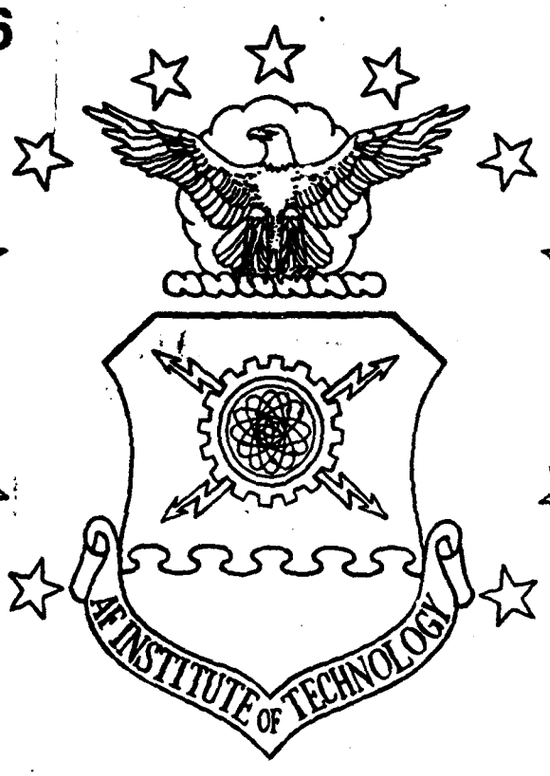


DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/91D-19

DESIGN OF STYLE-V -- A TRANSLATOR TO CONVERT STANDARD VHDL INTO A STYLIZED FORM FOR AUTOMATED MICROCODE GENERATION

THESIS

Dennis A. Rumbley, Captain, USAF

AFIT/GCS/ENG/91D-19

Approved for public release; distribution unlimited

AFIT/GCS/ENG/91D-19

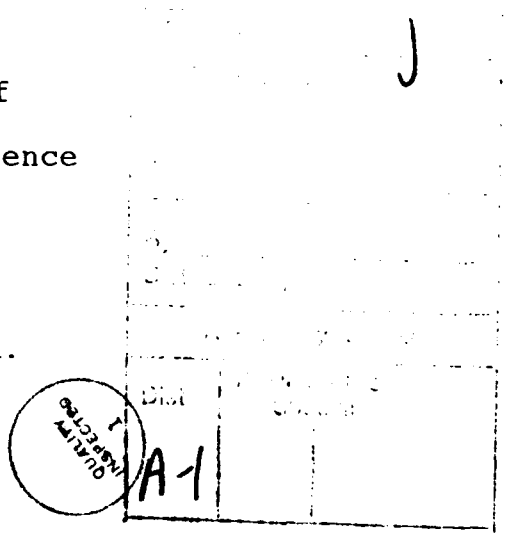
DESIGN OF STYLE-V --
A TRANSLATOR TO CONVERT STANDARD VHDL
INTO A STYLIZED FORM FOR
AUTOMATED MICROCODE GENERATION

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science

Dennis A. Rumbley, B.S., M.S.A.
Captain, USAF

December 1991



Preface

The purpose of this thesis was to develop a translator called Style-V to translate IEEE standard VHDL into a specially styled VHDL defined for the Integrated Design Automation System (IDAS). The two most difficult challenges of the translation were type conversion and function mapping. The remaining challenges were mostly textual conversions.

My original goal was to completely develop a translator for most of standard VHDL. However, the number and types of mappings required to fully implement this translator and the amount of work required to analyze, design, and implement all modules was very much more than I could finish in just one thesis cycle. Therefore, I reduced the scope of the thesis to defining the mappings, analyzing how to do the mappings, performing a manual simulation of a representative subset of the mappings, and implementing one mapping.

I could not have completed this thesis without the support of several people. Specifically, I thank Luis Concha, Russell Milliron, Donald Blankenship, Curtis Winstead, Ronald Comeau, and my advisor, Kim Kanzaki for sharing their technical expertise. I also thank my committee members, Keith Jones and Mark Mehalic, for sharing their time and advise. I thank my wife for her constant support, especially in the **final hours**. Finally, I thank God for giving me the ability and grace to complete this effort.

Dennis A. Rumbley

Table of Contents

List of Figures

Abstract

This thesis provides an analysis and preliminary design of Style_V, a source-to-source computer language translator. Style-V converts IEEE standard VHDL into a special style of VHDL defined for a commercial tool, the Integrated Design Automation System (IDAS). Thirteen mappings between standard VHDL and the IDAS subset were identified. The mappings were analyzed using Domain Analysis and Modern Structured Analysis techniques. Four processes covering several of the mappings were completely analyzed. One mapping to convert CASE statements to IF statements was implemented. Since the IDAS restricts designs to bit logic, a method for representing multilevel logic with bit logic was devised. Unacceptable multiple process architectures were converted to multiple single process architectures which are acceptable to IDAS. The IDAS microcode generator does not recognize user-defined procedures, but in one case, mapping user-defined procedures to IDAS defined procedures was not possible. In general, this problem amounts to showing two programs are functionally equivalent. Exhaustive testing was ruled out since proving two 32-bit adders are equivalent would take over 11 billion years at 100 procedure runs per second. The program equivalence problem was not solved by this thesis. Useful results were obtained, though IDAS failed to work.

DESIGN OF STYLE-V -- A TRANSLATOR TO CONVERT STANDARD VHDL INTO A STYLIZED FORM FOR AUTOMATED MICROCODE GENERATION

I. Introduction

1.1 Background

The process of developing electronic components, especially very large scale integrated (VLSI) circuits, has matured during the past decade. Designers used to plan out a design, draw a schematic, and go to an electronics laboratory where they used wires and breadboards to implement a prototype of the design. If the prototype worked, the design could be finalized and readied for production. Component prototyping was an especially error-prone and costly activity.

Another factor which caused component prototyping to become costly and error-prone was the evolution of electronic hardware. When a designer could only get hundreds or even a few thousand components on a single chip, an experienced designer could manage the activity. Now that designers can place hundreds of thousands of components on a chip, the activity becomes complex and therefore error prone. Obviously, the cost to produce a correct design also rises

with the complexity since more time is required to determine the causes of deficiencies in the design.

A solution for the aforementioned problems has evolved. With the advent of computer languages developed to enhance the hardware design process, much of the error-prone and costly work could be done by computer simulation. The ability to simulate a design on a computer made it unnecessary for a designer to spend the considerable time and money required to test a design with breadboard methods. Another benefit was a reduction in the number of interactions between a designer and the fabricating activity to produce a correct component.

Realizing the importance of the hardware description languages and their utility for reducing errors and development time and cost, the Department of Defense developed the VHSIC (Very High Speed Integrated Circuit) Hardware Description Language (VHDL) as a standard for DoD digital electronic hardware development efforts. VHDL provides any combination of behavioral, structural, or temporal views of digital electronic design. The most significant reason VHDL was more desirable than earlier hardware description languages was its robust accounting for component and circuit timing characteristics. Timing is a crucial factor in the physical behavior of a circuit and accounting for timing during development reduced design risk. As a result of the DoD actions and a general need for standardization throughout

the hardware design community, the Institute of Electrical and Electronics Engineers (IEEE) adopted VHDL as an industry standard in December 1987. This standard VHDL has become known as IEEE-1076 (Lipsett and others, 1989:2).

VHDL became popular in the industry, and many tools were developed to work with VHDL to make the hardware designer's job easier. These tools enhanced design capabilities of designers by providing services such as design analysis, timing analysis, design library management, and even automated microcode generation. However, some of these tools required a special interface, such as a specially styled VHDL input for the JRS Research Laboratories, Incorporated Integrated Design Automation System (IDAS), which is a software system that analyzes algorithms written in the Ada language and produces microcode based on the Ada algorithms for a hardware design (Software User's 1989:2).

To produce microcode, the IDAS converts a VHDL design into a hardware description database (HDD) format stored in the IDAS database. It also processes an application (or set of applications) written in a subset of Ada through the IDAS compilation system. The information from the hardware description database and the Ada applications is used by the IDAS compilation system to produce microcode for the system described by the input VHDL code. A VHDL simulation environment for testing the microcode is also generated. This process is depicted in Figure 1-1.

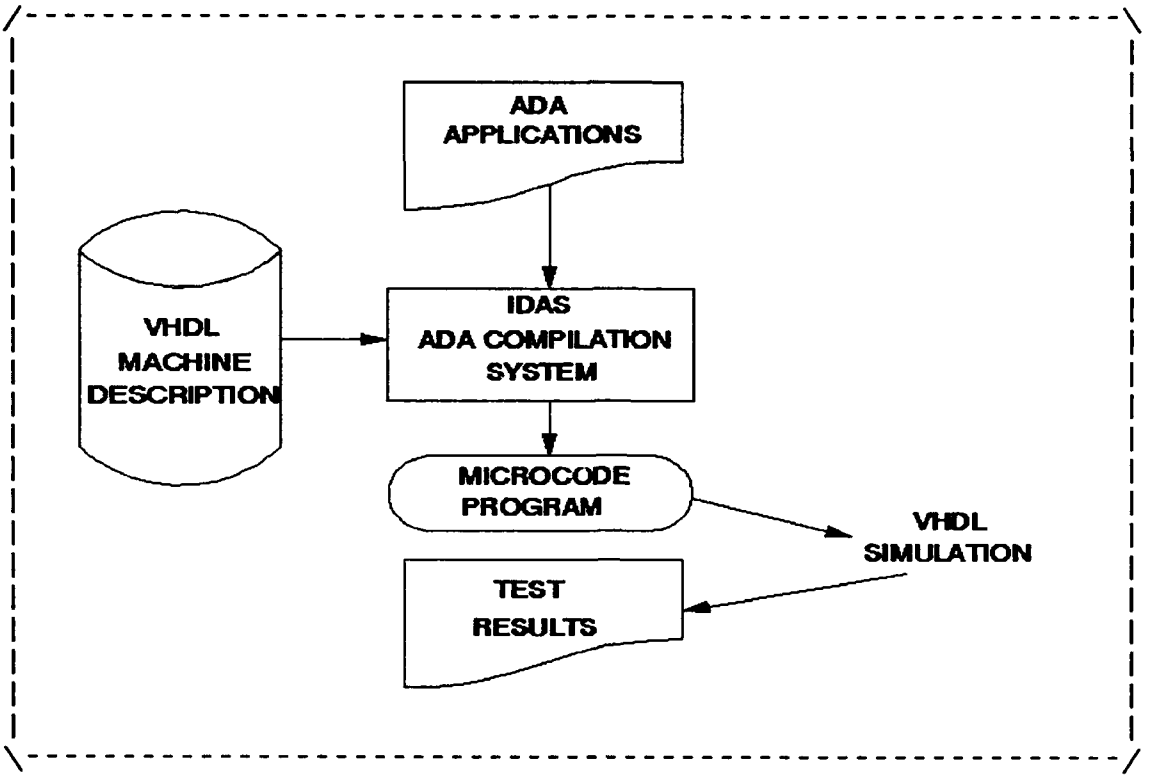


Figure 1-1. Ada to Microcode Compilation
(Integrated Design, 1988:4)

The IDAS claims to provide many benefits. First, a hardware designer need not be concerned about optimizing the microcode to control a hardware design, since the IDAS will do that task. Second, since the microcode generated for a design is optimized for a specific Ada program, the designer can be confident the design works for a desired application base. Third, the designer need not spend the considerable amount of time required to manually generate the proper microcode for any given hardware design. Finally, the designer can test a given design with the IDAS generated and optimized microcode to determine if the design is

satisfactory or if more design work is required. Thus, a designer can save time and money while producing better quality designs (Software User's 1989:4).

