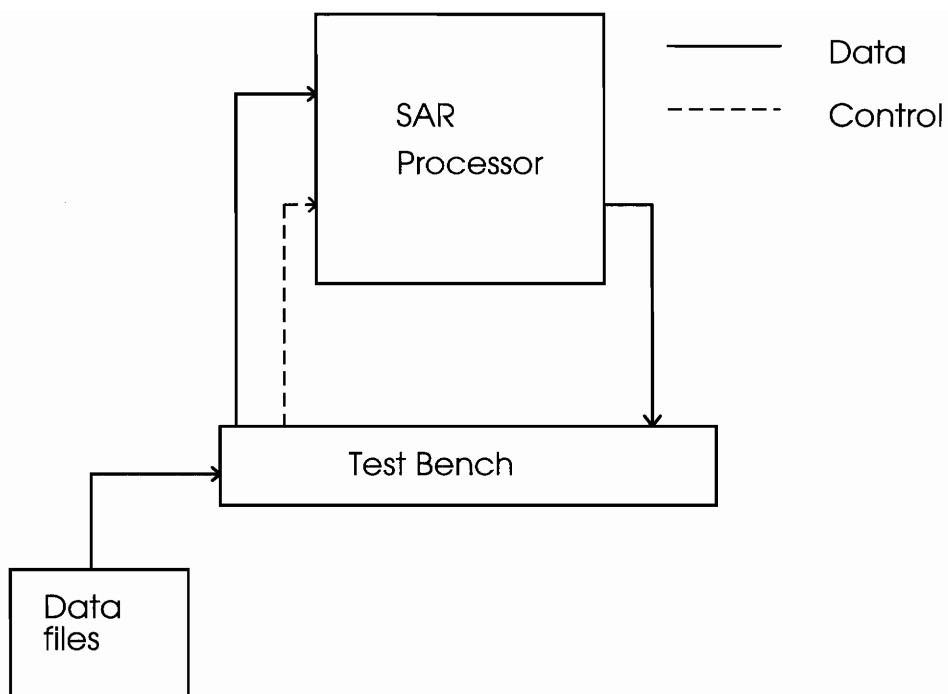


Figure 4.3. Processor Model[4]

The baseline I/Q filter coefficients are given in Table 4.1.

Index	Even sequence	Odd sequence
0	-0.021133	0.019827
1	0.055895	-0.011912
2	-0.148449	-0.067483
3	0.406139	0.917516
4	0.917516	0.406139
5	-0.067483	-0.148449
6	-0.011912	0.055895
7	0.019827	-0.021133

Table 4.1. Filter coefficients[4]

4.2.5 Range Compression

In pulse compression the I/Q samples of each pulse are transformed into a compressed range pulse. The first step in pulse compression is the application of Taylor weighting to the amplitude of the complex valued I/Q data. This weighting is used for side-lobe reduction. The complex input data are also modulated by $(-1)^n$ so that the compressed range pulses are centered in the range window. Weighted I/Q data are transformed to compressed range data using a fast fourier transform.

4.2.6 Azimuth Compression

Compressed pulses are placed in a time sequence into a 2D processing array and each row of the array is convolved with a row specific reference kernel. The convolution outputs are saved in an image array which becomes the strip map image of the SAR. In this process, compressed pulses are placed in a 2-D array referred to as a frame. Once the number of pulses have been accumulated (currently 512) to form a frame, the frame is shifted into a processing array. The processing array is a 2-D array where azimuth compression processing is performed. The processing array consists of two frames. As the new frame is shifted into the processing array the oldest frame is shifted out. A convolution is performed along each row of the processing array, where the convolution kernel is the approximate response of a point scatterer located at the range-gate of the row. The convolution kernel used for each row is selected from a database of 31 pre-calculated kernels, where the kernels have been Taylor Weighted, zero padded and fourier transformed. The choice of kernel is determined by the slant range to the middle of the most recent frame in the processing array. This slant range is given by the SLTRNG in the Aux record of the 256th pulse of the most recent frame. The kernels are calculated based on the slant range (SLTRNG) to the middle of the first frame of data obtained for a given pass, and are stored for use throughout the pass. Only 16 of the 31 kernels are used at one time in azimuth compression, but these 16 vary with each frame.

Convolution processing in azimuth compression is performed using fast fourier transforms with the overlap-save method. Each vector product of the fourier transformed row in the processing array and the convolution kernel is inverse transformed, and the last 512 samples of the each inverse transform represent valid convolution outputs and are saved in an image array.

Figure 4.4 depicts convolution processing for one row of the processing array. The processing array consists of 1024 pulses, with 2048 complex range samples each. The convolution kernels are 512 points long, but trailing zeros are used to pad-out the kernel to 1024 points. A 1024 point DFT of each row is multiplied with the 1024 point DFT of the associated convolution kernel. A new frame is shifted into the processing array after the inverse transform, the convolution process is repeated, and more are added to the image array thereby generating the output stripmap SAR image"[4].

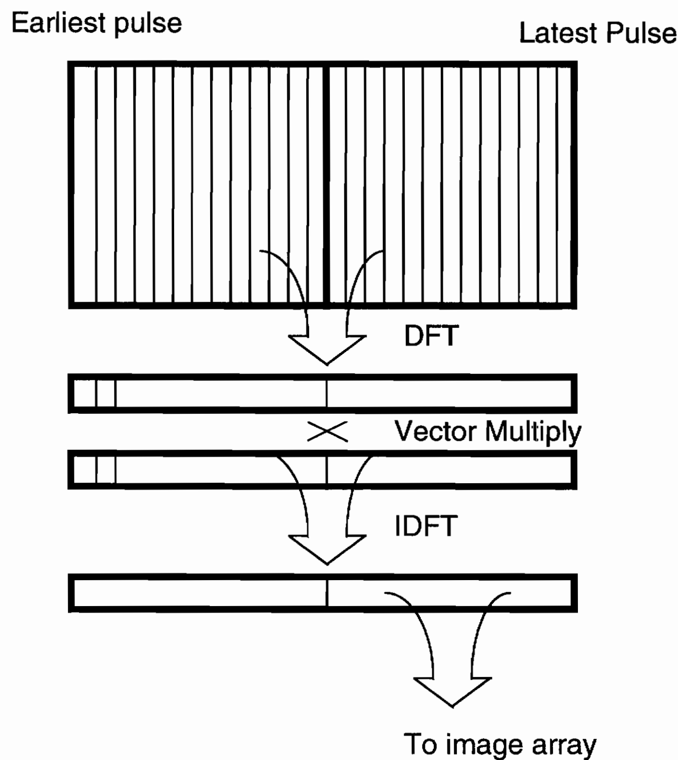


Figure 4.4. Cross-range Convolution Processing[4]

A Synthetic aperture radar (SAR) image processor is implemented in VHDL as part of the RASSP program at Virginia Tech[Appendix C]. It is tested with the input data sets (4 x 4

and 16 x 16 arrays) provided by the MIT Lincoln labs and the outputs are compared with the outputs provided by them.

4.3 Primitives for the synthetic aperture radar (SAR) image processor

The synthetic aperture radar image processor has different stages which are modeled as procedure calls in the behavioral model. The first stage is the Pulse repetition interval detection determines the rate at which the pulse is repeated. The second stage is the Video to Baseband I/Q conversion. The two odd and even samples are filtered using a finite impulse response filter with 8-coefficients and then the outputs are combined to form a single I/Q pulse. The third stage is the range compression stage in which the I/Q pulse is converted to a range gate using a 1024 point FFT and then Taylor weighting the samples. The fourth stage is the azimuth compression stage in which the pulses are collected into a processing array, corner turned and convolved with a convolution kernel. The convolution process is done using the overlap-save method i.e.. the inputs to be convolved are fourier transformed, multiplied and the result is inverse fourier transformed to obtain the convolution output.

The basic digital signal processing primitives required to support the above four stages are identified as:

1) Fast fourier transform in the range compression stage
2) Convolution in the azimuth compression stage
3) Finite impulse response filter in the Video to baseband I/Q conversion stage

The complete source code for the above primitives is in Appendix A.

4.4 SAR results

The SAR VHDL model was tested using simple data sets provided by MIT Lincoln Labs. The first test consists of four pulses where each pulse has four samples. The VBIQ and range compression operations are executed on each pulse. The range compressed pulses are collected in a processing array where the two dimensional processing array is corner turned and each row of the processing array is convolved with a convolution kernel to form the image array.

The image array formed by the SAR model is a four by four array of complex numbers. This two dimensional array for testing purpose is decomposed into a single 16 element array where the first row of the two dimensional array forms the first four elements of the 16 element array and so on.

The results from the "gold SAR" which is the Lincoln Lab's SAR model is also decomposed in the same way. Matlab is used for comparing the results. Both arrays are plotted on the same graph (Figure 4.5) with the x-axis scale ranging from 1 to 16. The curve in Figure 4.5 is formed by joining all the discrete points of the gold model results. The sharp corners of the curve indicate the gold model results. The output results of the VHDL model are indicated using '+' for clarity and are superimposed onto the graph. The variation in the results ranging from +0.0005207% to -0.0004355% can be attributed to the quantization noise.

The SAR VHDL model was also tested with a sixteen by sixteen array i.e.. there are sixteen pulses to be collected into the processing array. The sixteen by sixteen array is decomposed into a single array of 256 elements. In the same way as described before the outputs of the VHDL model and the "gold model" are both plotted on Figure 4.6.

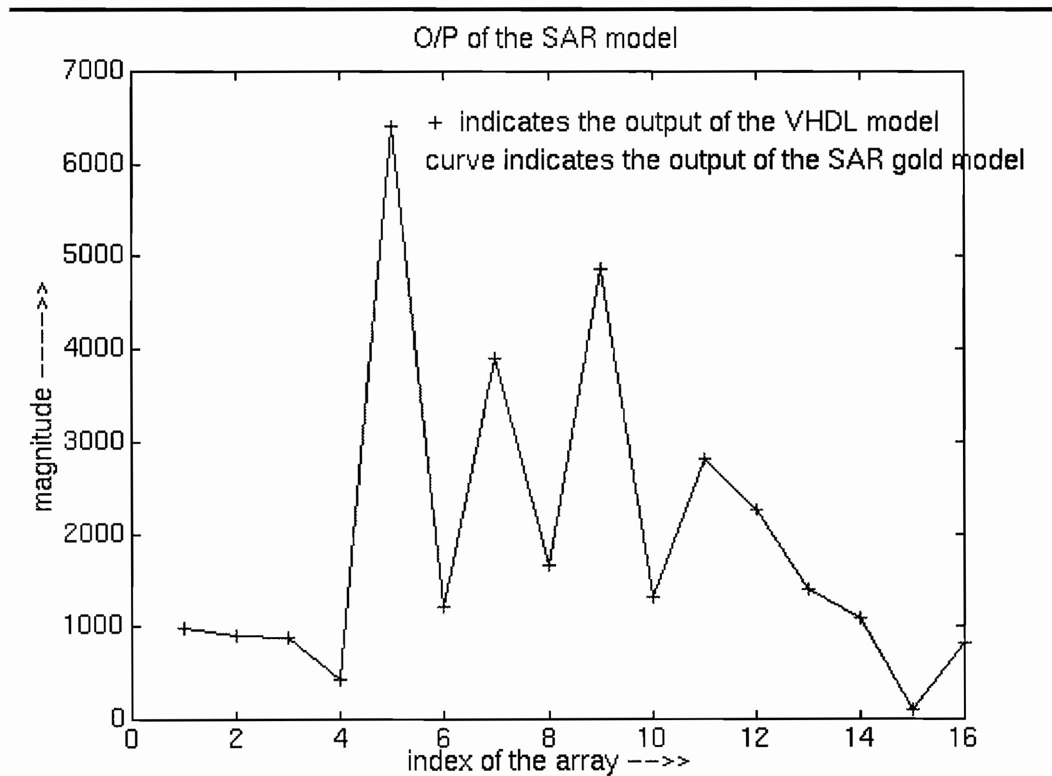


Figure 4.5. SAR simulation results for 4x4 array

For the sake of clarity only the first 25 values of the 256 element array are plotted. The variation in the results ranging from +0.0005460% to -0.0004650% can be attributed to the quantization noise.