To use the IDAS system, a designer must produce a specially styled VHDL input. The stylized VHDL is not simply a change of format, but in reality is the use of a subset of the VHDL language along with some restrictions on typing and format. For instance, the VHDL "case statement" is not supported in the stylized form (VHDL Style Guide 1989:11,13). However, conversion of a "case statement" to an "if statement" provides a satisfactory solution. The stylized VHDL also does not support sensitivity lists in process statements, but it handles sensitivities by using a required "wait statement" as the first statement of a process (VHDL Style Guide 1989:12). These along with several other restrictions make use of the stylized VHDL as a design tool less desirable than use of the standard IEEE VHDL.

1.2 Problem Statement

The IDAS takes as input a highly stylized form of VHDL, converts the VHDL into a Hardware Description Database (HDD), and uses the HDD "in the Ada to Microcode Compiler retargeting process" (VHDL Style Guide 1989:1) to produce microcode for the hardware described by the VHDL. However, VHDL programmers do not naturally produce stylized VHDL code. The freeform IEEE standard VHDL code produced by

designers must then be translated. The translation process consists of analyzing the standard VHDL code and translating it into the subset of VHDL accepted by IDAS. Hand translations of even simple designs would undoubtedly result in several iterations to ensure a correct translation. Therefore, an automated system to translate standard freeform VHDL into the stylized form is required to eliminate the need for programmers to manually translate their designs.

1.3 Research Objective

This thesis has researched and analyzed an automated tool to translate freeform VHDL code into JRS stylized VHDL code acceptable for input to the IDAS. The automated tool is called the Style-V Translator.

1.4 Overview of Current Knowledge

Since the focus of this research was to design a VHDL translator, the direction of the literature search (Chapter 2) concentrated on computer language translation, compiler theory, and related topics.

Much work has been done in the past thirty years in the computer language translation field. The literature review concentrated on articles and books which give insight into semantic and syntactic translation methods and procedures. Since many methods exist, the literature review helped narrow the field to the methods best suited for the Style-V hardware description language translator.

Most computer language translators translate one programming language into another. Some translate between dialects of the same language. In one case, a bidirectional translator was built which could translate between two programming languages.

The main lesson to learn from the existing translators is the necessity for an intermediate form to use during translation. An appropriate intermediate form consists of a canonical representation which is general and extensive enough to represent constructs available in the languages being translated. A translator then need only be able to translate from a source language into the canonical form and from the canonical form to the target language.

For simple translations between dialects of a language, the basic constructs of the general language may serve as the canonical form. The translator needs to be capable of translating high-level constructs into more general constructs of the same language. Also, some reformatting of language sentence structure may be required.

1.5 Assumptions

Whenever a major research effort is undertaken, the researcher must realistically project the expectations of the research, including limitations on the final product. This thesis is no different. The following assumptions document the expected results of this thesis and the expected limitations.

Since VHDL is an extensive hardware description language, describing levels of detail ranging from system level to gate level, not enough time was available for this thesis effort to develop a translator that completely translated all VHDL capabilities. A more reasonable expectation was to translate VHDL into the stylized form for designs down to the component level. A component here is a collection of gate-level primitives which form a component and perform a register-transfer language function (for example, add). This level of detail was sufficient for the IDAS which uses the stylized VHDL. Additionally, the considerable complexity of VHDL and the restricted nature of the subset comprising the stylized VHDL caused the research to center on demonstrating the feasibility for translating a freeform input to the stylized version. Therefore, component level translation of a simple CPU was the first goal of this thesis followed by the translation of portions of the Floating Point Application Specific Processor (FPASP) chip being jointly developed by Rome Laboratory and the Air Force Institute of Technology (AFIT). By completing these translations, enough synthesis of VHDL into the stylized subset was accomplished to demonstrate concept feasibility. A follow-on thesis effort will be required to complete the STYLE-V Translator for all feasible mappings of VHDL constructs.

A second assumption was that JRS' IDAS would work properly. The IDAS was a system developed for the DoD under a Navy contract to JRS Research Laboratories, Incorporated. Contact with agencies known to have used the IDAS indicated certain parts of the IDAS work well; however, none of the agencies contacted used the IDAS in the manner required for this research.

Another assumption was that future releases of the JRS IDAS system would continue to use the stylized format of VHDL as described by JRS. If the style were changed during this research effort, the change would have had a major impact. The research would have continued, but potentially fewer features would have been implemented should such a change have occurred.

1.6 Methodology

Thorough research was paramount to the successful completion of this effort. Manual and automated literature searches provided the basis for the design and implementation of the Style-V translator by providing references to literature containing information regarding the most modern techniques for development and implementation of translator software systems.

After enough information was gathered through the literature review, design followed. Design began with a domain analysis of the problem. Experts on the IDAS system,

the hardware development process, and FPASP design were interviewed.

Following domain analysis, requirements analysis produced what became the system specifications for Style-V. The requirements analysis process consisted of creating a context diagram of the proposed system. As the data flow for the system was analyzed, the initial context diagram was decomposed into lower levels of detail. The system analysis methods described above were originally espoused by Yourdon (Yourdon 1979 & 1989) and by Gane and Sarson (Gane and Sarson 1979) and have been shown successful by many large-scale software system implementations.

The next step was a functional demonstration of the translation process. Manual desktop implementations of portions of the Style-V system showed the feasibility of automating certain parts of Style-V and the apparent infeasibility of automating other parts.

One process of Style-V was chosen for prototyping to demonstrate a part of the translator which could be fully automated. Though it was a rapid prototype, care was taken to encapsulate the modules of the process. Properly encapsulated software modules hide their inner workings and interact with the remainder of the system through well defined interfaces. Since the implemented modules of the Style-V translator were written in C, these interfaces took the form of function calls. Thus, a change to the internal

workings of a function is transparent to the rest of the system. Therefore, changes or additions to the Style-V Translator are localized and easily done.

1.7 Thesis Overview

Chapter 1 provides an introduction to the need for an automated tool to translate standard VHDL into a stylized format. The sections of this chapter cover pertinent background information, a problem statement, the research objective, an overview of current knowledge, some key assumptions, the general methodology used, and an overview of the contents of the written thesis.

Chapter 2 reviews the literature pertinent to the completion of this thesis effort. Many of the books and articles read contained information crucial for the understanding and use of modern software development practices and knowledge necessary to understand and develop Style-V. Besides an introductory section, Chapter 2 includes sections on translation fundamentals, lexical analysis, parsing, grammar disambiguation, examples of translators, and a summary section.

Chapter 3 documents the analysis of the Style-V translator. The sections provide an introduction to requirements analysis, a review of available methods, a discussion of the method chosen, the process of domain analysis for Style-V, the system analysis for Style-V, and a review section.

Chapter 4 documents the manual desktop implementation of the Style-V translator. The sections provide an introduction, a discussion on the selection of the processes of Style-V chosen for implementation, the description of the implementations, a discussion of the lessons learned from the desktop implementation process, and a review section.

Chapter 5 discusses the results, conclusions, and recommendations resulting from this research effort.

Appendix A contains a simple VHDL hardware system description used to demonstrate the concept of using JRS IDAS procedures to produce a working machine.

Appendix B contains extracts from the Floating Point Application Specific Processor (FPASP) used as a real world example for translating standard VHDL into a stylized VHDL subset. Since this appendix includes proprietary Air Force design data, it is maintained in Volume II of this thesis and distribution is limited. See the Appendix B tab of this volume for more details.

Appendix C contains the results of applying Style-V translations to the FPASP code sections of Appendix B. Since this appendix includes proprietary Air Force design data, it is maintained in Volume II of this thesis and distribution is limited. See the Appendix C tab of this volume for more details.

Appendix D contains the implemented modules of the Style-V Translator.

II. Literature Review

2.0 Introduction

This literature review explores current literature on topics critical to the successful completion of this thesis. The topics covered concern issues related to translating computer programs from one form to another.

Some of the articles deal with maintaining the meaning of an input program for use in generating an output program. Other articles deal with the various components of translation systems, specifically the lexical analyzer and parser. The lexical analyzer, often called a scanner, reads an input program and passes each token it recognizes to the parser. The parsing process deals with comparing an input stream of tokens against a language's grammar in such a way to determine the meaning indicated by the input (Allman, 1988:76). To accomplish this task, the parser checks the syntactic correctness of the input. Given the input is syntactically correct, the parser then uses semantic routines to either generate an intermediate representation or generate the final output of the translator. If an intermediate interpretation is generated, a code generator produces the translated program (Fisher and LeBlanc, 1988:11-13,215-220; Aho and Ullman, 1977:7,19,254).

2.1 Translation Fundamentals

Some aspects of language translation can be classified as fundamental to the translation process. Any effort which produces a translator must address the fundamental issues of language translation. Style-V is no exception.

2.1.1 Analysis and Synthesis. Analysis and synthesis are the two actions required for a translator to convert an algorithm in a source language into a similar algorithm in a target language. Analysis determines what actions are ultimately required by the implementation in the target language. Synthesis is the process of transforming the source statements into direct execution or a target language form (Calingaert, 1988:5).

Analysis occurs in three distinct stages. The first stage is lexical analysis where words of the source language are recognized by an examination of the characters found in the input text. The second stage, syntactic analysis, examines and determines the correctness of the structure of the source language input. When the idea of grammar (the proper association of symbols in a language) is incorporated, the syntactic analysis is known as parsing. The final stage of analysis, semantic processing, entails associating the semantic information (meaning) of the input tokens in such a way that the meaning can be maintained in and transformed to a new representation (intermediate code or the target language) (Calingaert, 1988:5,235-6).

2.1.2 Compatibility of Languages. Commonalities and differences of languages must be precisely defined if one hopes to successfully translate between them. These definitions are particularly important when translating between dialects of the same language. Fortunately, techniques which define parts of a language in terms of itself lessen the definition effort (Krieg-Brückner, 1984:299).

Given two languages Alang and Blang, if the language concepts of the sublanguages, Asublang (the sublanguage for a language Alang) and Bsublang (the sublanguage for a language Blang), correspond in a one-to-one manner, then Asublang and Bsublang are said to be directly compatible. Additionally, if the concepts of a language, Aotherlang (a language other than Asublang or Bsublang), are mappable to Asublang, then Aotherlang is said to be indirectly compatible with Bsublang.

If Asublang is complete in the sense that all concepts of Alang can be mapped into it, and Bsublang is complete in the same respect with Blang, the Alang can be mapped to Blang. A problem exists when concepts in the language Alang cannot be mapped to Asublang, at which point Alang cannot be fully mapped to Bsublang, and therefore the languages Alang and Blang are said to be incompatible and translations between them are not generally possible (Krieg-Brückner, 1984:300).

If certain restrictions exist in one language, these restrictions may require an "applicability condition" hold for the other language being translated for a one-to-one translation to be possible. Krieg-Brückner gave the example of the for-loop variable in Ada and Pascal. If a direct translation is to be obtained, the variable must not be assigned values outside the loop body in both language implementations (Krieg-Brückner, 1984:303).

Another key point in having the ability to directly translate between sublanguages is that the sublanguages must have a certain level of expressive power. This expressive power between sublanguages enables one sublanguage to express concepts contained in the other sublanguage (Krieg-Brückner, 1984:303).