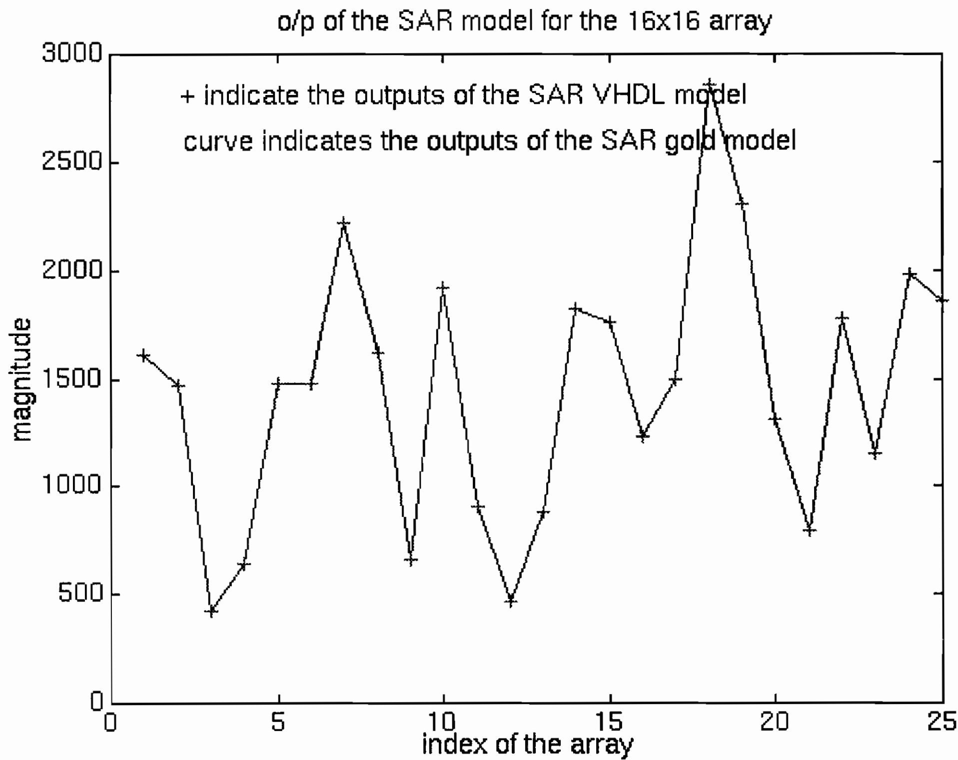


Figure 4.6 SAR simulation results for 16x16 array

Chapter 5. Automatic Target Recognition algorithm

5.1 Introduction

The automatic target recognition (ATR) algorithm is used to detect a particular region of interest in an image. The input image to this algorithm is the output of the SAR processor. The ATR algorithm is described in this chapter. The implementation of this algorithm is not a part of this thesis but the image processing primitives to support this algorithm are identified and developed as part of the image processing library.

5.2 Automatic target recognition algorithm

The system overview of the ATR algorithm is depicted in figure 5.1.

The automatic target recognition algorithm can be decomposed into three individual stages:

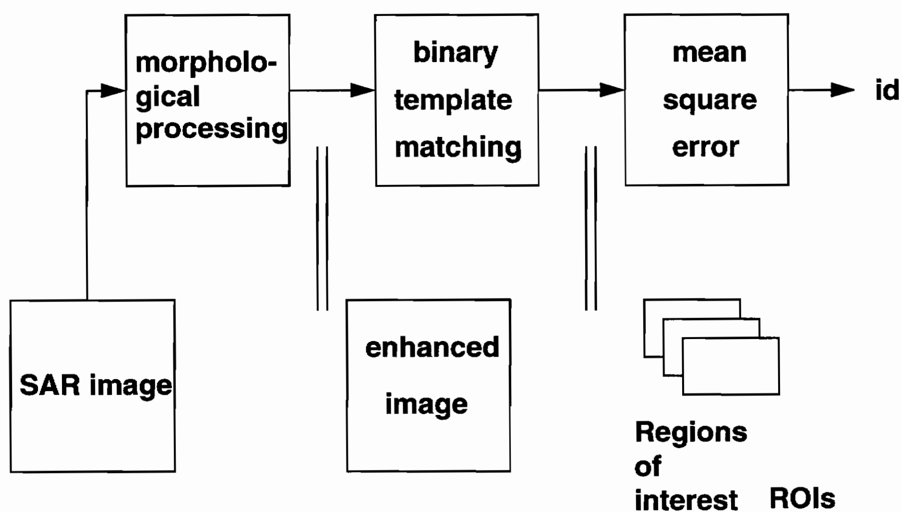


Figure 5.1. System overview[5]

1) Morphological Operations
2) Binary template matching
3) Mean square error (MSE) algorithm

5.2.1 Morphological operations

The two morphological operations are *erosion* and *dilation*. An Erosion filter is a minimum picking filter and tends to make the dark areas of an image larger and bright areas smaller. The maximum filter is called a dilation filter because it makes the bright

areas larger and dark areas smaller. These operations are very similar to the two dimensional convolution operation.

5.2.2 Binary template matching

The binary template matching stage consists of operations related to edge detection, thresholding and matching. The edge detection operation consists of smoothing filters, horizontal gradient, vertical gradient, and thresholding to obtain the final binary output image. The steps used in the edge detection process are:

1) Convolution with a 3 x 3 gaussian blur kernel
2) Edge finding with a 3 x 3 vertical sobel (convolution operation)
3) Edge finding with a 3 x 3 horizontal sobel (convolution operation)
4) Combination of the two sobel operator outputs using the largest absolute value
5) Detection of the combined matrix using a user specified threshold[2]

This edge detection operation is shown in figure 5.2.

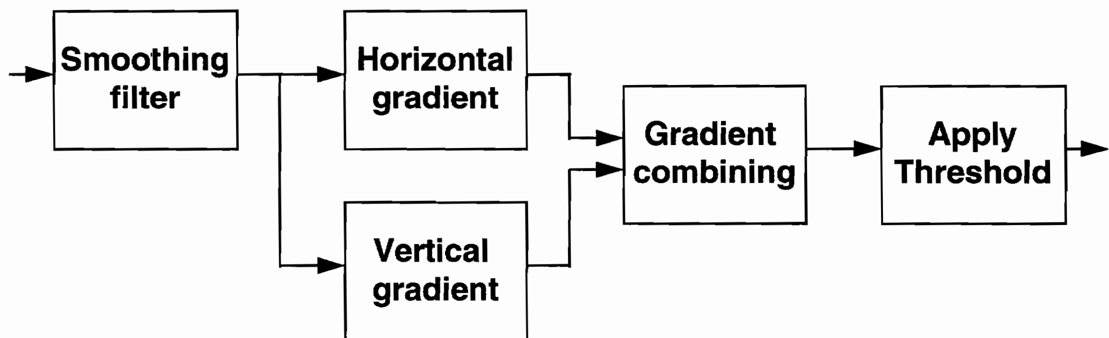


Figure 5.2. Edge detection[2]

5.2.3 Mean square error algorithm

The mean square algorithm which is explained in the MSE classifier by Sandia National Labs[5] is given below:

"1) It allocates memory for the input image (SAR output image)

2) Loads region of interest image R

3) Loads class C template images (M, S, B) where

M_i = ith mean template

S_i = ith standard deviation template

B_i = ith binary mask template

4) Calls a procedure which finds the *best hit* with inputs of the procedure being R, M, S and B. The output of this procedure are the discriminant surface D and its minimum d.

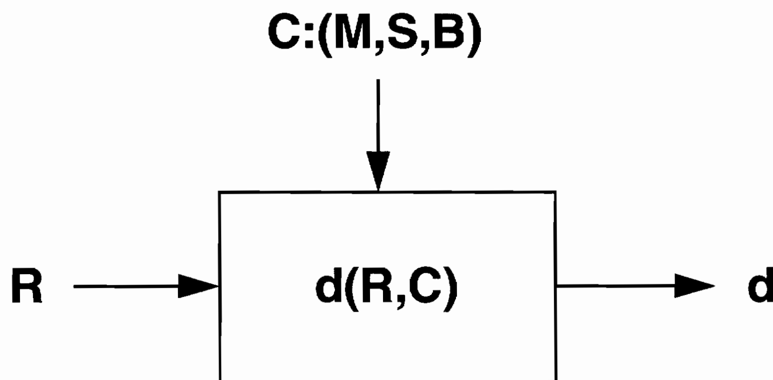


Figure 5.3. Diagram for best hit[5]

The intermediate steps within this procedure are as follows:

a) compute chip = low pass filter(R)

b) compute $chip^2$

c) compute Sc = clip(S)

d) compute V = Sc2

e) compute 1/V

f) compute 2DFFT(chip)

g) compute 2DFFT($chip^2$)

h) Call another procedure with inputs M, 1/V, 2DFFT(chip), 2DFFT($chip^2$) and the output of this procedure is discriminant surface D

i) Finds the minimum of all D's

j) Returns discriminant surface D and minimum d

5) Writes d and (m, n) at which it occurred "

The output of the MSE algorithm indicates the class with which there is a match between the input region of interest and the class by thresholding the distance measure as shown in the figure 5.4[5].

5.3 Primitives to support the Automatic target recognition algorithm

The primitives required to support the three stages of the ATR algorithm are:

1) Two dimensional convolution in the smoothing filter, morphological operations and the horizontal/vertical edge detection. The full source code for the two dimensional convolution, named CONV_SIG, is given in Appendix D.

2) Two dimensional fast fourier transforms in the mean square error algorithm to calculate the distance between the region of interest and each of the several classes of template images. The full source code for the two dimensional fast fourier transform, named FFT_SIG, is given in Appendix D.

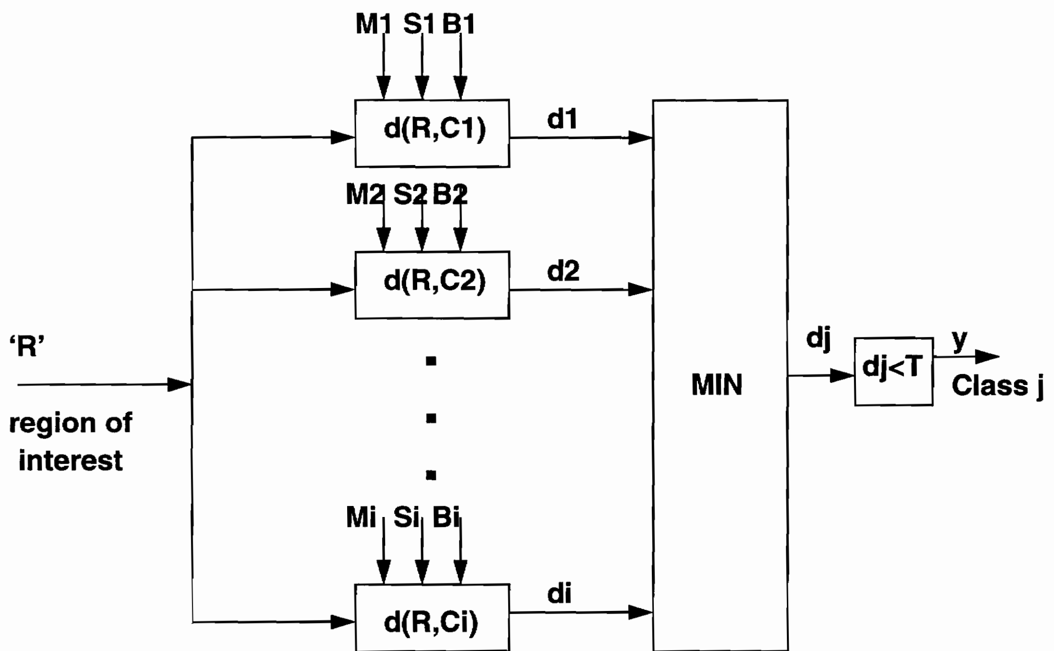


Figure 5.4. MSE algorithm[5]

Chapter 6. New implementation techniques

6.1 Introduction

Development of VHDL libraries to support various digital signal processing and image processing functions is a very complicated and time-consuming task. As the amount of complexity in the system increases, model development using ordinary hand coding techniques become very tedious. The model generator must be able produce code that models both control and data flow logic. Specific to the RASSP program, the code is related to signal processing systems, but the techniques should be applicable to a wide class of modeling problems. Hence the tools which are effective in model generation should be used. Tools such as Ilogix Express VHDL are effective for modeling control logic in statechart form, while Cadence SPW and UC Berkeley Ptolemy can be used to develop data flow models. The methodology in this thesis is not tied to any particular tool, and also can be applied to new tools as they come forward. VHDL code generation tools have several advantages:

- They reduce the time necessary to develop a VHDL model for a complex system.
- Most of the tools produce VHDL code which is synthesizable i.e., the code can be converted to a gate level layout and even further down the design cycle.
- Use of tools relieves the designer of actually writing VHDL code.
- Some tools such as Ptolemy provide the designer with a programming environment in with which the designer can add his own basic blocks to the actual tool palette.

There are numerous commercial tools available in the market which incorporate functions covering different aspects of signal processing and image processing. One such tool is Matlab which is a widely accepted tool that supports thousands of built-in functions in the fields of signal processing and image processing. As part of this thesis different ways of integrating Matlab code with VHDL have been studied using the simulation control language provided along with the Synopsys VHDL simulator. This relieves the designer of actually writing VHDL code for rarely used functions assuming that they are included in Matlab. This approach also brings the accuracy of numeric computations in Matlab to VHDL.