The semantic mapping from Alang to Asublang can be thought of as a homomorphism (they look similar) and the equivalence of the semantics can be proven using the semantic definitions of the languages. Asublang is a "subset" language of Alang, and all Alang constructs can map to Asublang constructs. The translation of Asublang to Bsublang would be an isomorphism (they look different) using an equivalent semantic definition kernel (Krieg-Brückner, 1984:304).

2.1.3 Two Fundamental Approaches. The research for this thesis effort determined that two fundamental approaches to program translation currently exist. The most common

approach is transliteration and refinement. An alternate method is to use abstraction and reimplementation. The remainder of this section describes these approaches and provides their respective advantages and disadvantages.

The approach of transliteration and refinement is based on a direct translation of statements in a source language to semantically equivalent statements in a target language. This is done by a translating statements in isolation from the overall context of the program. The output can be in the target language or a semantically similar intermediate language. The refinement step applies "correctness preserving transformations" to the generated target code to improve the quality of it (Waters, 1986:2).

Advantages of transliteration and refinement include a divide and conquer approach, localized nature of the translations, and ease of constructing families of translators. The transliteration step need not be concerned with later refinements -- the basic goal is to obtain a semantically correct translation. Since transliteration takes a localized approach to translation, the knowledge needed for translation is easily determined since it need not consider how special combinations of a target language might implement special combinations of the source language. Finally, translators which use similar methods of either transliteration or refinement can be easily adapted to similar languages (Waters, 1986:7).

Unfortunately, transliteration and refinement have some disadvantages. One of the main disadvantages is that the refinement process is complicated because the transliteration is a localized translation which often results in convoluted code in the target language. Additionally, it is not always practical to translate one construct of a source language into a construct of the target language. Sometimes, a source language contains a primitive which is not supported in the target language and transliteration cannot handle this problem. Finally, current translators suffer from the problem of being able to straightforwardly translate certain constructs most of the time, but on occasion they incorrectly translate the constructs when the translation is difficult or impossible (Waters, 1986:8-9).

Translation can alternatively be done via abstraction and reimplementation. The abstraction process first analyzes the program globally and uses this analysis to gain an understanding of the algorithms in the source program. The reimplementation process uses the knowledge of the source program algorithms to implement equivalent algorithms in the target language. Unlike transliteration, abstraction does not require a knowledge of the target language when analyzing the source language (Waters, 1986:10).

One of the advantages of abstraction and reimplementation over transliteration and refinement is the ability to satisfy the subsidiary goals of language translation -- a

readable translation. This is a natural consequence of the main goal of abstraction -- to simplify reimplementation. A second advantage of abstraction and reimplementation is ability to translate any possible translation task. Finally, "translation via abstraction and reimplementation lends itself to the construction of families of translators" (Waters, 1986:16).

Some of the problems of abstraction and reimplementation include a lack of completeness, the practicality of translation, and the complicated nature of the process. When it is not practical to use abstraction and reimplementation, a translator could fall back to a process of transliteration in most cases since the problems of abstraction and reimplementation are orthogonal to the problems of transliteration and reimplementation -- in other words, when it is not practical to use one method, the translation at that point will usually be easier using the other method. Since abstraction and reimplementation is significantly more complicated than transliteration and reimplementation, when transliteration is practical and little refinement is required, the method of choice is probably transliteration and refinement (Waters, 1986:16).

Calingaert claims that "translation from one machine-independent language to another is rarely feasible."(Calingaert, 1988:330). However, during this research effort,

several examples of source-to-source translators were found. A sampling of such translators are discussed in Section 2.5.

2.2 Lexical Analysis

Lexical analysis is the process of taking an input string and converting it into a sequence of tokens. Normally, some other program will use the tokens produced by the lexical analyzer for further processing (Fox, 1987:51).

Various methods exist to generate a lexical analyzer. Some lexical analyzers are "hand-generated" by a programmer. Hand-generated lexical analyzers can be very efficient but can be difficult to construct. Other lexical analyzers are generated using some type of lexical analyzer generating program. Normally, a programmer can generate a lexical analyzer much quicker and easier with an automated tool than by hand generating techniques.

Numerous lexical analyzer generators exist. Probably the most famous and most widely used lexical analyzer generator is the program called LEX written by Lesk in 1975 (Lesk, 1975). LEX considers an input string of characters and compares them with the definitions of acceptable tokens. The token definitions are in the form of regular expressions.

LEX matches the longest possible token as it reads an input string. Normally, tokens in the input string are delimited by white space characters (blank or tab) or the newline character. LEX identifies tokens by returning an

integer value through the function YYLEX and keeps the string value of the token in the variable YYLVAL.

The capabilities of LEX exceed those of a simple lexical analyzer (Fox, 1987:53,55). When a token is recognized, LEX provides a capability for a programmer to specify programming language (usually C) statements to be executed for each token that is recognized. The actions specified by the programmer are in addition to LEX's identifying the token and string value as stated above. The capability of executing programming statements upon recognition of tokens makes LEX a powerful tool for lexical analysis and text processing.

Besides LEX's power and flexibility, LEX is a readily available resource at AFIT and on other systems used for this thesis. Therefore, due to the availability and power of LEX, it was seriously considered for use in the development of STYLE-V.

2.3 Parsing

Computer languages are generated from Context-Free Grammars (CFG). A CFG is a collection of terminal symbols which represent the alphabet legal for a language, nonterminal symbols which are derived by some combination of terminal symbols, and a finite set of productions which are the rules that define how nonterminal symbols are derived in terms of terminal and nonterminal symbols (Cohen, 1986:247).

An analogy is to think of a program as a paragraph. The computer language statements in the program are the sentences of the paragraph. The identifiers and symbols in the statements are the words of the sentences. Finally, the characters composing the identifiers and symbols represent the alphabet of the language.

The rules which determine if a "sentence" of a computer language is correct are the syntax rules. The job of a parser is to receive tokens (words) from the lexical analyzer and apply the syntax rules to the sentences formed by the words to determine if the sentences are legal.

Semantic processing, on the other hand, deals with what a given sentence (or even program) means. Once a parser determines a portion of the input is syntactically correct, it can call semantic routines to assess the meaning of the input.

Sideri (and others) discussed the use of attribute grammars with no restrictions on the dependencies of the attributes used for parsing. An attribute grammar is recognized by the association of attributes to the nonterminal symbols of a CFG, the "functions on the attributes, and conditions over the attributes" (Sideri and others, 1989:91-92).

Using attribute grammars, Sideri proposed a Semantically Driven Parsing method for context-free languages. An attribute grammar is defined as a reduced CFG, a finite set

of attributes, the domains of the attributes, a set of semantic rules, and a set of semantic conditions. The semantic condition for each production is either satisfied and a correct parse is realized, or the semantic rules operating on the attributes violated a constraint on the attributes, thus signifying an incorrect parse of a production (Sideri and others, 1989:91).

The semantic assessment of the input strings is accomplished during the parsing of the input stream. With this method, meaning is only assigned to semantically correct input strings and any incorrect string is rejected, an advantage over other methods that form and search parse trees for all input strings. Also, attribute reevaluation between multiple parse trees for the same input string is avoided (Sideri and others, 1989:91).

2.4 Disambiguating Grammars

During syntactic analysis, the productions defined by a grammar may lead to conflicting reductions. Normally, some standard or default reduction is chosen over the other conflicting reductions. Since a standard or default production is reduced, the other productions are ignored. Therefore, because conflicting reductions are present, the grammar is ambiguous. An example of ambiguous productions is:

```
P --> c
Q --> c
```

where the parser does not know whether to reduce P or reduce Q (Ganapathi, 1989:25).

Ganapathi provides a solution to disambiguate the way parsers reduce productions that only involves modification of the parser driver, so Ganapathi's solution is applicable to presently available off-the-shelf parser generators. Using Ganapathi's method, disambiguating predicates can be added and disambiguation is done dynamically (Ganapathi, 1989:25,29-30).

According to Ganapathi, a production may have many predicates. However, at least one and only one predicate of a production must evaluate to true and all other predicates of the production must evaluate to false. (Ganapathi, 1989:25-26).

Ganapathi's method is reasonably simple to implement and does not require rewriting the parser generator. When implementing Ganapathi's method,

> conflicting productions in the grammar are expanded with distinct terminal symbols and an ? production is introduced to invoke a predicate-routine and select a production. (Ganapathi, 1989:29)

The following example shows the ease of implementing Ganapathi's method. Given the ambiguous productions:

```
P1:  A --> C
P2:  B --> C
```

replace them by productions using look-ahead disambiguation. Lookahead disambiguation is added by rewriting the ambiguous productions with a new nonterminal and disambiguating

terminals at the end of the ambiguous portion of the production. The new nonterminal is then formed into a production and the disambiguation is triggered. The resulting productions are:

```
P1:  A --> C V Ta
P2:  B --> C V Tb
P3:  V --> ? disambiguate(Ta,Tb)
```

The correct choice between P1 and P2 will occur when P3 reduces and the disambiguate predicate triggers. The predicate reduces by considering the context or by using conditions defined by the system developer (Ganapathi, 1989:30).

In the conclusion of the article, Ganapathi points out two benefits of including semantic predicates as context sensitive parts of the grammar. First, the inclusion of the predicates "contributes to an overall gain in convenience" for the grammar writer (Ganapathi, 1989:32). Finally, "resolution by a linear ordering of predicates permits incremental addition of productions in a grammar" (Ganapathi, 1989:32).

2.5 Examples of Existing Translator Systems

As part of this research effort, a review of existing language translators provided invaluable insight into potential methods for implementing Style-V. The purpose of this section is to provide a review of the articles and books researched for knowledge to complete this thesis.

2.5.1 PasTran -- A Pascal to Ada Translator. PasTran is a translator developed for the purpose of translating Pascal programs to the Ada programming language. Since PasTran translates 95 percent of the standard Pascal language, if the Pascal programs are in standard Pascal, the conversion is virtually automatic and most required editing is for esthetic reasons only (Owen, 1987:423,425).

The translator takes three passes to process a Pascal program. The first pass analyzes the syntax of the Pascal program. The second pass analyzes the semantics. The third and final pass generates the translated Ada code (Owen, 1987:425).

During the translation process, PasTran generates error messages and provides the user with the option of terminating the translation process or continuing. This allows a user to let PasTran translate as much as possible automatically. The user can then hand translate the portions PasTran could not understand. This option allows the translation of nonstandard Pascal and the few standard constructs PasTran is unable to handle (Owen, 1987:425).

A final item of note about PasTran is it's ability to handle constructs of Pascal for which no equivalent Ada construct exists, specifically, the Pascal record WITH clause. PasTran creates a new identifier and uses the Ada dot notation to emulate the Pascal WITH construct.

2.5.2 PCC -- A Pascal to C Translator. PCC is a Pascal to C translator written in the C programming language for portability. It has been ported to several configurations. PCC satisfies two main tasks. First, applications written in standard Pascal which now need further features added (features not available in Pascal) may be converted to C which has the required language features. Second, with PCC it is possible to port Pascal programs to a newly developed computer system which has a C compiler but not one for Pascal (Bothe and others, 1989:60).