6.2 Overview of Matlab

Matlab is a technical computing environment for high performance numeric computation and visualization. Matlab integrates numerical analysis, matrix computation, signal processing and graphics where problems and solutions are expressed just the way they are written mathematically[9].

The signal processing toolbox is a collection of functions built on Matlab's powerful numeric computing environment. The toolbox functions support a wide range of signal processing operations from wave form generation to filter design and implementation, parametric modeling and spectral analysis.

The image processing toolbox provides the user with a wide range of image processing operations from color operations, image enhancements/analysis to morphological operations.

6.3 Integrating matlab code with VHDL

6.3.1 Purpose

This approach which integrates Matlab code and VHDL makes use of the simulation control language available with the Synopsys VHDL simulator. The two main purposes are:

1) This approach relieves the designer of writing VHDL code for functions which are rarely used. This integration also allows the designer to use the full power of Matlab which is known for its high performance numeric computations and the wide range of functions available within Matlab to support digital signal processing and image processing operations.

2) Verification of a VHDL model is a time consuming and laborious task. This approach is used to generate input data to test the VHDL primitives and also allows testing parts of

a VHDL model before the other parts are developed. With this approach, at the end of the VHDL simulation, the functioning of the VHDL models is verified by plotting the results of both Matlab and VHDL simulations on a single graph with both of them being driven by the same input data set. The two modes of operations describe how the VHDL models are verified along with the integration of Matlab code with VHDL.

6.3.2 Modes of operation

The two modes of operations which can be used to develop a digital signal processing system are illustrated with the help of an example.

Example

A filter design includes three steps

1) Calculate the filter coefficients using classical filter design procedures.

   e.g., Chebyshev, Butterworth, Elliptic

2) Find the outputs by applying some type of alteration on the input data.

$$H(z)=\frac{b_0+b_1 z^{-1}+b_2 z^{-2}+\ldots}{a_0+a_1 z^{-1}+a_2 z^{-2}+\ldots}=\frac{Y(z)}{X(z)}$$ transfer function of a filter

where b's and a's represent the filter coefficients.

3) Find the frequency response by fourier transforming the filter output.

The *first approach* to the integration of matlab code with VHDL invokes Matlab at the beginning and at the end of simulation. This mode of operation is illustrated in Figure 6.1. Step one of the filter design which is calculating the filter coefficients by any one of the classical filter design methods such as Butterworth, Elliptic and Chebyshev can be a laborious task for a modeler to accomplish in VHDL. Matlab has all these functions in its library. The designer can accomplish this task i.e.. calculation of filter coefficients by just writing a single line of matlab code. At the beginning of the simulation Matlab is invoked with the matlab file containing the filter design commands piped into matlab for the commands in that file to be executed.

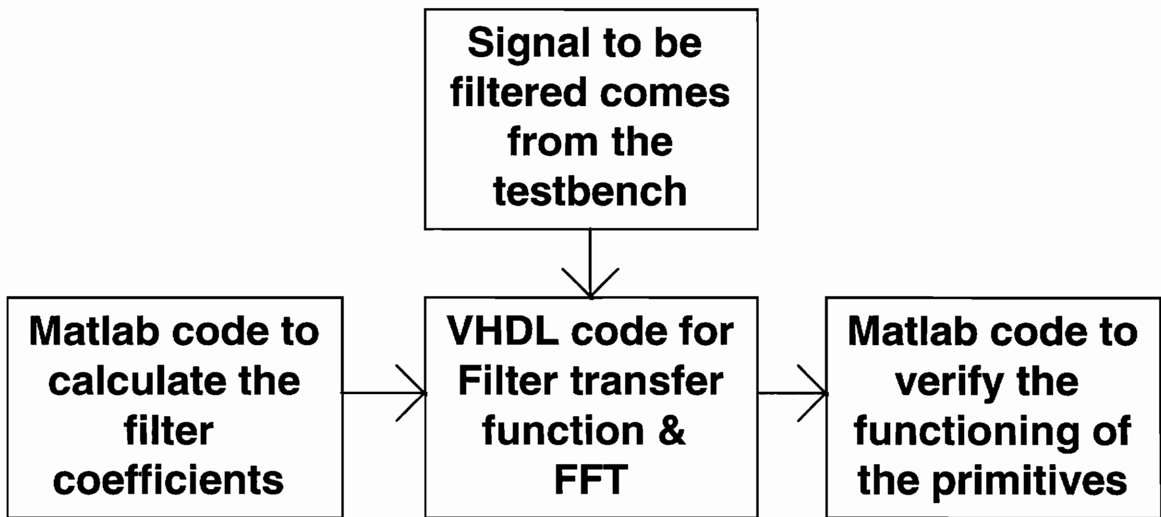


Figure 6.1. Mode one.

The Matlab engine executes the code written in that Matlab file and generates filter coefficients according to the given constraints and writes them to a file. These filter coefficients are read into VHDL using file I/O as soon as the Matlab simulation is over.

The filter coefficients are read into the filter primitive developed in VHDL as part of the signal processing primitives library. The input signal to be filtered comes from the VHDL test bench. As soon as the filtering operation is over the fourier transform operation is computed on the filter output data by matlab. This approach shows a method to develop a digital signal processing system using both the VHDL primitives and the built-in functions in Matlab.

Verification of the VHDL primitives is another laborious task. A matlab file which consists of Matlab code representing the DSP system modeled in VHDL is created and at the end of the VHDL simulation this file is piped into Matlab. Both models are driven with the same input data set. The final output results of both the Matlab simulation and the VHDL models can be plotted on the same graph using the extensive graphic capabilities of Matlab. This is a clear way of testing the primitives with the VHDL simulation results being compared with the results of the widely accepted Matlab's signal processing toolbox.

A typical simulation control file is shown below[12]:

```
! matlab < fil_coeff.m
run
! matlab < comp_res.m
quit
```

where the file fil_coeff.m consists of matlab code to calculate the filter coefficients and write them to an intermediate file. The file comp_res.m consists of matlab code which

reads the VHDL simulation results from another intermediate file to compare them with the results obtained from executing matlab code consisting of the same operations on the same input data set. Finally matlab plots out the results of both the VHDL and matlab simulation on a single graph.

The *second approach* to the integration of Matlab code with VHDL is illustrated in Figure 6.2.

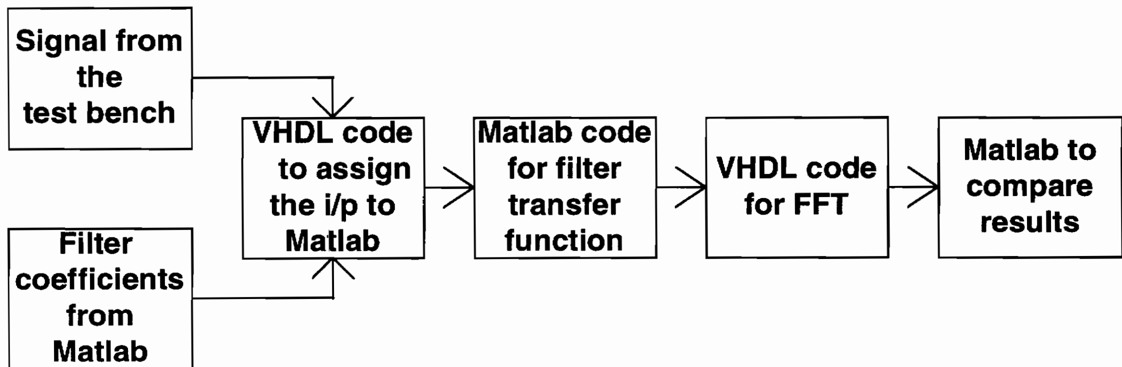


Figure 6.2. Mode two

The first approach assumes that the fourier transform and filter are already available in the digital signal processing primitives library. The second approach deals with the situation when the intermediate function to be performed is absent in the VHDL primitives library. This approach interrupts the VHDL simulation as needed. Considering the same example just described: in this case let's assume that the filter

primitive is not available in the signal processing VHDL primitives library. This approach accomplishes the task in the following way:

1) Filter coefficients are generated using Matlab filter design functions
2) Input signal to be filtered comes from the VHDL test bench
3) This input signal is assigned as input to Matlab filtering operation
4) Again the fourier transform primitive from the signal processing library is used to calculate the frequency response i.e.. input to this primitive is the output data generated by the matlab filter function and the output is the frequency response of the filter
5) Finally verification of the VHDL primitives can be done by the performing the same set of operations both in Matlab and VHDL on the same set of input data and plotting out the output results of both Matlab and VHDL simulation using Matlab

A typical simulation control file for mode two is shown below:

```
! matlab < fil_coeff.m
monitor -s on 'vhdl_file_name' line_no
run
! matlab < filt_op.m
run
! matlab < comp_res.m
quit
```

where line_no is the line number in the VHDL file at which the VHDL simulation should stop. The file fil_coeff calculates the filter coefficients. The file filt_op.m computes the output for the filtering operation in matlab with input coming from the VHDL testbench.

The file comp_res.m consists of matlab code which reads the VHDL simulation results from another intermediate file and executes the same operations on the same input data set using matlab functions. Finally matlab plots out the results of both the VHDL and matlab simulations on a single graph.

6.3.3 Advantages/Disadvantages, Comparator

The advantages with the two approaches are: 1) They reduce lot of VHDL coding. 2) They show a quick and accurate way of testing the VHDL primitives.

The disadvantages with the two approaches are: 1) They use extensive file I/O because the interaction between the VHDL simulator and matlab engine is through files. This extensive usage of file I/O may result in increased execution times. 2) The inputs to the models are generated randomly using matlab. Although this random test generation is an efficient way to test models it may not have 100% coverage i.e.. the models are not exhaustively tested for all cases. Synopsys VHDL analyzer and VHDL simulator have been extensively used for the RASSP project at Virginia Tech.

The task of the comparator is to compare the output of the model against the expected response and creates error reports if responses from the model are outside the acceptable range, either in terms of values or in terms of timing. If this comparator is constructed for back-to-back testing, both the "gold" version and the model under test (MUT) are provided with the same stimuli, and the comparator ensures that the difference in the outputs of the "gold" version and the MUT is in a particular range. For example: for the RASSP benchmark SAR, "acceptably close" means that the "gold" synthetic aperture

radar image and the synthetic aperture radar image produced by the model under test (MUT) must have an RMS difference within 3dB. The comparator used for the SAR VHDL model calculates the precision obtained. In the two approaches described above the "gold" models are the matlab functions and the models under test are the primitives from the RASSP VHDL libraries.

6.3.4 Results

Figure 6.3 depicts results from a filter design example built with the VHDL primitives using both the approaches described above. Results are the same for both the approaches.

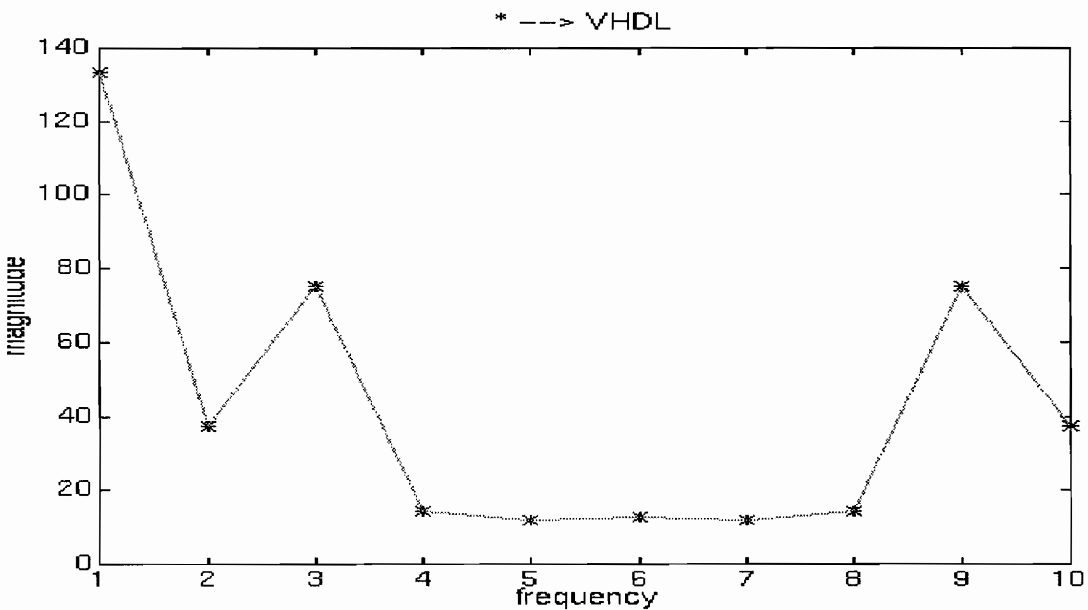


Figure 6.3. Response from the VHDL primitives

6.4 Introduction to tools

"The RASSP digital signal processing environment requires floating point, fixed point (integer), and bit vector models. Therefore tools should be able to support models with floating point, fixed point and bit vector data types.