The requirements for PCC were derived from the two main uses described above. Since the C programs generated by PCC will be used for further program development, the transformed C code must be comprehensible. Also, the generated C programs must be efficient to maintain the efficiency of the source Pascal programs. Some technology must exist to handle the dialects of Pascal as many dialects exist with extensions to the standard language, the most famous being Turbo Pascal. Since the translator is being developed as a translation tool, it must be portable. Finally, the translator must be efficient since not all Pascal programs run through may be fully tested -- the case when PCC is used as a development tool for new Pascal programs until a Pascal compiler is available (Bothe and others, 1989:61).

Comprehensible C code is generated by PCC because the PCC system follows four principles. The first principle PCC

observes is to preserve the structure of the original Pascal program in the translated C program. The second principle involves using rules for translating language constructs that maintain comprehensibility. Third, identifiers from the Pascal source should be analogous in the translated C code. Fourth, and finally, Pascal comments must be translated to equivalent C comment statements (Bothe and others, 1989:62).

One feature of Pascal that makes abiding by the principle of maintaining original structure is the use of local procedures or functions. Since C does not have this feature (all functions are global), maintaining the functionality will be possible but maintaining the original structure is not (Bothe and others, 1989:62).

Improving the efficiency of a translation is sometimes difficult depending on the optimization scheme chosen. Sometimes complex optimization attempts achieve very little real gain in efficiency. For PCC, it was decided to implement local optimizations, which are simple and obvious, and leave further optimization to the discretion of the using programmer (Bothe and others, 1989:62).

PCC uses a pipelined, four-pass design to perform translations. One pass is for lexical and syntactic analysis. Another pass analyzes the semantics of the Pascal source. A third pass generates the C program. The final pass is uses to beautify the output C program. Using this

architecture, PCC can translate extremely large programs because it limits the size of each pass to 40 KBytes and the symbol table is only maintained as long as the variables are in a current scope -- like a one-pass compiler (Bothe and others, 1989:63).

Three methods exist to handle dialects of Pascal. The first is for the programmer to manually convert the source program into standard Pascal which is accepted by PCC. Second, a nonstandard Pascal program can be partly translated by PCC (about 90 percent) and the remainder done by hand. Thirdly, the PCC itself may be modified to handle the nonstandard constructs of the dialect (Bothe and others, 1989:63-64).

Portability of PCC was maintained first by using the C programming language and second by avoiding language features not supported by some of the existing C compilers (Bothe and others, 1989:64).

Deriving the requirements for PCC based on the uses of translating systems provided insight into the need for proper domain analysis prior to implementing a project. This principle will be applied in Chapter 3 of this thesis.

2.5.3 Small Euclid to Pascal. The Small Euclid to Pascal translator was developed to translate a medium size application (an assembler). Understanding the application and recoding it in Pascal would have been an error prone activity. Developing the translator ensured a correct

translation and provided a tool for future use (Pintelas and others, 1989:93).

Because it is easier to translate to a related language, Pascal was chosen as the target language since Euclid development was based on Pascal. Also, the popularity of Pascal means more machines can process the resulting program code (Pintelas and others, 1989:93).

To avoid portability problems, the translator was written in C and designed to translate to a core of Pascal (those features of Pascal available on all or most machines). Due to this choice, some drawbacks were unavoidable. One drawback is the extended use of goto statements in the translated code. Another drawback is that new logical variables are required in the target code. Further, the descriptive Euclid identifiers must be somehow shortened to a maximum of eight characters. Due to these drawbacks, the translated code is not very readable or well structured. However, since the goal of the translation process was to produce compilable code and not so much for human intervention, this is not too much of a problem. Should humans need to use the output, a pretty printer program could improve the Pascal text (Pintelas and others, 1989:93).

The translator expects semantically correct Small Euclid programs. It does a complete syntax check, but not a exhaustive semantic check on the source (Pintelas and others, 1989:94).

One of the differences between Small Euclid and Pascal which the translator must address is the declaration of constants, variables, types, and procedures. In a Small Euclid program, these may be declared in any order. However, Pascal has a fixed order of declarations for a program (Pintelas and others, 1989:94).

Another difference is the scoping of variables. In Small Euclid, an import list is used to identify which variables are visible to an internal procedure. Small Euclid also provides a constant, type, variable, procedure, or function to be declared as pervasive and then it does not need to be included in an access list to be visible. In Pascal, all variables in the scope of the procedure are visible and are therefore equivalent to the Small Euclid pervasive declarations (Pintelas and others, 1989:94).

Various statements of the two languages differ. The IF statements are different. Pascal uses an IF-THEN-ELSE structure where Small Euclid uses an IF-THEN-(ELSIF-THEN)-ELSE-ENDIF structure. The loop constructs of Pascal are replaced in Small Euclid with a more general loop statement. Unlike Pascal, the Small Euclid CASE statement provides an OTHERWISE clause (Pintelas and others, 1989:95).

Some of the other differences have to do with declarations. In Pascal, local declarations come right after the procedure header, but in Small Euclid they come after the first BEGIN. Also, name equivalence is used for type check-

ing in Pascal (types of items are equal only if the type name matches) whereas Small Euclid uses the harder to check structural equivalence for types (types of items are equal if they have the same structure and their description values are equal) (Pintelas and others, 1989:95).

The translator was designed to perform the translation in three passes. The first pass simply converted a multiple file program into a single file for translation. The second pass was used for textual processes such as the removal of white space, removal of comments, and conversion of upper-case characters to lower case characters. The translator developers decided to remove comments because leaving them in might be confusing as the resultant Pascal program was a drastic change from the input Small Euclid code. The third pass was the most significant and consisted of three basic modules: the lexical analyzer, the symbol table, and the syntax analyzer (Pintelas and others, 1989:98-99).

The tools LEX and YACC were used extensively and a large number of action routines. The lexical analyzer's basic job was to deliver tokens to the syntax analyzer. Since assertion statements were turned into comments, the lexical analyzer handled all assertion statements by delivering the comments to the current output file (Pintelas and others, 1989:99).

The symbol table was needed for several important reasons, some of which follow. Declaration names needed to

be transformed before being transferred to the Pascal output file. Some new names had to be created due to the length restrictions imposed by Pascal. Some structure had to store all information regarding a name for use during the translation process. Finally, declaration names had to be collected before being sent to the Pascal output (Pintelas and others, 1989:99).

The syntax analyzer initialized and managed the symbol table task. It also called the lexical analyzer when tokens were needed. It managed all temporary files. Finally, it checked the input Small Euclid and output Pascal program for syntactic correctness. Since the translator expected a syntactically correct input Small Euclid program, it would attempt translation until a syntax error was recognized and then terminate (Pintelas and others, 1989:100).

The building of the Small Euclid to Pascal translator was a significant effort that was eased by the use of LEX and YACC tools. Yet, the project took ten man months and produced over 3,000 lines of source code (Pintelas and others, 1989:100).

2.5.4 Lisp to Fortran. TAMPR was a program transformation system written in applicative LISP which was translated into Fortran. The method used was to analyze the required transformation and break it into smaller steps. The idea is to preserve the correctness of the program

through a sequence of relatively small alterations (Boyle, 1984:291).

The first idea was to transform the LISP program to Fortran in two stages. The first stage was to convert LISP code to Recursive Fortran. The second stage would then convert the Recursive Fortran into an executable form of Fortran, either Fortran 66 or Fortran 77. In the final design, about 20 levels of transformations were used (Boyle, 1984:292).

The first levels of the transformer converted the input LISP into a form that could be translated into Fortran. The last of these steps resulted in a form of LISP that had nontrivial functions evaluated in the argument part of lambda expressions. This form was then translatable into Recursive Fortran statements with no more than one recursive call. The latter steps involve recursion removal from the Fortran code (Boyle, 1984:292,294).

An interesting benefit of using small transformation steps to translate programs is that intermediate results are available for optimization. In this way, correct and optimized code is used for each input to the next transformation step (Boyle, 1984:294-295).

The transformation of TAMPR from LISP into Fortran was accomplished in a totally automated fashion (no manual intervention), and the resultant Fortran program ran 25 percent faster than the original LISP program on the same

platform. This example showed that a practical approach to synthesizing programs is to use program transformation (Boyle, 1984:296-297).

2.5.5 Ada to Pascal and Pascal to Ada. A method of translation was described that allows bidirectional translation between two languages. In the case of the article, Ada and Pascal were used. The translation method used is to define a subset of each language for which there exists a direct translation into the subset of the other language. PascalA and AdaP were defined and their semantic concepts map to the other's in a simple one-to-one manner. In this sense, PascalA and AdaP represent two equivalent forms of a new language, PascAda (Albrecht and others, 1980:183-184).

Using the similar sublanguage approach, the sublanguages correspond in three ways. First, program semantics are preserved. Second, the form and structure are preserved. Third, local transformations result (Albrecht and others, 1980:184).

The first step of sublanguage definition is to define an extended sublanguage of the original language for which all constructs are mappable to the other original language. Obviously, this extended sublanguage should be as rich as possible. Then determine the mappings between this extended sublanguage and the sublanguage which is the syntactic equivalent of the other original language's sublanguage

(Albrecht and others, 1980:184). Figure 2-1 provides an example to clarify this concept.

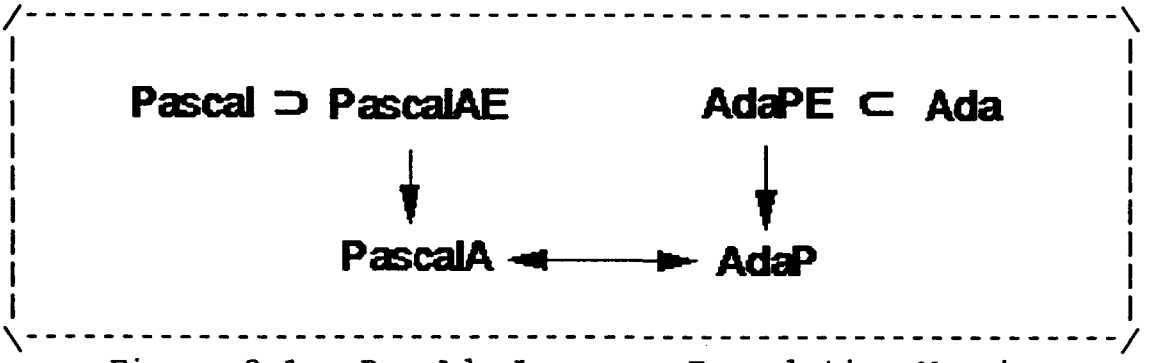


Figure 2-1. PascAda Language Translation Mapping (Albrecht and others, 1980:184)

Consider source-to-source translation between Pascal and Ada depicted in Figure 2-1. Given that not all Pascal constructs may map to Ada, the extended subset of Pascal called PascalAE for which all constructs are mappable to Ada is defined. Next, a similar extended subset for Ada called AdaPE is defined. Now, these extended subset languages are mappable to the sublanguages PascalA and AdaP (respectively) which are easily translated between because they are the two syntactic forms of the PascAda language (Albrecht and others, 1980:184).

The structure of the PascAda system consists of seven routines which provide eight logical units. Four units translate the Pascal side and the other four units translate the Ada side. The unit PascalToTree translates Pascal source into a nonstandard PascalAE tree structure, producing

error messages if language constructs occur for which there is not a PascalAE representation. The unit PascalAEToA does a tree to tree transformation to convert PascalAE into PascalA syntax. A unit called PascalACheck then ensures the resultant tree contains only semantically correct PascalA constructs, and if the tree is in a nonstandard form, it is converted to a standard tree. The TreeToPascal unit produces Pascal code from a standard PascalA tree. Similarly for Ada, the AdaToTree unit constructs a nonstandard tree from Ada source code. The AdaPEToP unit uses a tree-to-tree transformation to convert AdaPE constructs into AdaP syntax. The AdaPCheck unit then checks that only semantically correct constructs are in the tree. Finally, the TreeToAda unit uses a standard AdaP tree to produce Ada code (Albrecht and others, 1980:188).

2.6 Summary

This chapter has reviewed the literature to provide a basis of knowledge for further work toward development of the Style-V translator. Section 2.1 covered the fundamentals of computer language translation. Lexical analysis which is part of every translation task was reviewed in Section 2.2. The parsing task and current work on techniques comprised Section 2.3. The need to ensure language grammars are processed without ambiguity was covered in Section 2.4, and methods were described for disambiguating a

grammar. Finally, Section 2.5 provided examples of translators which have been developed and described some of the techniques used.

The information in this chapter provides the foundation for the research to continue into the requirements definition phase for the Style-V translator -- the subject of Chapter 3 of this thesis.

III. Requirements Analysis

3.0 Introduction

This chapter details the requirements analysis process used to develop the Style-V translator for translating standard IEEE VHDL into a specially styled VHDL.

To design a successful program, the developer must understand the problem at hand. This requires a careful study of all aspects of the problem -- analysis. The Webster's II dictionary defines analysis as "separation of an intellectual or substantial whole into its constituent parts for individual study" (Webster's, 1984:104). In the context of a computer system, the developer must define what is to be done, by whom, when, and how.

For the purposes of this thesis, analysis is the process of specifying the parts of the Style-V Translator system, how they interact, and what they produce. The results of analysis should completely specify the behavior of the software system (Davis, 1990:7,17). The parts of the system are then categorized as hardware, software, and peopleware. Hardware consists of the computers and peripheral equipment on which the software programs and data are processed. The software component consists of programs and data needed to provide the functionality desired of the system. Finally, the peopleware component consists of the users of the system.

This chapter will review the methods available for conducting systems analysis, describe in detail the method chosen, provide a domain analysis of the problem domain, use the domain analysis results to analyze the Style-V system, and end with a summary of the analysis effort.

3.1 Review of Methods

As with most development processes in computer science, no one method of analysis has been adopted as the standard best way. Instead, a plethora of methods exists, and a developer must choose a method (or combination of methods) that suits both the type of problem at hand and the developer's personal preferences, since more than one method may be equally valid.

All current methods of analysis can be classified as either functional or object-oriented analysis. In functional analysis, the problem is analyzed to determine the functions and data needed to solve it. Functional analysis considers the functions and their associated data or control flow. Object-oriented analysis takes a somewhat different approach by determining the objects which exist in the problem space (application domain). The services provided by and required by the objects are identified. Also, the attributes of objects and their relationships are identified (Davis, 1990:46).

Several notations for analysis are common. Data flow diagrams (DFD) were popular for problem solving even before the time of computers, and they are quite applicable to computational problem solving. A DFD represents data flow (by labeled directed arrows) between transformation centers (represented by bubbles), and sources and destinations of data called terminators (Davis, 1990:57).

In addition to DFDs, control flow diagrams (CFDs), Mealy or Moore machine representations, process activation tables, and requirements dictionaries are further analysis tools and are quite helpful when analyzing real-time systems (Davis, 1990:60-61).

A companion for DFDs is the data dictionary (DD). The DD stores information on data items such as name, aliases, descriptions, relations, values, data flows, and structure definitions (Davis, 1990:62).

Another notation is the entity-relationship diagram. Using this notation, a designer identifies the entities in the problem placing them in rectangles. Relations are identified in diamonds and placed between the entities on the diagram. Entity attributes are noted in circles attached to the entity's rectangle.

In the object oriented world, Peter Coad developed a diagram that combines the advantages of DFDs and ER diagrams. His COAD objects contain attributes, suffered operations, and required operations. They also have two types of

structural relationships: classification and assembly. Classification relations show a class (or type) of an object (in_patient and out_patient are of class patient). Assembly relations show a "part-of" relation between objects (heart and liver objects are parts of a person object) (Davis, 1990:63-69).

One technique used by analysts to gain an understanding of a problem domain is Domain Analysis (DA). The general class of problems in which the current problem of interest is a part is analyzed. Then, the processes required by any system which would operate in the problem domain are identified. Finally, the system is related to the inputs, outputs, and processes pertinent to the system. Section 3.3 provides an example of this process.

Davis describes eight methods of analysis (Davis, 1990:71-100). Each of the methods uses a combination from the notations described in this section or possibly some method specific notations. The eight methods Davis described are:

1. Listing All Inputs and Outputs.
2. Listing Major Functions.
3. Structured Requirements Definition (SRD).
4. Structured Analysis and Design Technique (SADT).
5. Structured Analysis and System Specification (SASS).
6. Modern Structured Analysis (MSA).

7. Problem Statement Language / Problem Statement Analyzer (PSL/PSA)^{TM}.

8. Object-Oriented Problem Analysis (OOA).

3.2 Method Chosen

Domain Analysis (DA) and Modern Structured Analysis (MSA), as defined by Ward and Mellor, were chosen for this thesis effort. They provide a logical and straightforward process for performing system analysis.

Domain analysis helps the analyst understand the class of problems at hand by describing the parts of a system. MSA uses the parts of the system defined in DA to solve a particular problem -- in the case of this thesis it solves the problem of translating standard VHDL into a subset.

Structured Analysis has been around for a long time. Modern structured analysis has evolved from a great wealth of experience and ideas. Davis provides a summary of two methodologies, that of Ward and Mellor and that of Yourdon (Davis, 1990:91).

Briefly, Ward and Mellor's method consists of four main activities. The first step is to identify all terminators to the system, the system, and all data flows; in other words, define the system context. The second step is to create a narrative event list which defines all external events. Third, the behavior of each event is captured as a single-bubble DFD. Finally, the fourth step is to

successively group (combine) the disjoint DFDs created in step three into more abstract models.

Ward's advice for grouping DFDs consists of minimizing interfaces, identifying control hierarchies, grouping common responses, grouping processes based on their sharing of data, grouping of common terminators, and grouping so abstract groups logically have different names. Yourdon suggests to simply group diagrams sharing common data and group diagrams which can hide information in the system.

Yourdon's method of structured analysis is to define a system as two models: behavioral and environmental; these then define the essential model of the system. The models use DFDs, DDs, ERDs, and process specifications. The environmental model consists of a statement of purpose for the system, a context diagram that shows the system and objects external to it and any interfaces, and a list of the events of the system. The behavioral model consists of a complete set of DFDs by Ward's method, entity-relationship diagrams of all objects in the system and those with which the system interacts, state transition diagrams for the control process behavior, a data dictionary, and process specifications for the lowest level processes.

Modern structured analysis does not require the use of automated programs which are required by some of the other methods and which may not be readily available. An automated tool to create DFDs and DDs and ensure their consistency

would be desirable, but any graphics package can suffice for creating modern structured analysis drawings and any text editor or word processor will suffice for creating related documentation.

Another benefit of modern structured analysis is how the results consist of well-defined functional entities. These functional entities then form the foundation for the functional design of a system.

Obviously, from the discussion thus far, the approach to the development of Style-V is functional decomposition. The original observation guiding this decision was that few objects exist in the problem space -- mainly only input files, the Style-V translator, and output files. In Section 3.3 the parts of a translator are defined. Most of the real work (translation tasks) of the translator is done by one of the parts. Therefore, since the largest part of the translator would be functionally defined, even in an object-oriented approach, a functional approach to analysis was deemed best.

3.3 Domain Analysis of Style-V

The analysis of Style-V is unlike many of the analysis techniques described by Davis and reviewed in Section 3.2. Most of the methods, including MSA, start by defining how an organization is currently doing a task and then analyze the task performance of parts of the organization. With

Style-V, no organization is currently doing the translation work of Style-V; therefore, some other method was required to provide a starting point for MSA. Domain analysis provides a way to gain a broad understanding of a problem and a place to start the analysis process.

The translators reviewed in Chapter 2 provide a basis for determining the generic components of a translator. By analyzing how those translators work, it was determined that a translator is made up of three general processes -- lexical, syntactic, and semantic. These processes operate on an external input to produce an external output (input and output files).

Lexical analysis is the process of determining the tokens of an input file to pass to the syntactic or semantic processes. Generally for programming languages, the tokens are words, delimiters, and operators. In some cases, the tokens may be entire strings (such as comments) if this level of intelligence is coded into the lexical analyzer; otherwise, syntactic and semantic routines process complex tokens.

Syntactic analysis determines if a sequence of tokens is grammatically correct. If a sequence of tokens is not allowed by the rules of the source language, a correct translation is highly unlikely and an error should be generated. Some translators do not include a syntactic analyzer

(or contain a very simple one) because the input is assumed to be syntactically correct. This is the approach of Style-V.

Style-V is not meant to replace a VHDL analyzer. The input files to Style-V are assumed to be syntactically correct VHDL, since translating an incorrect (untried) VHDL description to a tool such as JRS' IDAS does not make sense. It is easy to see the old computer science adage "garbage in, garbage out" applies in this case.

Semantic processing analyzes the sequence of tokens of a language to determine the meaning of the input. Meaning can be on a statement-by-statement basis for some translation tasks. However, for other translation tasks, a more global understanding of a set of statements is required for a translator to be able to generate a correct translation. The semantic process does most of the translation work.

Figure 3-1 provides the highest level pictorial view of the concept of a translator system. It has one bubble representing the translator system, one storage symbol for input files, and one storage symbol for output files. Figure 3-2 is a decomposition of the translator system into its constituent parts: the lexical analyzer, the syntactic analyzer, and the semantic analyzer. Figures 3-3, 3-4, and 3-5 are the conceptual descriptions of the three processes of a translator system. Now that the domain is defined, the next step is to analyze the problem into its constituent

parts. Then, the functions necessary to implement the subpart can be defined and assigned to one of the translator processes.