Unless VHDL models of DSP circuits can simulate in a reasonable amount of time , the usefulness of VHDL in the DSP paradigm is limited. Thus the code developed from any tool should simulate as efficiently as possible. A major factor affecting simulation efficiency is the number of signals in a model, because each signal requires time queue processing. Thus the use of complex structural models should be avoided because they tend to have large signal interconnects. Similarly large behavioral models with many processes, will have a relatively large number of signals accompanied by the associated queue processing overhead. One possible way to eliminate this problem is to code single process models which rely on variables to communicate values and use procedure calls to represent DSP primitives. Therefore tools should be able to support models with a minimum number of signals"[7].

6.4.1 Ptolemy

"Ptolemy is a very flexible foundation for building simulation environments, where the key objective is the ability to combine simulation environments into multi-paradigm simulations as necessary. Large systems often mix hardware, software and communication systems. They may also combine hardware targets, including custom logic, processors with varying degrees of programmability, systolic arrays, and other multiprocessor subsystems. Tools supporting each of those components are different,

using for instance data flow principles, regular iterative algorithms, communicating sequential processes, control/dataflow hybrids, functional languages, and discrete system theory and simulation. Ptolemy is a third generation software environment that supports heterogeneous system specification, simulation, and design. It is an object oriented framework within which diverse models of computation can coexist and interact. It uses hierarchy to mix heterogeneous models of computation"[10].

"Each model of computation in Ptolemy is called a Domain. It currently supports synchronous data-flow (SD) domain, a dynamic data flow domain (DD), a discrete-event (DE) domain, a Thor domain (for circuit simulation) and various code generation domains. In ptolemy the overall simulation is decomposed into software modules called *blocks.* There are two types of blocks: *Star* and *Galaxy*. The *Star* is elemental, in the sense that it is implemented by user provided code. There are many pre-coded blocks in ptolemy but adding a new block is easy hence the system can be viewed as a programming environment, and not just as a monolithic tool to be used unmodified. A *Galaxy* is a block which internally contains *Star*s as well as possibly *Galaxi*es. A *Universe* is a complete program, or application"[10].

Example:

The schematic of an FIR filter with four coefficients in Ptolemy is represented in Figure 6.4. It consists of through, register, adder and gain stars. The registers are used to incorporate the unit delays in the filter transfer function. The gain stars are used to multiply the input signal with the filter coefficients. The through star is used to copy one input to multiple outputs. The structural VHDL code generated by ptolemy uses

component instantiations for registers, adders and gain elements. The basic behavioral models for the registers, adders and gain elements are modeled by the user.

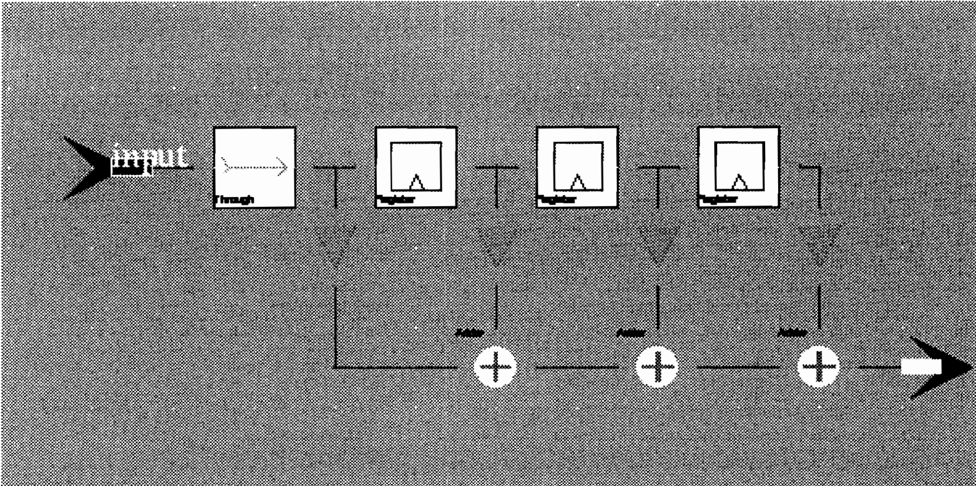


Figure 6.4. Ptolemy schematic[10]

6.4.2 Purpose/current status of Ptolemy

Any tool which is being used to generate VHDL code should be able to support floating point arithmetic, fixed point(integer) and bit vector models in a DSP environment. Comdisco SPW has a Hardware Design System (HDS)[11] which enables the designer to quickly design, optimize and implement DSP and other systems in ASICs and FPGAs. With HDS the designer can apply SPW's object oriented design and modeling approach to fixed point system development and digital hardware design, then automatically generate VHDL code targeted to any synthesis tool. The problem with this tool with respect to the

RASSP environment is that it doesn't generate VHDL supporting floating point arithmetic models. It only generates VHDL code which can be targeted to any synthesis tool and hence can't generate VHDL code supporting real data types.

Ptolemy provides the user with a programming environment in which the user can build his own stars and galaxies supporting required data types. This feature of Ptolemy is used to overcome the problems that were faced in using Comdisco SPW as a tool for generating VHDL code supporting floating point arithmetic.

Ptolemy has a number of simulation domains as well as code generation domains. It has two VHDL domains which are: *VHDLF* domain to support functional models which can be used in the DSP type of environment and *VHDLB* domain to support behavioral models. The next version of ptolemy is expected to have a new VHDL domain which provides the user with the capability of generating behavioral (procedural based) VHDL without any internal signals.

6.4.3 Generation of VHDL code supporting floating point arithmetic models

Ptolemy provides rich libraries of stars for each domain. The ptolemy preprocessor language can be used to develop new stars because star libraries can't possibly be complete. The newly designed stars can be dynamically linked into Ptolemy when required. A typical star which has multiple input ports and one output port is represented in Ptolemy as in Figure 6.5.

This is a star which has multiple input ports (represented by numerous arrows on the input) of type real and one output port (represented by a single arrow) of type real.

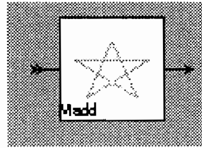


Figure 6.5. Star in Ptolemy.

This star adds all the values available at the input ports and provides the resulting value at the output port.

A file that creates this type of star written in Ptolemy preprocessor language follows:

```
defstar {
    name {Madd}
    domain {VHDLF}
  desc { Output the sum of the inputs, as a floating value. }
    author { }

    location { VHDLF main library }
    inmulti {
        name { input }
        type { float }
        }
    output {
        name { output }
        type { float }
        }
    go {
        MPHIter nexti(input);
        PortHole *p;
        double sum = 0.0;
        while ((p = nexti++) != 0)
```

```
        sum += double((*p)%0);
    output%0 << sum;
        }
}
```

Figure 6.6 shows an example representing complex multiplication with the real and imaginary parts of the two numbers being reals. This is based on the complex multiplication example which is built into Ptolemy where the real and imaginary parts of the numbers are integers. In figure 6.7 the block named "test1" represents the complex multiplication. The input blocks which are connected to the four input ports of "test1" represent the inputs to this particular function. The first instance of Mramp represents the real part of the first number and the first instance of Mdc represents the imaginary part of the first number. The second instance of Mramp represents the real part of the second number and the second instance of Mdc represents the imaginary part of the second number. The two blocks which are connected to the two output ports of "test1" represent sinks. The first sink represents the real part of the output and the second sink represents the imaginary part of the output. As illustrated in Figure 6.7 "test1" can be divided into a connection of basic blocks which are: Mfork which copies the input to multiple outputs, ReAdder which adds multiple inputs of type real and produces one output of type real, Remult which multiplies multiple inputs of type real and produces one output of type real, and Mgain where the output is -1 times the input. VHDL code is generated using Ptolemy for this Galaxy in which the stars are built by the user supporting real data type.

At the time of this research Ptolemy is capable of generating structural VHDL code using component instantiations of behavioral models developed by the user. Here the VHDL linked to the basic block is written by the user himself.

Although structural models are inefficient because of their usage of signals for interconnections between blocks which require queue processing Ptolemy provides the user a programming environment rather than a piece of unmodifiable software.

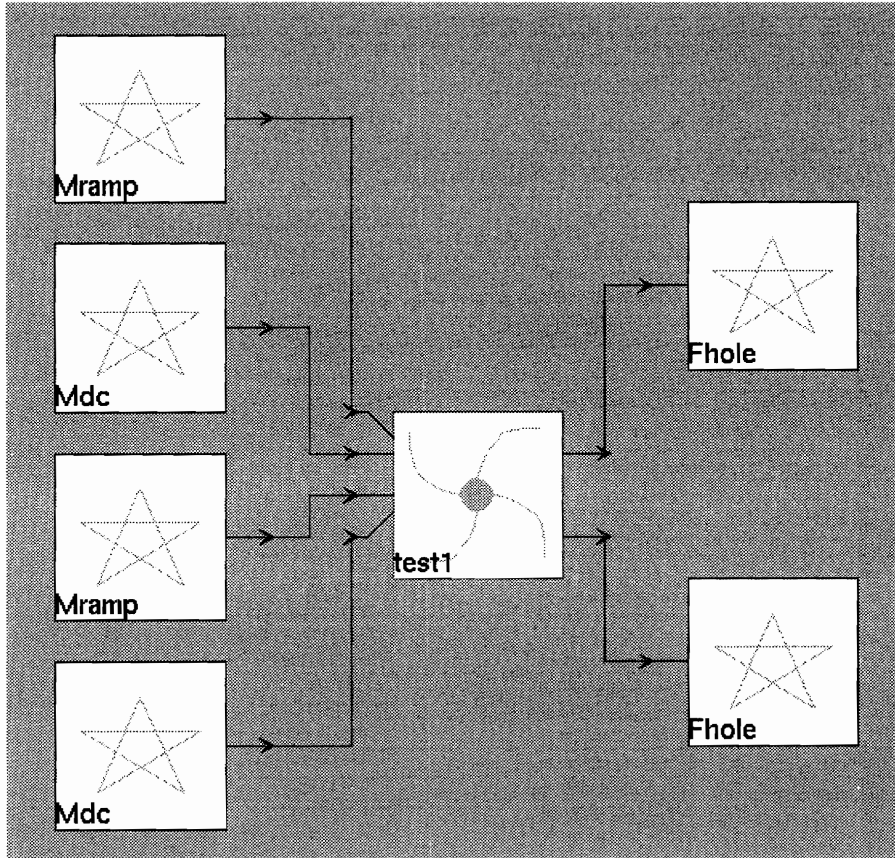


Figure 6.6. Complex multiplication galaxy

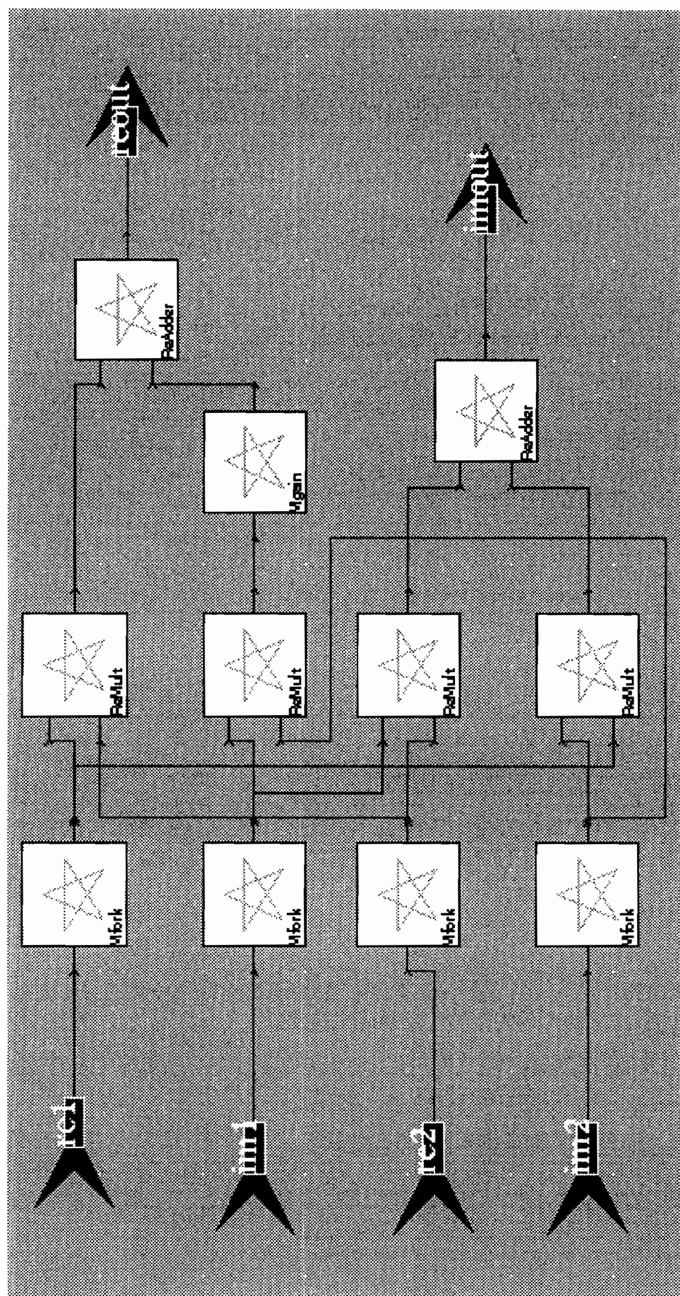


Figure 6.7. "test1" galaxy

6.4.4 Ptolemy results

Figure 6.8 shows the results obtained from simulating the structural model generated by ptolemy using synopsys VHDL simulator & analyzer. The VHDL code is given in Appendix B.