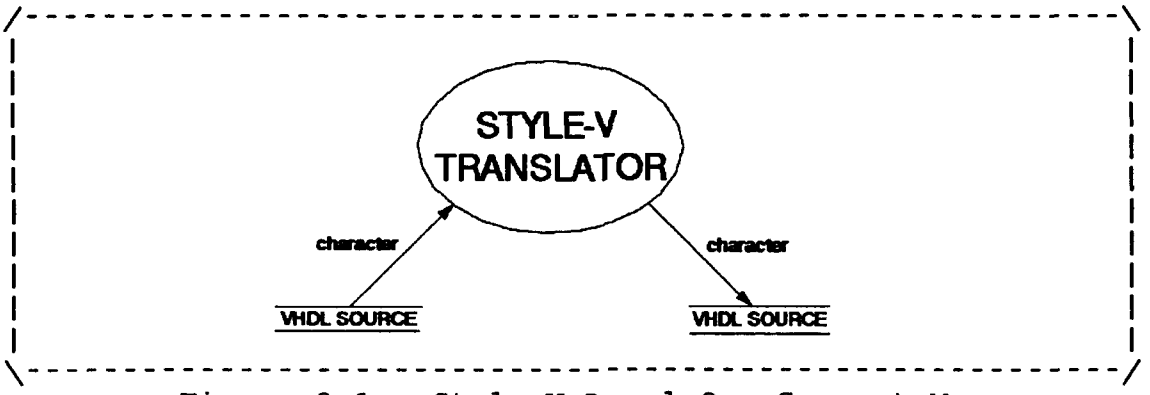


Figure 3-1. Style-V Level One Concept Map

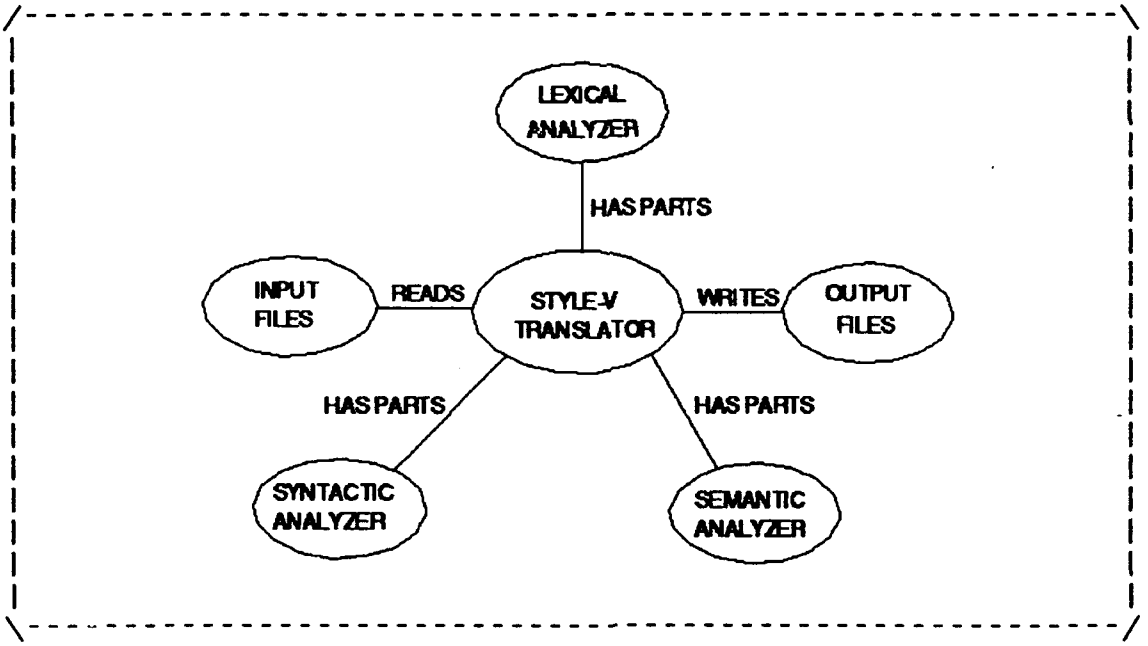


Figure 3-2. Decomposition of Translator Concept

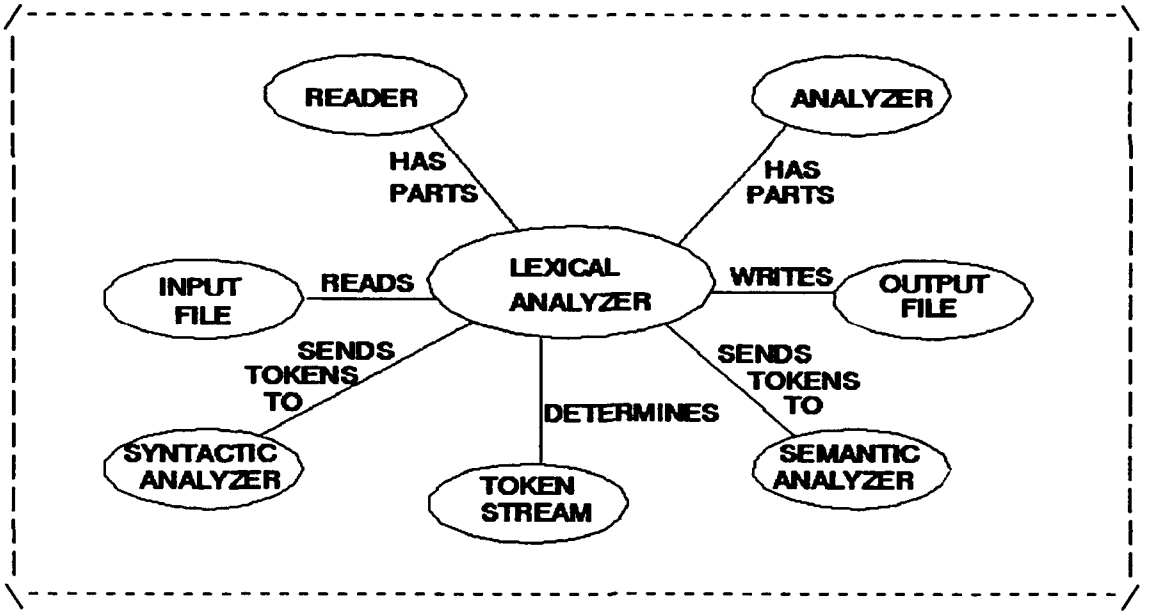


Figure 3-3. Lexical Analyzer Concept Map

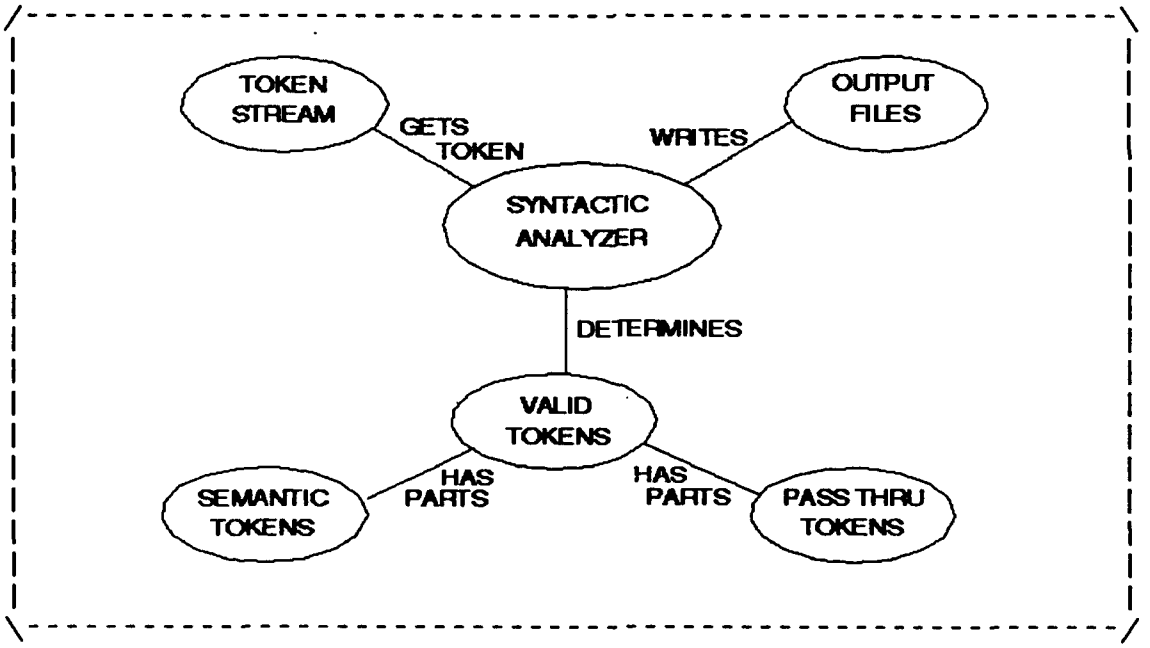


Figure 3-4. Syntactic Analyzer Concept Map

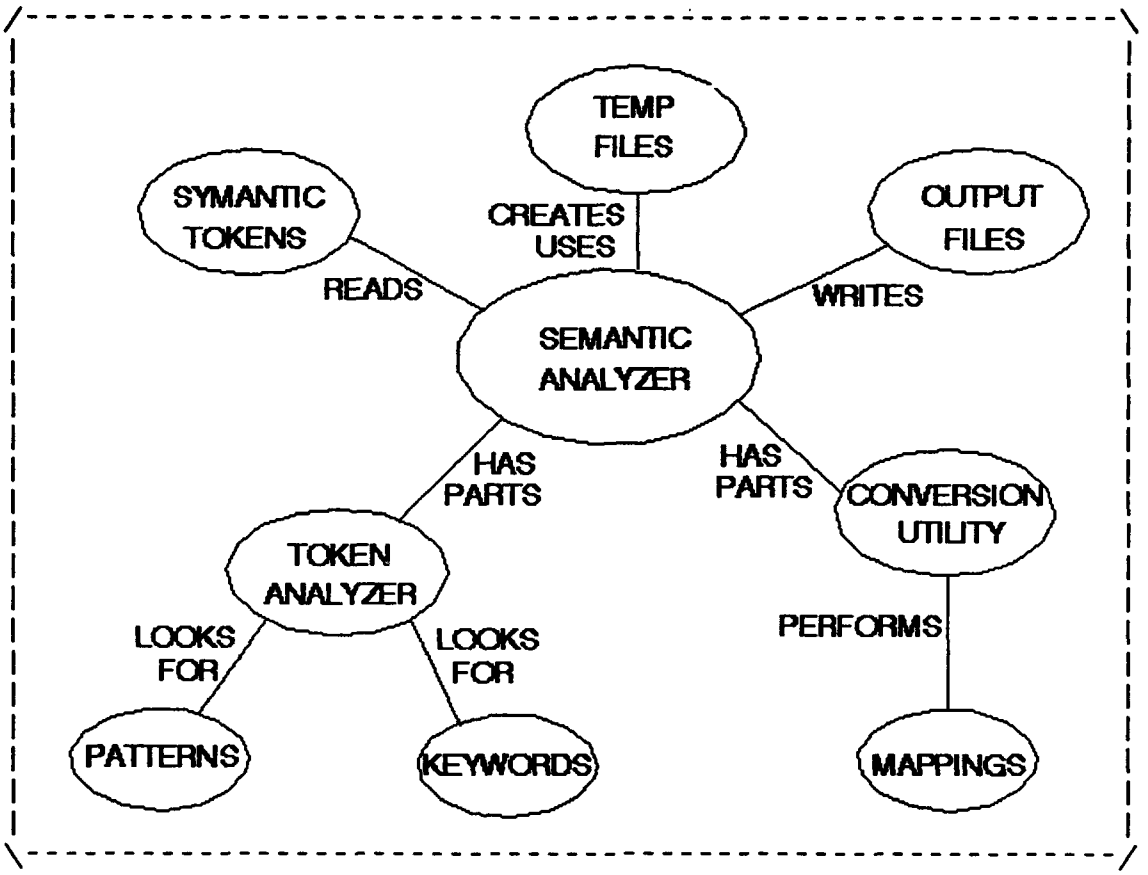


Figure 3-5. Semantic Analyzer Concept Map

Domain analysis has shown that lexical, syntactic, and semantic analysis are essential parts of a translation system. However, since the input to Style-V is syntactically correct VHDL, Style-V need not worry about checking syntax. The other functions, lexical and semantic analysis, are the basis for how Style-V will perform the translation task. Any major function of Style-V that reads input and produces output must perform lexical and semantic processing. The way Style-V will do this is described in Section 3.5. However, before a description of how the translation will be done is possible, an analysis of the problem to

determine what must be translated is required, and this is the subject of Section 3.4.

3.4 Problem Analysis

A fundamental truth about program translation is that a syntactically and semantically correct input must be translated into a syntactically and semantically correct output. Since translation of standard VHDL into the subset defined for the JRS IDAS tool has not been done, a careful definition of the differences between the subset and the standard was imperative. Without knowing what is in the standard which is not allowed in the subset, creating a translator to consistently produce an output which contains only constructs allowed in the subset is impossible.

Section 3.4.1 provides a discussion of the differences between standard VHDL and the JRS styled subset including a possible mapping from the standard to the subset. Section 3.4.2 describes an example system coded in JRS styled VHDL to gain an understanding of the requirements and limitations of the subset. Section 3.4.3 is a discussion of the lessons learned from the example system. Finally, Section 3.4.4 is a discussion of the limitations imposed by using the subset of VHDL defined by the JRS style.

3.4.1 Stylized and Standard VHDL Compared. The only document which provides insight into what is allowed for the JRS style of VHDL required for IDAS is the VHDL Style Guide

for Ada to Microcode Compiler Retargeting and VHDL Simulation (VHDL Style, 1989). For each restricted construct, a mapping from the standard to some construct or group of constructs in the subset must be possible before translation is possible. The remainder of this section describes the restrictions imposed by JRS on VHDL written for the IDAS and one or more possible mappings from standard VHDL. Figure 3-6 provides an abbreviated description of the mappings any successful translator must address.

3.4.1.1 Type Differences. A translator for standard VHDL to a JRS style VHDL must address several type conversion issues. The restrictions imposed by the JRS style are quite severe. A careful review of the JRS Style Guide resulted in documentation of the restrictions and found that types in JRS styled VHDL are limited to the following (VHDL Style, 1989:4):

a. BIT -- a binary 0 or 1,

b. BIT_VECTOR -- a one-dimensional array of BIT,

c. ARRAY of BIT_VECTOR -- a one-dimensional array of BIT_VECTOR,

d. STATUS_TYPE -- a special enumerated type.

Since standard VHDL allows user defined types, such as enumerated types to model multilevel logic, some mapping from multilevel logic to the types BIT and BIT_VECTOR was required.

This mapping was originally thought to be a trivial matter of assigning the value '0' or '1' to each of the enumerated type fields to derive a type compatible with the type BIT. However, a careful study of a real-world design, the Floating Point Application Specific Processor (FPASP), which uses four-level logic showed that the arbitrary

Standard or User Types	==>	BIT Types
CASE Statements	==>	IF Statements
Machine Specific Declarations Scattered	==>	Machine Specific Declarations Packaged
Modes Buffer and Linkage	==>	Mode INOUT
Allowed Types in Generics	==>	INTEGER, TIME, FLOAT, or STRING for Generics
Multiple Concurrent Statement Architectures	==>	Single PROCESS Statement Architectures
Guards and Sensitivity Lists	==>	WAIT Statements
Tests for Data Values	==>	No Tests for Data Values
Signal Assign Delay in Structural Architectures	==>	No Signal Assign Delay in Structural Arch.
WAIT as Signal Delay Method	==>	Signal Assignment with "AFTER" Clause for Delay
User Defined Procedures	==>	IDAS Provided Procedures
Variables or Signals for Tracking Status	==>	Special Status_Type for Tracking Status
Component Names and Ports Can Differ w/ Entity	==>	Component Names and Ports Must Match Entity

Figure 3-6. Standard VHDL Mapped to JRS Style VHDL

assigning of the value '1' or '0' to the levels 'X' or 'Z' would result in incorrect behavior of the procedures defined for the design. Consider the segment of code from the FPASP design in Figure 3-7. This code segment looked for illegal values in the input early in the procedure and if found, exited the procedure without further processing. A whole-sale conversion of 'Z' and 'X' to either '0' or '1' would have caused this procedure to never execute the correct code -- it would always exit at the example FOR loop.

```
for I in A_COPY'RANGE loop
     if (A_COPY(I) = 'Z') or (A_COPY(I) = 'X') then
          return (1 ns);
     end if;
end loop;
```

Figure 3-7. FPASP Four-Level Logic Sample

Another considered option was the conversion of the bits of the four-level logic into two-bit bit_vectors (one-dimensional arrays) such that '0' = "00", '1' = "01", 'X' = "10", and 'Z' = "11". Then a four-level logic bit vector such as "01XZ" would appear as " "00" "01" "10" "11" " -- an array of bit_vectors (a two-dimensional array of bit). Unfortunately, the JRS MOVE procedures would only accept the type DATA and DATA_VECTOR which are subtypes of BIT and

BIT_VECTOR respectively (one-dimensional arrays) (VHDL Style, 1989:29,50). The JRS MOVE procedure was critical to the simulation of data moving through a design; therefore, finding a data representation using JRS types and compatible with the JRS MOVE procedure was imperative.

Given the JRS MOVE restriction, the answer was to convert multilevel logic into multiple-bit single-level logic. Then, JRS' MOVE procedure could move data around a design. When it came time to process the data through another JRS procedure, the data could be converted to BIT_VECTOR logic (note: since only BIT logic is allowed for JRS procedures, processing JRS procedures other than MOVE requires conversion of the multibit four-level logic to single-bit two-level logic, thus forcing 'X' and 'Z' to either '0' or '1'). Upon exiting JRS procedures, the BIT_VECTOR could be again converted to multibit four-level logic where each bit is now "00" or "01". A manual review of this method showed it practicable and a hand translation of the extensive FPASP type definitions demonstrated its feasibility.

A second type restrictions imposed by JRS was that status of the model would be tracked using a special enumerated type called a STATUS_TYPE (VHDL Style, 1989:18-20). This is a device independent representation of status values. JRS IDAS provided functions to map between this STATUS_TYPE and a STATUS_VECTOR which is a machine dependent

bit_vector used to track the status of a certain machine. The problem is that most status tracking in real machine designs is to use either bits or bit_vectors (in the logic scheme of the design) to pass status values.

Therefore, a method was required to convert status represented as bit or bit_vector to STATUS_TYPE required by JRS IDAS procedures and back again to bit or bit_vector for movement through the design. Fortunately, the JRS IDAS provision of the MAP_STATUS function to convert between STATUS_TYPE and STATUS_VECTOR (which is a bit_vector representation) provides the first step in solving this problem (VHDL Style, 1989:18-20). It is then a simple matter to map a field of the STATUS_VECTOR to a value for a bit, bit_vector variable, or signal representing status in a design. Using MAP_STATUS, the other procedures for processing types, and a simple procedure to convert STATUS_VECTOR to a variable or signal solves the status mapping problem.

However, a far greater problem is how to determine if a variable, port, or signal represents a field dedicated to tracking status. The analysis of this problem did not reveal any automated way to accomplish this task. Using standard VHDL, a designer can name variables, ports, and signals with any legal identifier for the language and no standard convention exists (for example, signal GEZRO : DESIGN_BIT; -- a status of greater than or equal to zero). Given this fact, human intervention will be necessary to

determine which variables, ports, and signals represent status in a design.

A third restriction on types has to do with the modes allowed. The JRS style of VHDL does not allow the modes BUFFER and LINKAGE (VHDL Style, 1989:8). After a careful review of the modes BUFFER and LINKAGE in the VHDL Language Reference Manual (IEEE Standard VHDL, 1988:4.10), the analysis determined that the mode INOUT can be substituted for modes BUFFER or LINKAGE. This may seem incorrect at first since the mode INOUT allows more operations than the modes BUFFER or LINKAGE. However, remember that the translation is for syntactically and semantically correct VHDL input, since Style-V expects an input of a design that works through VHDL the way the designer intended. Therefore, conversion to a more general mode for the purposes of input to a special tool (in this case, JRS IDAS) should not cause problems.

A fourth restriction on types is the limited number of types allowed in GENERIC statements. JRS IDAS limits the variables of generics to INTEGER, TIME, STRING, or FLOAT types (VHDL Style, 1989:9). This disallows user defined types or other VHDL defined types such as POSITIVE. The translator must convert more restrictive VHDL types to a more general, allowed type. One example would be to convert any generic variable with a type POSITIVE to the type INTEGER. Another consideration is user defined types. If a

base type of the user defined type is one of the allowed types or can be converted to one as explained above, change the user-defined type tc that base type. Otherwise, if the translator cannot convert a disallowed type to an allowed type, manual intervention is required and redesign may be necessary. An example of a case requiring manual intervention is a type resolving to an enumerated type not allowed by IDAS styled VHDL.

3.4.1.2 Declaration Differences. The length of time required for one instruction fetch cycle must be defined as a constant in the package Machine_Declarations with the name CYCLE_LENGTH (VHDL Style, 1989:6). This "clock cycle" is normally defined in the test bench for a design or through a generic in the highest level architecture under the test bench. The main issue is how to detect it for definition in Machine_Declarations -- no standard naming convention exists for the clock period or cycle time. Again, manual intervention is necessary to identify this value, unless only one variable of type CLOCK is defined.

Another difference between standard VHDL and the JRS IDAS styled VHDL is the definition of arrays of BIT_VECTOR which are machine specific. These machine specific declarations must be located in package MACHINE_DECLARATIONS (VHDL Style, 1989:6). Therefore, any declaration in any file of the design which has a type that does not map to the type definitions in the JRS IDAS provided package DATA_TYPES must

be moved to package MACHINE_DECLARATIONS. This means package MACHINE_DECLARATIONS must be included in a USE clause in all files with packages, entities, or architectures which use a type defined in MACHINE_DECLARATIONS.

Additionally, all declarations of bus subtypes and their associated resolution functions must be defined in package MACHINE_DECLARATIONS (VHDL Style, 1989:6). These must be moved to MACHINE_DECLARATIONS in the same manner as described for moving machine specific data type definitions.

A fourth problem is that names used in components must match the design entity for which they are defined. Additionally, component ports must be exactly the same as the design entity ports (VHDL Style, 1989:22,24). In standard VHDL, neither of these restrictions apply. To stylize component ports, it will be necessary to maintain a table of entity declarations with all port definitions. Then, when processing component declarations, their port list can be compared to the entity port list and adjustments made. Ports of entities not used in components can be added to the components with dummy variables or signals. The actual design should still work if it worked before the translation.

3.4.1.3 Structural Differences. A major difference between JRS IDAS styled VHDL and standard VHDL is the limitation of only one concurrent statement (a process statement) for any architecture (VHDL Style, 1989:11). At

first glance, this appears to be a monumental problem as VHDL allows any combination of block and process statements within an architecture to model parallel activities. Combining all these concurrent statements into one process would destroy the concurrent model. The answer lies in creating several single-process architectures as subarchitectures to a new upper level architecture (with the name of the original multiprocess architecture) which instantiates these new "component" architectures. This process is described in detail in Section 3.5.3.4.

A second structural issue is the prohibition in JRS IDAS styled VHDL for sensitivity lists for a process (VHDL Style, 1989:12). Instead, when a block guard or process sensitivity list is recognized, collect the information and after the process BEGIN, insert a VHDL WAIT statement with an UNTIL clause using the guard condition and an ON clause using the signals of the sensitivity list. This effectively causes the process to wait until the guard condition is true and one of the inputs has changed before the process begins -- just like a sensitivity list. The UNTIL clause will be added to the WAIT statement for each process that was part of the block.