	0		10		20
/TEST1/Mramp1_output	1.0	2.0	3.0	4.0	5.0
/TEST1/Mdc1_output	0.0	1.0	2.0	3.0	4.0
/TEST1/Mramp2_output	1.0	2.0	3.0	4.0	5.0
/TEST1/Mdc2_output	0.0	1.0	2.0	3.0	4.0
/TEST1/ReAdder2_output	1.0	3.0	5.0	7.0	9.0
/TEST1/ReAdder1_output	0.0	4.0	12.0	24.0	40.0

Figure 6.8 Simulation results

The equations which represent the complex multiplication are;

(ReAdder2_output + j* ReAdder1_output) = (Mramp1_output + j* Mdc1_output) * (Mramp2_output + j* Mdc2_output)

which reduce to:

ReAdder2_output = Mramp1_output*Mramp2_output - Mdc1_output*Mdc2_output;

ReAdder1_output = Mramp1_output*Mdc2_output + Mdc1_output*Mramp2_output;

where Mramp1_output, Mdc1_output, Mramp2_output, Mdc2_output are reals.

6.5 Techniques for writing VHDL code

The techniques used for writing VHDL models to populate the VHDL primitive libraries are:

1) Overloading operators
2) Overloading subprograms
3) Data types to decrease the execution time

6.5.1 Overloading operators

Overloading allows the designer to write much more readable code. An object is overloaded when the same object name exists for multiple type values. The VHDL compiler will select the appropriate object to use in each instance. Overloading an operator to perform the same operation on multiple types results in models that are easier to read and maintain.

The + operator works only with integers, real and physical types. The other frequently used operators are *, -. The libraries consist of various other data types such as a COMPLX_TYP in the digital signal processing primitives library which is used to represent a complex number and is a record consisting of a real and a imaginary part. An example is shown below:

```
type COMPLX_TYP is record
        RE: REAL;
        IMAG: REAL;
end record;
```

In the same way the image processing primitives library consists of data types such as TOD1_COMPLX_TYP which is two dimensional array of COMPLX_TYP. An example is shown below:

```
type TOD1_COMPLX_TYP is array( POSITIVE range <>, POSITIVE range <>) of
COMPLX_TYP;
```

The DSP primitives such as FFT require additions and multiplications of two complex numbers. Hence functions that overload the operators to accomplish the operations on various data types are coded in VHDL and are built as part of the libraries. An example is shown below:

```
function "+"(X: in COMPLX_TYP; Y: in COMPLX_TYP) return COMPLX_TYP is
variable TEMP: COMPLX_TYP;
begin
  TEMP.RE := X.RE+Y.RE;
  TEMP.IMAG := X.IMAG+Y.IMAG;
  return TEMP;
end "+";
```

The image processing primitives have additions, multiplications, subtractions of two images which are two dimensional arrays of numbers. Functions that overload these operators to accomplish the operations on images have been coded in VHDL and are part of the libraries. An example is shown below:

```
function "+"(X: in TOD1_CMPLX_TYP; Y: in TOD1_CMPLX_TYP) return
TOD1_CMPLX_TYP is
variable TEMP: TOD1_CMPLX_TYP;
begin
  for I in 1 to X'HIGH(1) loop
    for J in 1 to X'HIGH(2) loop
      TEMP(I, J) := X(I, J)+Y(I,J);
    end loop;
  end loop;
  return TEMP;
end "+";
```

6.5.2 Overloading subprograms

Subprogram overloading allows the designer to write multiple subprograms with the same name, but the number of arguments, the type of arguments, and return value can be different. The VHDL compiler, at compile time, will select the subprogram that matches the subprogram call. If no subprogram matches the call, an error is generated. A subprogram in a library say an FFT can be called with different input/output data types. For example the fourier transform can be used on an array of complex numbers or an

array of reals. Hence all the procedure names are overloaded within the libraries. The operation to be performed can be called with the same procedure name regardless of the difference in the input/output data types or the number of input arguments or the number of dimensions in the input array. An example is shown below:

type ARR1_RE_TYP is array(POSITIVE range <>) of REAL;
type ARR1_CMPLX_TYP is array(POSITIVE range <>) of COMPLX_TYP;
procedure FFT(DIN: in ARR1_RE_TYP; ISIGN: in INTEGER; DOUT: out ARR1_CMPLX_TYP);
procedure FFT(DIN: in ARR1_CMPLX_TYP; ISIGN: in INTEGER; DOUT: out ARR1_CMPLX_TYP);

All subprograms especially the procedures can have the input/output parameters as variables or signals. However the synopsys simulator doesn't allow the same procedure name for different cases when the inputs/outputs are declared as variables in one case and as signals in the second case. Hence for every primitive two procedure names are used: one to deal with input/output ports being variables and the second to deal with input/output ports being signals. For example, the following overloading is not allowed,

procedure FFT(variable DIN: in ARR1_CMPLX_TYP; variable ISIGN: in INTEGER; variable DOUT: out ARR1_CMPLX_TYP);
procedure FFT(DIN: in ARR1_CMPLX_TYP; ISIGN: in INTEGER; DOUT: out ARR1_CMPLX_TYP);

Hence the above example is represented as:

procedure FFT_VAR(variable DIN: in ARR1_CMPLX_TYP; variable ISIGN: in INTEGER; variable DOUT: out ARR1_CMPLX_TYP);
procedure FFT_SIG(DIN: in ARR1_CMPLX_TYP; ISIGN: in INTEGER; DOUT: out ARR1_CMPLX_TYP);

Both of these procedure names are overloaded to support different cases such as: different number of arguments, different data types etc. The result is that the models are easier to read and maintain.

6.5.3 Data types

The data types used to develop the libraries have been verified for maximum efficiency. A record is used to represent a complex number rather than an array of two numbers representing the real and imaginary parts of the complex number.

```
type COMPLX_TYP is record
        RE: REAL;
        IMAG: REAL;
end record;
```

is more efficient than

```
type COMPLX_TYP is array(1 to 2) of REAL;
```

The execution times for models with different data types to represent a complex number are given below using the UNIX command "time" in two different cases:

Number of inputs for a fourier transform: 21

Data type	t1	t2	t3
Array	8.8	4.8	0.8
Record	6.4	4.9	0.7

Number of inputs for a fourier transform: 168

Data type	t1	t2	t3
Array	142.4	132.2	0.6
Record	135.6	129.9	0.6

where t1 represents the elapsed time during the command

t2 represents the time spent in the system

t3 represents the time spent in the execution of the command.

It is verified that the record data type is a better way of representing a complex number than representing it as an array of two numbers.

The access data type which is very rarely used in VHDL modeling is shown to be an effective way to decrease the execution times. The execution times of the same model with and without using access types are given below:

Number of inputs to the fourier transform: 10

Data type	t1	t2	t3
without "access"	4.609	0.72	0.75
with "access"	1.5	0.65	0.44

where t1 represents the elapsed time during the command

t2 represents the time spent in the system

t3 represents the time spent in the execution of the command.

Because any access data type should be a variable, the overloaded operators no longer are used in the fourier transform model built with access types. Instead for "+" a procedure is given below:

```
type COMPLX_ACCESS is access COMPLX_TYP;
procedure add(variable X: in COMPLX_TYP; variable Y: in COMPLX_ACCESS;
variable Z: inout COMPLX_ACCESS) is
begin
   Z := new COMPLX_TYP'(X.RE+Y.RE, X.IMAG+Y.IMAG);
end add;
```

The disadvantage in using this data type is that the VHDL code cannot be fed into any synthesis tool to convert the code into a gate level layout.

Chapter 7. Conclusions and Future Work

This thesis has discussed a methodology for VHDL model development using tools such as Matlab, Ptolemy and Comdisco SPW along with developing primitives to support the two RASSP specific applications: 1) Synthetic aperture radar image processor (SAR) 2) Automatic target recognition (ATR) algorithm. It describes problems which are relevant to the RASSP VHDL libraries that can be faced in using these tools to generate VHDL code along with a solution for this problem. It explains how a SAR image processor has been implemented with DSP primitives from the libraries.

Some of the avenues in which future work is possible are as follows. Translators from Matlab language or 'C' to VHDL should be developed to automate the process of populating the libraries. The new VHDL domain in the next release of ptolemy should be used extensively in the population of these libraries. Different stars in ptolemy should be developed using the ptolemy language so that they support a wide range of DSP/IP applications.

Bibliography

[1] Apostolo Dollas and Nick Kanopoulos, "Reducing time to market through Rapid Prototyping", *IEEE Computer, February, 1995,* Page 14

[2] Paul M. Embree and Bruce Kimble, *C Language Algorithms For Digital Signal Processing*, Prentice Hall, 1991

[3] William D. Stanley, Gary R. Dougherty and Ray Dougherty, *Digital Signal Processing*, Reston publishing company, 1984

[4] G.A.Shaw, *Synthetic Aperture Radar Image Processor - RASSP Benchmark* , M.I.T. Lincoln Laboratory, Lincoln Labs, MA, January 1994.

[5] Ron Dilsavor, *Sandia National Lab's MSE Classifier*, Sverdrup Technology Inc.,

[6] James R. Armstrong and F.G. Gray, *Structured Logic Design with VHDL*, Prentice Hall, 1993.

[7] Hrishikesh Srinivasan, *Behavioral Testbench Development for DSP Models*, Master of Science thesis, Electrical Engineering Department, Virginia Tech, February 1995.

[8] Xu Zhen, *Modeling Signals and Sensors Using VHDL For SAR System*,

Master of Science thesis, Electrical Engineering Department, Virginia Tech, February 1995.

[9] *MATLAB Reference Guide*, August 1992.

[10] *Ptolemy Manual, Version 0.3.1*, University of California at Berkeley.

[11] Comdisco Systems Inc., *SPW - The DSP Framework, Hardware Design System*, March 1994.

[12] *Synopsys VHDL System Simulator - Command Reference Manual, Version 3.0*, December 1992.

Appendix A: VHDL code for the primitives

```
-- Title: PACKAGE DSP_PRIMS
--
-- Purpose: VHDL declarations for package dsp primitives which contains common
--        real constants, common trigonometric functions, primitve  dsp
--        procedures, and common complex arithmetic functions
--
-- References:1.. PACKAGE MATH_REAL by IEEE VHDL Math Package Study Group
--          2.. SIGNAL PROCESSING ALGORITHMS IN FORTRAN AND C by ----
--              STEARNS,DAVID
--          3.. DISCRETE TIME SIGNAL PROCESSING by OPPENHEIM & SCHAFER
--          4.. Guidelines for VHDL models in a team environment
--                                              by janick bergeron

--------------------------------------------------------------------------
package DSP_PRIMS_PKG is
   --
   type ARR1_RE_TYP is array(POSITIVE range <>) of REAL;
   type ARR0_RE_TYP is array(NATURAL range <>) of REAL;
   type COMPLX_TYP is record
                RE: REAL;
                IMAG: REAL;
   end record;
   type ARR1_CMPLX_TYP is array(POSITIVE range) of COMPLX_TYP;
   type TOD1_CMPLX_TYP is array(POSITIVE range <>, POSITIVE range <>) of
   COMPLX_TYP;

   procedure FFT_SIG(
      signal  DATA : in ARR1_CMPLX_TYP;
      signal  ISI : in INTEGER;
```

```
    signal  FFT_OUT : out ARR1_CMPLX_TYP );

  procedure FILT_SIG(
    signal   DATA : in ARR1_RE_TYP;
    signal   A : in ARR1_RE_TYP;
    signal   B : in ARR0_RE_TYP;
    signal   FILT_OUT : out ARR1_RE_TYP );

  procedure CONV_SIG(
    variable A: inout TOD1_RE_TYP;
    variable B: inout TOD1_RE_TYP;
    variable CONV_OUT: inout TOD1_RE_TYP);

  procedure FFT_VAR(
    variable DATA: inout TOD1_CMPLX_TYP;
    variable ISIGN: in INTEGER;
    variable FFT_OUT: out TOD1_CMPLX_TYP );

  procedure CONV_VAR(
    variable A: inout TOD1_RE_TYP;
    variable B: inout TOD1_RE_TYP;
    variable CONV_OUT: inout TOD1_RE_TYP);

procedure CONV_VAR(
    variable A: inout ARR1_RE_TYP;
    variable B: inout ARR1_RE_TYP;
    variable CONV_OUT: out ARR1_RE_TYP);

end DSP_PRIMS_PKG;

package body DSP_PRIMS_PKG is
  --
  --commonly used constants
  --
  constant  MATH_PI :  real := 3.14159_26535_89793_23846; --value of pi
  constant  HALF_PI :   real := MATH_PI/2.0;

  function "+" ( X: COMPLX_TYP ;
             Y: COMPLX_TYP
           ) return COMPLX_TYP is
    -- if X=(a,b)  & Y=(c,d)  it returns ((a+c),(b+d))
  begin
       return ( X.RE+Y.RE, X.IMAG+Y.IMAG ) ;
```

```
end "+" ;

function "*" ( X : COMPLX_TYP ;
            Y: COMPLX_TYP
          ) return COMPLX_TYP is
  --if X=(a,b) & Y=(c,d)  it returns ((a*c-b*d),(b*c+a*d))
begin
      return ( X.RE*Y.RE-X.IMAG*Y.IMAG , X.IMAG*Y.RE+X.RE*Y.IMAG ) ;
end "*" ;

procedure FFT_SIG(
    signal  DATA : in ARR1_CMPLX_TYP;
    signal  ISI : in INTEGER ;
    signal  FFT_OUT : out ARR1_CMPLX_TYP) is
     --returns the fast fourier transform DATA in (real,imaginary)

    variable W : COMPLX_TYP  ;
    variable TEMP : COMPLX_TYP ;
    variable  DATA1: ARR1_CMPLX_TYP (1 to DATA'HIGH ) ;
    variable MMAX : INTEGER :=1 ;
    variable ISTEP : INTEGER ;
    variable  X : INTEGER ;
    variable J : INTEGER :=1 ;
    variable M : INTEGER ;
    variable THETA : REAL ;
  begin
     DATA1:=DATA;
        FFT1 :    for I in DATA1'range loop
           if (I < J) then
              TEMP := DATA1(J);
              DATA1(J) := DATA1(I);
             DATA1(I) := TEMP;
          end if;
          M := DATA1'HIGH / 2;
         FFT2 : while ( J > M ) loop
            J:=J-M;
                 M := ((M + 1)/2);
            end loop FFT2 ;
           J := J+M;
        end loop FFT1;
        FFT3: while ( MMAX < DATA1'HIGH ) loop
           ISTEP :=2*MMAX;
           FFT4 : for M in 1 to MMAX loop
```

```
            THETA :=MATH_PI*REAL(ISI*(M-1))/REAL( MMAX );
              W := (COS(THETA),SIN(THETA));
              X := M;
             FFT5 : while (X <= DATA'HIGH)  loop
                 J := X+MMAX;
                 TEMP := W*DATA1(J);
                 DATA1(J) := DATA1(X)-TEMP;
                 DATA1(X) := DATA1(X)+TEMP;
                 X := X+ISTEP;
             end loop FFT5;
          end loop FFT4 ;
          MMAX := ISTEP;
       end loop FFT3 ;
       if ( ISI >= 0 ) then
           FFT6: for I in DATA1'range loop
              DATA1(I) := DATA1(I)/DATA1'HIGH;
          end loop FFT6;
       end if;
      FFT_OUT <= DATA1;
 end FFT ;

procedure FILT_SIG(
   signal  DATA : in ARR1_RE_TYP ;
   signal  A : in ARR1_RE_TYP ;
   signal  B : in ARR0_RE_TYP ;
   signal  FILT_OUT : out ARR1_RE_TYP ) is
   -- returns filtered data according to A's and B's

   variable INTY : ARR1_RE_TYP(1 to A'HIGH) ;
   variable INTX : ARR0_RE_TYP(0 to B'HIGH) ;
   variable SUM : REAL := 0.0 ;
   variable Y1 : ARR1_RE_TYP(1 to DATA'HIGH) ;
   variable L : INTEGER := 0 ;
begin
   INITB : for I in B'range loop
      INTX(I) := 0.0 ;
      end loop INITB ;
      INITA : for I in A'range loop
         INTY(I) := 0.0 ;
      end loop INITA ;
      FILT1 : for I in DATA'range loop
         INTX(0) := DATA(I) ;
         SUM := 0.0 ;
```

```
        FILT2 : for L in B'range loop
          SUM := SUM+B(L)*INTX(L) ;
        end loop FILT2 ;
        FILT3 : for L in A'range loop
            SUM := SUM-A(L)*INTY(L) ;
        end loop FILT3 ;
        L := B'HIGH ;
        FILT4 : while (L>=1) loop
          INTX(L) := INTX(L-1) ;
            L := L-1 ;
        end loop FILT4 ;
        L := A'HIGH ;
        FILT5 : while (L>=2) loop
          INTY(L) := INTY(L-1) ;
            L := L-1 ;
        end loop FILT5 ;
        Y1(I) := SUM ;
        INTY(1) := SUM ;
      end loop FILT1 ;
      FILT_OUT<=Y1;
  end FILT;

procedure CONV_VAR(
    variable A: inout ARR1_RE_TYP;
    variable B: inout ARR1_RE_TYP;
    variable CONV_OUT: out ARR1_RE_TYP ) is
    -- variable version of convolution

  variable I: INTEGER;
  variable A_FILT: ARR1_RE_TYP(1 to 2) := (0.0,0.0);
  variable TEMP: ARR1_RE_TYP(1 to A'HIGH+B'HIGH-1);
  variable TEMP1: ARR1_RE_TYP(1 to A'HIGH+B'HIGH-1);
  variable TEMP2: ARR0_RE_TYP(0 to A'HIGH-1);
  variable TEMP3: ARR0_RE_TYP(0 to B'HIGH-1);
  begin
    if (A'HIGH > B'HIGH) then
      if (B'HIGH > 1) then
        for I in 1 to A'HIGH+B'HIGH-1 loop
          if ( I <= A'HIGH) then
            TEMP(I) := A(I);
          else
            TEMP(I) := 0.0;
          end if;
```

```
          end loop;
          for I in B'range loop
             TEMP3(I-1) := B(I);
          end loop;
        end if;
        FILT_VAR(TEMP, A_FILT, TEMP3, CONV_OUT);
      else
        if (A'HIGH > 1) then
          for I in 1 to A'HIGH+B'HIGH-1 loop
             if (I <= B'HIGH) then
                TEMP1(I) := B(I);
             else
                TEMP1(I) := 0.0;
             end if;
                end loop;
        end if;
        for I in A'range loop
          TEMP2(I-1) := A(I);
        end loop;
        FILT_VAR(TEMP1, A_FILT, TEMP2, CONV_OUT);
      end if;
   end CONV_VAR;

procedure FFT_VAR( variable DATA: inout TOD1_CMPLX_TYP;
                   variable ISIGN: in INTEGER;
                   variable FFT_OUT: out TOD1_CMPLX_TYP ) is
   variable J: INTEGER;
   variable I: INTEGER;
   variable TEMP: ARR1_CMPLX_TYP(1 to DATA'HIGH(1));
   variable TEMP2: ARR1_CMPLX_TYP(1 to DATA'HIGH(2));
   variable TEMP3: ARR1_CMPLX_TYP(1 to DATA'HIGH(2));
   variable ISI: INTEGER := -1;
   variable DATA1: TOD1_CMPLX_TYP(1 to DATA'HIGH(2),1 to DATA'HIGH(1));
   variable TEMP1: ARR1_CMPLX_TYP(1 to DATA'HIGH(1));
   variable DATA2: TOD1_CMPLX_TYP(1 to DATA'HIGH(1),1 to DATA'HIGH(2));
   begin
      if (ISIGN=1) then
        for I in 1 to DATA'HIGH(1) loop
          for J in 1 to DATA'HIGH(2) loop
             DATA(I,J) := CONJ(DATA(I,J));
          end loop;
        end loop;
      end if;
```

```
for J in 1 to DATA'HIGH(2) loop
     for I in 1 to DATA'HIGH(1) loop
        TEMP(I) := DATA(I,J);
     end loop;
   FFT_VAR(TEMP, ISI, TEMP1);
   for I in 1 to DATA'HIGH(1) loop
      DATA1(J,I) := TEMP1(I);
   end loop;
 end loop;
for J in 1 to DATA'HIGH(1) loop
    for I in 1 to DATA'HIGH(2) loop
       TEMP2(I) := DATA1(I,J);
   end loop;
   FFT_VAR(TEMP2, ISI, TEMP3);
   for I in 1 to DATA'HIGH(2) loop
      DATA2(J,I) := TEMP3(I);
   end loop;
 end loop;
if (ISIGN=1) then
   for I in 1 to DATA'HIGH(1) loop
      for J in 1 to DATA'HIGH(2) loop
         DATA2(I,J) := CONJ(DATA2(I,J))/(DATA'HIGH(1)*DATA'HIGH(2));
      end loop;
   end loop;
end if;
FFT_OUT := DATA2;
end FFT_VAR;

procedure CONV_VAR(
     variable A: inout TOD1_RE_TYP;
     variable B: inout TOD1_RE_TYP;
     variable CONV_OUT: inout TOD1_RE_TYP) is
     variable I, J: INTEGER := 1;
     variable K, L: INTEGER := 0;
     variable TEMP: TOD0_RE_TYP(0 to A'HIGH(1)-1, 0 to A'HIGH(2)-1);
     begin
        for I in 1 to (A'HIGH(1)+B'HIGH(1)-1) loop
           for J in 1 to (B'HIGH(2)+A'HIGH(2)-1) loop
              CONV_OUT(I,J) := 0.0;
           end loop;
        end loop;
        for J in 1 to B'HIGH(2) loop
           for I in 1 to B'HIGH(1) loop
```

```
            for K in 0 to A'HIGH(1)-1 loop
               for L in 0 to A'HIGH(2)-1 loop
                  if (B(I,J) /= 0.0) then
                     TEMP := A*B(I, J);
                     CONV_OUT(I+K,J+L) :=
                     CONV_OUT(I+K,J+L)+TEMP(K,L);
                  end if;
               end loop;
            end loop;
         end loop;
      end loop;
   end CONV_VAR;

procedure CONV_SIG(
   signal A: inout TOD1_RE_TYP;
   signal B: inout TOD1_RE_TYP;
   signal CONV_OUT: inout TOD1_RE_TYP) is
   variable I, J: INTEGER := 1;
   variable K, L: INTEGER := 0;
   variable TEMP: TOD0_RE_TYP(0 to A'HIGH(1)-1, 0 to A'HIGH(2)-1);
   begin
      for I in 1 to (A'HIGH(1)+B'HIGH(1)-1) loop
         for J in 1 to (B'HIGH(2)+A'HIGH(2)-1) loop
            CONV_OUT(I,J) <= 0.0;
         end loop;
      end loop;
      for J in 1 to B'HIGH(2) loop
         for I in 1 to B'HIGH(1) loop
            for K in 0 to A'HIGH(1)-1 loop
               for L in 0 to A'HIGH(2)-1 loop
                  if (B(I,J) /= 0.0) then
                     TEMP := A*B(I, J);
                     CONV_OUT(I+K,J+L) <=
                     CONV_OUT(I+K,J+L)+TEMP(K,L);
                  end if;
               end loop;
            end loop;
         end loop;
      end loop;
   end CONV_SIG;

end DSP_PRIMS_PKG ;
```

Appendix B: Ptolemy VHDL Code supporting floating point arithmetic

```
-- Code from Universe: test1

entity test1 is
end test1;

architecture test1_structure of test1 is
   component VHDLFReMult port(input1: in REAL; input2: in REAL; output: out
REAL); end component;
   component VHDLFReAdder port(input1: in REAL; input2: in REAL; output: out
REAL); end component;
   component VHDLFMramp generic(init: REAL; step: REAL); port(output: out REAL);
end component;
   component VHDLFMdc generic(level: REAL); port(output: out REAL); end
component;
   component VHDLFFhole port(input: in REAL); end component;
   component VHDLFMgain generic(gain: REAL); port(input: in REAL; output: out
REAL); end component;
   signal ReMult1_output: REAL;
   signal ReMult2_output: REAL;
   signal ReMult3_output: REAL;
   signal ReMult4_output: REAL;
   signal ReAdder1_output: REAL;
   signal Mramp1_output: REAL;
   signal Mdc1_output: REAL;
   signal Mramp2_output: REAL;
   signal Mgain1_output: REAL;
```

```
  signal ReAdder2_output: REAL;
  signal Mdc2_output: REAL;


begin
  ReMult1:VHDLFReMult port map(input1 => Mramp1_output, input2 =>
Mramp2_output, output => ReMult1_output);--2
  ReMult2:VHDLFReMult port map(input1 => Mdc1_output, input2 => Mdc2_output,
output => ReMult2_output);--2
  ReMult3:VHDLFReMult port map(input1 => Mdc1_output, input2 =>
Mramp2_output, output => ReMult3_output);--2
  ReMult4:VHDLFReMult port map(input1 => Mdc2_output, input2 =>
Mramp1_output, output => ReMult4_output);--2
  ReAdder1:VHDLFReAdder port map(input1 => ReMult3_output, input2 =>
ReMult4_output, output => ReAdder1_output);--3
  Mramp1:VHDLFMramp generic map(init => 0.0, step => 1.0) port map(output =>
Mramp1_output);--1
  Mdc1:VHDLFMdc generic map(level => 0.0) port map(output => Mdc1_output);--1
  Mramp2:VHDLFMramp generic map(init => 0.0, step => 1.0) port map( output =>
Mramp2_output);--1
  Fhole1:VHDLFFhole port map(input => ReAdder2_output);--5
  Fhole2:VHDLFFhole port map(input => ReAdder1_output);--4
  Mgain1:VHDLFMgain generic map(gain => -1.0) port map(input => ReMult2_output,
output => Mgain1_output);--3
  ReAdder2:VHDLFReAdder port map(input1 => ReMult1_output, input2 =>
Mgain1_output, output => ReAdder2_output);--4
  Mdc2:VHDLFMdc generic map(level => 0.0) port map(output => Mdc2_output);--1
end test1_structure;

configuration test1_parts of test1 is
for test1_structure
 for all:VHDLFReMult use entity work.VHDLFReMult(VHDLFReMult_behavior); end
for;
for all:VHDLFReAdder use entity work.VHDLFReAdder(VHDLFReAdder_behavior);
end for;
  for all:VHDLFMramp use entity work.VHDLFMramp(VHDLFMramp_behavior); end
for;
   for all:VHDLFMdc use entity work.VHDLFMdc(VHDLFMdc_behavior); end for;
   for all:VHDLFFhole use entity work.VHDLFFhole(VHDLFFhole_behavior); end for;
  for all:VHDLFMgain use entity work.VHDLFMgain(VHDLFMgain_behavior); end
for;
end for;
end test1_parts;
```

Appendix C: SAR VHDL model

The below math package is provided by the IEEE VHDL math study group. It is included here for the sine and cosine functions used in the SAR VHDL model.

```
package IEEE_math is

  function  COS ( X : REAL ) return REAL;
        --cosine function

  function SIN ( X : REAL ) return REAL;
        --sine function

end IEEE_math;

package body IEEE_math is

  --commonly used constants
  --
  constant  MATH_PI :  real := 3.14159_26535_89793_23846; --value of pi
  constant  HALF_PI :   real := MATH_PI/2.0;
  constant  MAX_ITER :  integer := 27;
  constant KC : REAL := 6.0725293500888142e-01; -- constant for cordic
  constant  MATH_E :   real := 2.71828_18284_59045_23536;
                                                    -- value of e

  -- some type declarations for cordic operations
```

```
--
type REAL_VECTOR is array (NATURAL range <>) of REAL;
type NATURAL_VECTOR is array (NATURAL range <>) of NATURAL;
subtype REAL_VECTOR_N is REAL_VECTOR (0 to max_iter);
subtype REAL_ARR1_TYP_3 is REAL_VECTOR (0 to 2);
subtype QUADRANT is INTEGER range 0 to 3;
type CORDIC_TYP_MODE_TYPE is (ROTATION, VECTORING);
--
--auxiliary function used for CORDIC
--
function POWER_OF_2_SERIES (d : NATURAL_VECTOR; initial_value : REAL;
        number_of_values : NATURAL) return REAL_VECTOR is

  variable v : REAL_VECTOR (0 to number_of_values);
  variable temp : REAL := initial_value;
  variable flag : boolean := true;
begin
 for i in 0 to number_of_values loop
   v(i) := temp;
   for p in d'range loop
     if i = d(p) then
       flag := false;
     end if;
   end loop;
   if flag then
     temp := temp/2.0;
   end if;
   flag := true;
  end loop;
  return v;
end POWER_OF_2_SERIES;
--
--constants used for sine and cosine functions
--
constant two_at_minus : REAL_VECTOR := POWER_OF_2_SERIES(
                              NATURAL_VECTOR'(100,90),
                              1.0,
                              MAX_ITER);
constant epsilon : REAL_VECTOR_N := (
                        7.8539816339744827e-01,
                        4.6364760900080606e-01,
                        2.4497866312686413e-01,
                        1.2435499454676144e-01,
```

```
                              6.2418809995957351e-02,
                              3.1239833430268277e-02,
                              1.5623728620476830e-02,
                              7.8123410601011116e-03,
                              3.9062301319669717e-03,
                              1.9531225164788189e-03,
                              9.7656218955931937e-04,
                              4.8828121119489829e-04,
                              2.4414062014936175e-04,
                              1.2207031189367021e-04,
                              6.1035156174208768e-05,
                              3.0517578115526093e-05,
                              1.5258789061315760e-05,
                              7.6293945311019699e-06,
                              3.8146972656064960e-06,
                              1.9073486328101870e-06,
                              9.5367431640596080e-07,
                              4.7683715820308876e-07,
                              2.3841857910155801e-07,
                              1.1920928955078067e-07,
                              5.9604644775390553e-08,
                              2.9802322387695303e-08,
                              1.4901161193847654e-08,
                              7.4505805969238281e-09
                             );
 --
 -- auxiliary functions for cordic algorithms
 --

function CORDIC_TYP ( x0 : REAL;
                y0 : REAL;
                z0 : REAL;
                 n : NATURAL;
      CORDIC_TYP_MODE : CORDIC_TYP_MODE_TYPE
                ) return REAL_ARR1_TYP_3 is
   variable x : REAL := x0;
   variable y : REAL := y0;
   variable z : REAL := z0;
   variable x_temp : REAL;
begin
   if CORDIC_TYP_MODE = ROTATION then
      for k in 0 to n loop
         x_temp := x;
```

```
        if ( z >= 0.0) then
            x := x - y * two_at_minus(k);
            y := y + x_temp * two_at_minus(k);
            z := z - epsilon(k);
        else
            x := x + y * two_at_minus(k);
            y := y - x_temp * two_at_minus(k);
            z := z + epsilon(k);
        end if;
      end loop;
   else
      for k in 0 to n loop
         x_temp := x;
         if ( y < 0.0) then
            x := x - y * two_at_minus(k);
            y := y + x_temp * two_at_minus(k);
            z := z - epsilon(k);
         else
            x := x + y * two_at_minus(k);
            y := y - x_temp * two_at_minus(k);
            z := z + epsilon(k);
         end if;
      end loop;
   end if;
  return REAL_ARR1_TYP_3'(x, y, z);
end CORDIC_TYP;


function  SIN (x : real ) return real is
      -- returns sin X; X in radians

    variable n : INTEGER;
begin
   if (x < 1.6 ) and (x > -1.6) then
      return    CORDIC_TYP( KC, 0.0, x, 27, ROTATION)(1);
   end if;
   n := INTEGER( x / HALF_PI);
   case QUADRANT( n mod 4 ) is
   when 0=>
     return  CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(1);

   when 1=>
     return  CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(0);
```

```
    when 2=>
      return -CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(1);

    when 3=>
      return -CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(0);
    end case;
  end SIN;


  function COS (x : REAL) return REAL is
        -- returns cos X; X in radians

    variable n : INTEGER;
  begin
    if (x < 1.6 ) and (x > -1.6) then
      return CORDIC_TYP( KC, 0.0, x, 27, ROTATION)(0);
    end if;
    n := INTEGER( x / HALF_PI);
    case QUADRANT( n mod 4 ) is
    when 0=>
      return  CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(0);

    when 1=>
      return -CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(1);

    when 2=>
      return -CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(0);

    when 3=>
      return  CORDIC_TYP( KC,0.0,x - REAL(n) * HALF_PI,27,ROTATION)(1);
    end case;
  end COS;


end IEEE_math;



package types_pkg is

  -- complex type representation
  type COMPLX_TYP is
```

```
    record
      RE: REAL;
      IMAG: REAL;
    end record;
  -- array of reals with index starting from one
  type ARR1_RE_TYP is array ( POSITIVE range <> ) of REAL;
  -- array of reals with index starting from zero
  type ARR0_RE_TYP is array ( NATURAL range <> ) of REAL;
  -- array of complex numbers with index starting from zero
  type ARR0_CMPLX_TYP is array ( NATURAL range<> ) of COMPLX_TYP;
  -- array of complex numbers with index starting from one
  type ARR1_CMPLX_TYP is array ( POSITIVE range <> ) of COMPLX_TYP;
  -- two dimensional array of complex numbers with index starting from zero
  type TOD0_CMPLX_TYP is array ( NATURAL range <>, NATURAL range <> ) of
  COMPLX_TYP;
  -- two dimensional array of complex numbers with index starting from one
  type TOD1_CMPLX_TYP is array (POSITIVE range <>, POSITIVE range <> ) of
  COMPLX_TYP;
  -- two dimensional array of reals with index starting from 1.
  type TOD1_RE_TYP is array ( POSITIVE range <>, POSITIVE range <> ) of REAL;
  -- two dimensional array of reals with index starting from 1.
  type TOD0_RE_TYP is array ( NATURAL range <>, NATURAL range <> ) of REAL;

end types_pkg;
```

A typical SAR VHDL model which has four pulses with each pulse consisting of four samples is given below.

```
--------------------------------------------------------------------------------
--
-- TITLE : SAR VHDL model                    status : developed
--
-- PATH : /usr4/home/student/gummadi/RA/SAR/final_ver
--
-- Analyzed using : Synopsys 1076 VHDL analyzer Version 3.0a-10052
--
-- Simulated using : Synopsys 1067 VHDL analyzer Version 3.0a-11594
--
-- REFERENCES
--
-- 1. SYNTHETIC APERTURE RADAR BY J.PATRICK FITCH
-- 2. RASSP BENCHMARK BY MIT LINCOLN LABORATORY
```

```
-- 3. MULTISENSOR VHSIC SIGNAL PROCESSOR REQUIREMENTS ANALYSIS
     NADC
--
-- Authors Dr F. Gail Gray & Ram Gummadi
--
-- Modifications Log :
--
-- Modified By :                          Modified On :
--
-- Modifications :
--
--------------------------------------------------------------------------------

use work.DSP_PRIMS1_PKG.all; -- Appendix A
use work.IEEE_math.all;
use work.bascfuncs.all;
use work.types_pkg.all;

package PRIMS is

   -- data type for processing array
   type PROC_TYP is array(1 to 4) of ARR1_CMPLX_TYP(1 to 4);

   -- data type for final output array
   type IMAG_ARR is array(1 to 4) of ARR1_CMPLX_TYP(1 to 4);

   -- procedure for video to baseband I/Q conversion stage
   procedure VBIQ(
                  variable HH_DATA: in ARR1_RE_TYP(1 to 24);
                  variable  V2BHH: out ARR1_CMPLX_TYP(1 to 4));

   -- procedure for range compression stage
   procedure RG_COMPR(
                  variable V2BHH: inout ARR1_CMPLX_TYP(1 to 4) ;
                  variable    RG_OUT: inout ARR1_CMPLX_TYP(1 to 4));

   -- procedure for cornerturn
   procedure CORN_TURN(
                  variable PROC_ARRHH: in PROC_TYP;
                  variable  FIN_PROC: out PROC_TYP);

   -- procedure for azimuth compression
   procedure AZI_COMPR(
```

```
                variable  FIN_PROC: in PROC_TYP;
                variable SAR_OUT: inout IMAG_ARR);

end PRIMS;


package body PRIMS is

  -- value of pi
  constant  MATH_PI :  real := 3.14159_26535_89793_23846;

  procedure VBIQ(  variable HH_DATA: in ARR1_RE_TYP(1 to 24);
                  variable V2BHH: out ARR1_CMPLX_TYP(1 to 4)) is
    variable P: INTEGER := 1;
    variable P1: INTEGER := 1;
    variable EVEN_HH: ARR1_RE_TYP(1 to 12);
    variable ODD_HH: ARR1_RE_TYP(1 to 12);
    variable TEMP: ARR1_RE_TYP(1 to 12);
    -- numerator coefficients for even pulse
    variable EVEN_COEFF: ARR0_RE_TYP(0 to 7) := (-0.021133,
                                                   0.055895,
                                                  -0.148449,
                                                   0.406139,
                                                   0.917516,
                                                  -0.067483,
                                                  -0.011912,
                                                   0.019827);
    -- denominator coefficients zero for an FIR filter
    variable DEN_COEFF: ARR1_RE_TYP(1 to 2) :=  (0.0,0.0);
    -- numerator coefficients for odd pulse
    variable ODD_COEFF: ARR0_RE_TYP(0 to 7) :=  (0.019827,
                                                 -0.011912,
                                                 -0.067483,
                                                 0.917516,
                                                 0.406139,
                                                 -0.148449,
                                                 0.055895,
                                                -0.021133);
    begin
        P := 1;
        P1 := 1;
        -- modulating the pulses by (-1)**n
      for I in 1 to 24 loop
```

```
      if (I rem 2 = 0) then
        EVEN_HH(P) := REAL((-1)**(P+1))*HH_DATA(I);
        P := P+1;
      else
        ODD_HH(P1) := REAL((-1)**(P1+1))*HH_DATA(I);
        P1 := P1+1;
      end if;
    end loop;
      -- pass the even pulse through FIR filter
    TEMP := EVEN_HH;
    FILT_VAR(TEMP, DEN_COEFF, EVEN_COEFF, EVEN_HH);
      -- pass the odd pulse through FIR filter
    TEMP := ODD_HH;
    FILT_VAR(TEMP, DEN_COEFF, ODD_COEFF, ODD_HH);
      -- Truncate the first 8 samples
    for I in 1 to 4 loop
      V2BHH(I) := (ODD_HH(I+7), EVEN_HH(I+7));
    end loop;
  end VBIQ;

procedure RG_COMPR(  variable V2BHH: inout ARR1_CMPLX_TYP(1 to 4) ;
                     variable RG_OUT: inout ARR1_CMPLX_TYP(1 to 4)) is
  variable FORW: INTEGER := -1;
  variable TAYLW: ARR1_CMPLX_TYP(1 to 4) :=
                                        ((0.40801, 0.0),
                                         (-0.408012, 0.0),
                                         (0.408019, 0.0),
                                         (-0.408031, 0.0));
  begin
      -- Multiply the samples with taylor weights
    for I in 1 to 4 loop
      V2BHH(I) := V2BHH(I)*TAYLW(I);
    end loop;
      -- fourier transform the resulting pulse
    FFT_VAR(V2BHH, FORW, RG_OUT);
  end RG_COMPR;

procedure CORN_TURN(  variable PROC_ARRHH: in PROC_TYP;
                      variable FIN_PROC: out PROC_TYP) is
  variable TEMP: ARR1_CMPLX_TYP(1 to 4);
  variable TEMP1: ARR1_CMPLX_TYP(1 to 4);
  variable TEMP2: ARR1_CMPLX_TYP(1 to 4);
  variable FORW: INTEGER := -1;
```

```
    begin
          -- corner turn the processing array
      for J in 1 to 4 loop
            for I in 1 to 4 loop
               TEMP := PROC_ARRHH(I);
                  TEMP1(I) := TEMP(J);
            end loop;
       --fourier transform for convolution process
           FFT_VAR(TEMP1, FORW, TEMP2);
           FIN_PROC(J) := TEMP2;
       end loop;
end CORN_TURN;

   procedure AZI_COMPR(  variable FIN_PROC: in PROC_TYP;
                         variable  SAR_OUT:inout IMAG_ARR) is
         variable K: INTEGER;
        variable I: INTEGER := 1;
        variable TEMP: ARR1_CMPLX_TYP(1 to 4);
        variable TEMP1: ARR1_CMPLX_TYP(1 to 4);
        variable IFOR: INTEGER := 1;
        variable OUT_KERNS: ARR1_CMPLX_TYP(1 to 4) :=
                                            ((38.1769, 4.33472),
                                             (4.07455, -37.6975),
                                             (-37.9898, -3.95914),
                                             (-3.78726, 38.4994));
        begin
           while (I <= 4) loop
           TEMP := FIN_PROC(I);
                 -- multiply with convolution kernels
           for K in 1 to 4 loop
                 TEMP(K) := TEMP(K)*OUT_KERNS(K);
              end loop;
                 -- inverse fourier transform to end the convolution process
           FFT_VAR(TEMP, IFOR, TEMP1);
              SAR_OUT(I) := TEMP1;
              I := I+1;
           end loop;
    end AZI_COMPR;

end PRIMS;

use work.DSP_PRIMS1_PKG.all;
use work.PRIMS.all;
```

```
use STD.textio.all;
use work.types_pkg.all;

    entity SAR is
    end SAR;

    architecture BEHAV of SAR is
        begin
            process
                variable VLINE: LINE;
                variable V: REAL;
                file INVECT: TEXT is in "ASTEST.ADTS";
                variable HH_TB: ARR1_RE_TYP(1 to 24);
                variable I: INTEGER := 1;
                variable CHECK1 : BIT := '0';
                variable K: INTEGER := 1;
                variable V2BHH: ARR1_CMPLX_TYP(1 to 4);
                variable RG_OUT: ARR1_CMPLX_TYP(1 to 4);
                variable FIN_PROC: PROC_TYP;
                variable PROC_ARRHH: PROC_TYP;
                variable SAR_OUT: IMAG_ARR;
                variable B: INTEGER := 1;
                variable B1: INTEGER := 1;
                begin
                    L: while not ENDFILE(INVECT) loop
                            READLINE(INVECT, VLINE);
                            READ(VLINE, V);
                            HH_TB(I) := V;
                              -- checks for the total of 24 (odd + even) samples
                            if (I= 24) then
                                CHECK1 := '1';
                                I := 1;
                            end if;
                            if (CHECK1 ='1') then
                                -- video to baseband I/Q conversion
                                VBIQ(HH_TB, V2BHH);
                                -- range compression stage
                                RG_COMPR(V2BHH, RG_OUT);
                                -- collect all the pulses in the processing array
                                PROC_ARRHH(K) := RG_OUT;
                                if (K=4) then
                                      -- corner turn the processing array
                                CORN_TURN(PROC_ARRHH, FIN_PROC);
```

```
                    -- azimuth compression stage
                AZI_COMPR(FIN_PROC, SAR_OUT);
                end if;
                CHECK1 := '0';
                K := K+1;
                next L;
            end if;
            I := I+1;
        end loop;
        wait;
    end process;
end BEHAV;
```

Appendix D: VHDL code for the ATR primitives

```
-- Title: PACKAGE DSP_PRIMS1
--
-- Purpose: VHDL declarations for package dsp primitives which contains common
--        real constants, common trigonometric functions, primitve  dsp
--        procedures, and common complex arithmetic functions
--
-- References:1.. PACKAGE MATH_REAL by IEEE VHDL Math Package Study Group
--          2.. SIGNAL PROCESSING ALGORITHMS IN FORTRAN AND C by ----
--              STEARNS,DAVID
--          3.. DISCRETE TIME SIGNAL PROCESSING by OPPENHEIM & SCHAFER
--          4.. Guidelines for VHDL models in a team environment
--                                        by janick bergeron

------------------------------------------------------------------------
use work.dsp_prims_pkg.all;

package DSP_PRIMS1_PKG is
  --
type COMPLX_TYP is record
               RE: REAL;
               IMAG: REAL;
  end record;
  type TOD1_RE_TYP is array(POSITIVE range <>, POSITIVE range <>) of
  REAL;
  type TOD1_CMPLX_TYP is array(POSITIVE range <>, POSITIVE range <>) of
  COMPLX_TYP;

procedure CONV_SIG(
    signal A: inout TOD1_RE_TYP;
    signal B: inout TOD1_RE_TYP;
    signal CONV_OUT: inout TOD1_RE_TYP);

  procedure FFT_VAR(
    variable DATA: inout TOD1_CMPLX_TYP;
    variable ISIGN: in INTEGER;
    variable FFT_OUT: out TOD1_CMPLX_TYP );
```

```
end DSP_PRIMS1_PKG;

package body DSP_PRIMS1_PKG is

--
   --commonly used constants
   --
   constant  MATH_PI :  real := 3.14159_26535_89793_23846; --value of pi
   constant  HALF_PI :   real := MATH_PI/2.0;

   function "+" ( X: COMPLX_TYP ;
                Y: COMPLX_TYP
              ) return COMPLX_TYP is
     -- if X=(a,b)  & Y=(c,d)  it returns ((a+c),(b+d))
   begin
        return ( X.RE+Y.RE, X.IMAG+Y.IMAG ) ;
   end "+" ;

   function "*" ( X : COMPLX_TYP ;
                Y: COMPLX_TYP
              ) return COMPLX_TYP is
     --if X=(a,b) & Y=(c,d)  it returns ((a*c-b*d),(b*c+a*d))
   begin
        return ( X.RE*Y.RE-X.IMAG*Y.IMAG , X.IMAG*Y.RE+X.RE*Y.IMAG ) ;
   end "*" ;


procedure FFT_VAR( variable DATA: inout TOD1_CMPLX_TYP;
                          variable ISIGN: in INTEGER;
                       variable FFT_OUT: out TOD1_CMPLX_TYP ) is
    variable J: INTEGER;
    variable I: INTEGER;
    variable TEMP: ARR1_CMPLX_TYP(1 to DATA'HIGH(1));
    variable TEMP2: ARR1_CMPLX_TYP(1 to DATA'HIGH(2));
    variable TEMP3: ARR1_CMPLX_TYP(1 to DATA'HIGH(2));
    variable ISI: INTEGER := -1;
    variable DATA1: TOD1_CMPLX_TYP(1 to DATA'HIGH(2),1 to DATA'HIGH(1));
    variable TEMP1: ARR1_CMPLX_TYP(1 to DATA'HIGH(1));
    variable DATA2: TOD1_CMPLX_TYP(1 to DATA'HIGH(1),1 to DATA'HIGH(2));
    begin
      if (ISIGN=1) then
        for I in 1 to DATA'HIGH(1) loop
          for J in 1 to DATA'HIGH(2) loop
```

```
            DATA(I,J) := CONJ(DATA(I,J));
          end loop;
        end loop;
      end if;
      for J in 1 to DATA'HIGH(2) loop
            for I in 1 to DATA'HIGH(1) loop
              TEMP(I) := DATA(I,J);
            end loop;
          FFT_VAR(TEMP, ISI, TEMP1);
          for I in 1 to DATA'HIGH(1) loop
            DATA1(J,I) := TEMP1(I);
          end loop;
      end loop;
     for J in 1 to DATA'HIGH(1) loop
          for I in 1 to DATA'HIGH(2) loop
             TEMP2(I) := DATA1(I,J);
          end loop;
          FFT_VAR(TEMP2, ISI, TEMP3);
          for I in 1 to DATA'HIGH(2) loop
            DATA2(J,I) := TEMP3(I);
          end loop;
      end loop;
     if (ISIGN=1) then
        for I in 1 to DATA'HIGH(1) loop
          for J in 1 to DATA'HIGH(2) loop
             DATA2(I,J) := CONJ(DATA2(I,J))/(DATA'HIGH(1)*DATA'HIGH(2));
          end loop;
        end loop;
     end if;
     FFT_OUT := DATA2;
   end FFT_VAR;

procedure CONV_SIG(
      signal A: inout TOD1_RE_TYP;
      signal B: inout TOD1_RE_TYP;
      signal CONV_OUT: inout TOD1_RE_TYP) is
      variable I, J: INTEGER := 1;
      variable K, L: INTEGER := 0;
      variable TEMP: TOD0_RE_TYP(0 to A'HIGH(1)-1, 0 to A'HIGH(2)-1);
      begin
        for I in 1 to (A'HIGH(1)+B'HIGH(1)-1) loop
          for J in 1 to (B'HIGH(2)+A'HIGH(2)-1) loop
             CONV_OUT(I,J) <= 0.0;
          end loop;
```

```
        end loop;
        for J in 1 to B'HIGH(2) loop
           for I in 1 to B'HIGH(1) loop
              for K in 0 to A'HIGH(1)-1 loop
                 for L in 0 to A'HIGH(2)-1 loop
                    if (B(I,J) /= 0.0) then
                       TEMP := A*B(I, J);
                       CONV_OUT(I+K,J+L) <=
                       CONV_OUT(I+K,J+L)+TEMP(K,L);
                    end if;
                 end loop;
              end loop;
           end loop;
        end loop;
     end CONV_SIG;

end DSP_PRIMS1_PKG ;
```

Vita

Ram Gummadi was born on the 5th of May, 1972 in Gudivada, India. He graduated with a Bachelor of Engineering degree in Electronics and Communication Engineering from the J.N.T.U College of Engineering, in May 1993. He attended graduate school at Virginia Polytechnic Institute and State University and received a Master of Science degree in Electrical Engineering in April 1995. He has been employed with Texas Instruments, Texas since April 1995.

Ram Gummadi