Thirdly, the placement of the wait for a behavioral IF is more restricted in JRS styled VHDL than in standard VHDL. Any delay for the result of the behavior is modeled as an AFTER clause in a signal assignment statement following the

behavioral IF (VHDL Style, 1989:14,16). When a WAIT statement precedes an IF or CASE structure or a signal assignment statement, if it is not the first WAIT of the process, then the time should be saved and placed in an AFTER clause in the appropriate signal assignment statements.

A final structural issue is that a structural architecture statement part may include only component instantiation statements and simple signal assignment statements without delay. If delay is required, the behavior must be modeled by connecting the involved signals to a behavioral design entity with a MOVE that has the required delay (VHDL Style, 1989:24).

3.4.1.4 Statement Differences. Several differences in the statement composition or format exist between JRS IDAS style VHDL and standard VHDL. The first is the fact that the behaviors of JRS styled VHDL are represented in IF statements (VHDL Style, 1989:11,13). Standard VHDL allows the more versatile CASE statement, so a CASE to IF conversion is required. The mapping of CASE to IF statements entails recognizing the CASE statement, identifying the conditional variable, and processing the WHEN clauses into the appropriate clauses of the IF-THEN-ELSIF-ELSE structure.

A second statement difference between standard VHDL and JRS styled VHDL concerns tests for value in the guard of a behavioral IF statement. In JRS styled VHDL, the guard can

only contain tests for phases of the system clock and microword fields, clock tests must come before microword tests, and no test for the value of a data path are allowed (VHDL Style, 1989:14). Since none of these restrictions apply in standard VHDL, whenever an IF is encountered, the guard must be analyzed and reorganized as needed. Also, since no test of a data line is allowed, a startup input defining the name of the control word and all clocks will be required for the translator to be able to ensure no data path test is attempted. If a data path test is attempted, the translator could optionally print a warning message and continue processing, stop for manual intervention, or report an error and terminate processing.

A final statement difference between standard VHDL and JRS styled VHDL is the requirement for procedure parameters of mode OUT or INOUT to be variables (VHDL Style, 1989:15). If mode OUT or INOUT parameters are not variables, the translator must insert variable declarations of the appropriate type in the process containing the procedure and use them in the procedure call. Additionally, the procedure call must be followed by a signal assignment statement for each variable reassigning the value of the variable to the appropriate signal.

3.4.1.5 Other Differences. Several differences between JRS styled VHDL and the standard do not fit into any of the aforementioned categories. One concerns the JRS IDAS

code generator, since it does not recognize or understand behaviors other than those provided with the IDAS (VHDL Style, 1989:18). This means as many procedures as possible of the input design should be converted to JRS IDAS procedures. The implications of this requirement to convert any given function of a design to an "equivalent" function in another library will be covered in Section 3.4.4.

Another issue is the handling of tristate busses. According to the JRS VHDL Style Guide, all devices driving a tristate bus should be controlled by one field of the microword (VHDL Style, 1989:25). However, if the type conversion procedure described in Section 3.4.1.1 works as designed, the procedures of the original design which operated on tristate values could be converted and still operate as before -- the only thing different would be the representation of the logic.

3.4.2 Example Stylized Machine. As the work for this thesis began, a search was made for potential example systems to code in the stylized form. It was hoped that coding an example system in the JRS styled VHDL would give insights into possible translation difficulties.

The first design selected for conversion to JRS IDAS style was a central processing unit (CPU) controller written by Richard L. Miller as part of his AFIT thesis effort (Miller, 1990). After a couple of weeks of trying to determine how the IDAS environment (including the styled VHDL,

the IDAS procedures, and the IDAS system) could model the controller, a careful investigation revealed the CPU controller was modeled as a state machine instead of a "real" machine, and the CPU controller did not represent the minimum system required for the IDAS to generate microcode.

After the CPU controller was ruled out, it was decided the simple CPU described by Hayes would serve as the basis for the IDAS example (Hayes, 1988:315). Hayes' CPU provided all the components necessary for the IDAS to generate microcode. It consists of a control store, instruction register, program counter, address register, memory, data register, accumulator, and arithmetic logic unit.

The first attempt at the Hayes model was to design the entities with ports that connected to all associated entities. This is a direct translation from the diagram (see Figure 3-8) provided by Hayes (Hayes, 1988:315). However, after coding the example in that fashion, it was determined that the direct translation of the diagram was not a realistic implementation.

After careful consideration, the model in Figure 3-8 was perceived to be a logical representation of the system. A more accurate "physical" representation of the system was needed to gain insight into the correct structure for the VHDL code. Figure 3-9 provides this structural view of the simple Hayes model.

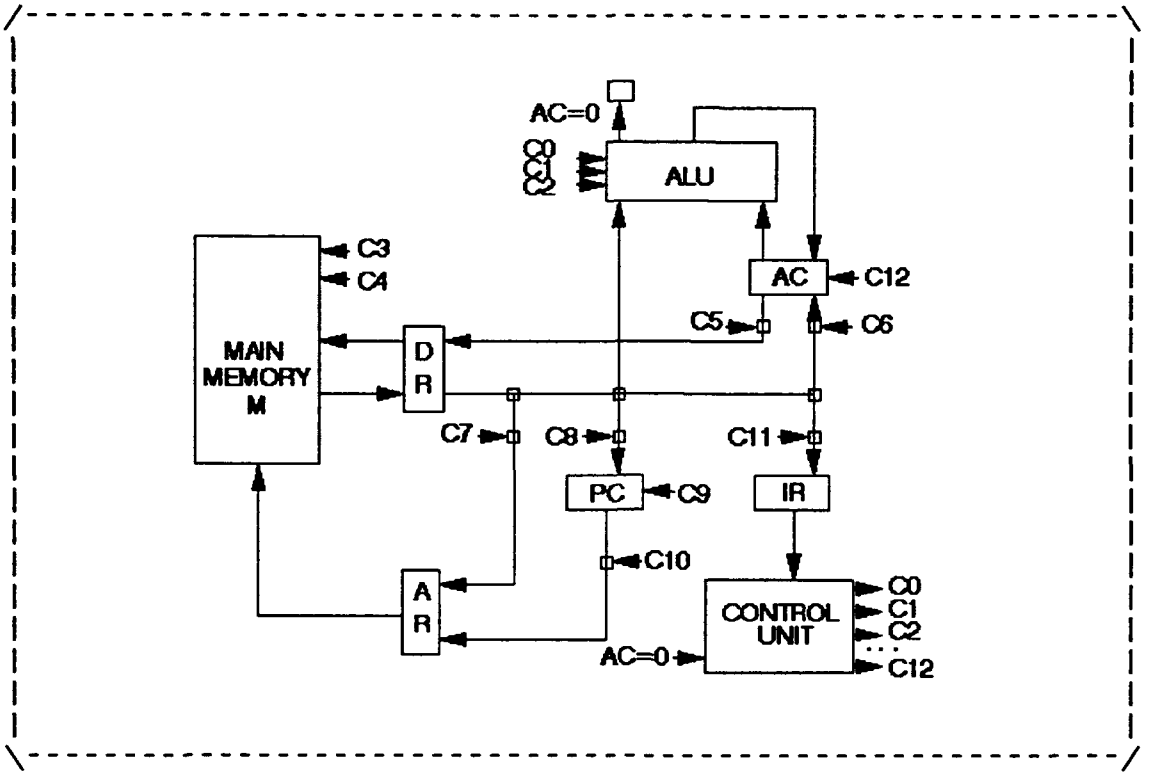


Figure 3-8. Simple CPU as Presented by Hayes (Hayes, 1988:315)

Early versions of the Hayes model did not properly account for circuit timing characteristics. The first version was the idealistic no-delay version. Then, delays were added to the bus and in later versions to the components.

Through version 4, the Hayes model was written using a WAIT statement inside the behavioral IF statements containing IDAS procedure calls. During a review of the VHDL Style Guide, it was discovered that this is not allowed. The delay had to be moved to the signal assignment statements after the procedure call. Hayes simple CPU version 5 is now believed to be totally in the style required by the JRS

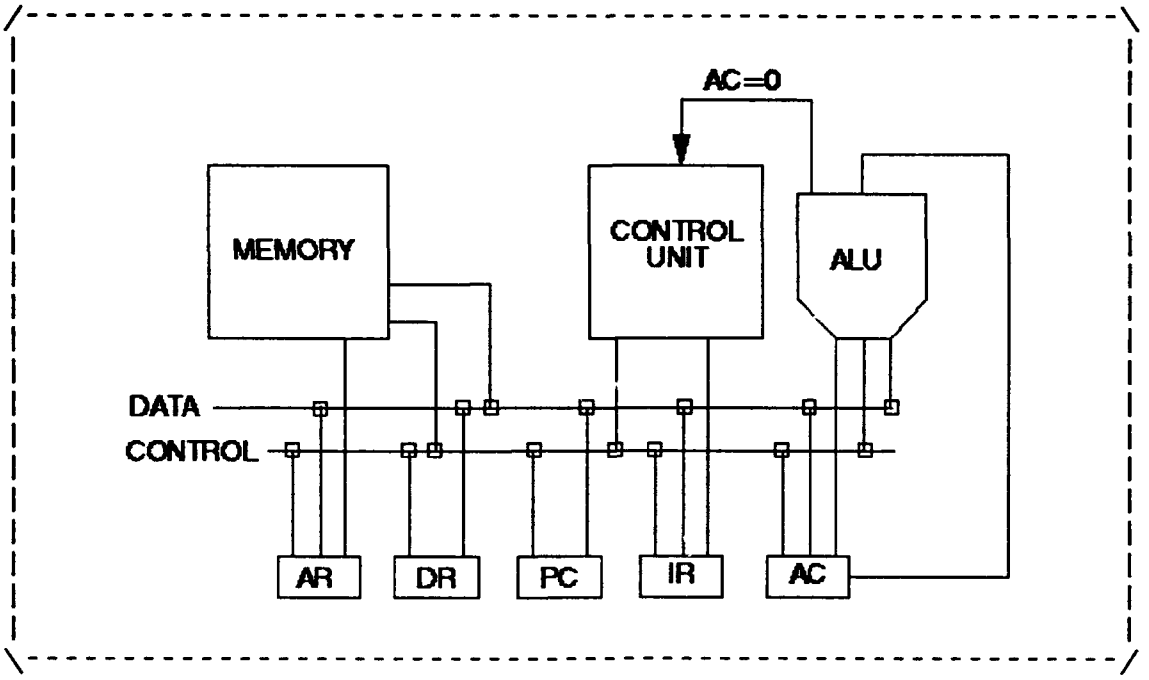


Figure 3-9. Structural View of Hayes Model

IDAS. Appendix A contains the final version of the stylized Hayes CPU model, and Volume II contains a simulation listing which is annotated with the machine instructions being executed each cycle. (Note: Volume II is used to maintain restricted access information and computer listings which could not be formated for inclusion in Volume I).

Unfortunately, when the JRS IDAS was run to enter the Hayes model, it would not take it. The nice graphical interface written to enter VHDL designs, analyze them in the IDAS database, enter Ada code to tune the generated micro-code, and check the VHDL design into the microcode generation process did not work for checking in the Hayes units.

Additionally, a local IDAS expert was contacted (who in turn contacted JRS) and manual procedures were attempted to

load the units in the system through a complicated sequence of maneuvers, but to no avail. The portion of the IDAS required for this thesis did not work. This was a setback for this thesis, for it was now not possible to test the conclusions of the analysis against an actual tool. Yet, the analysis provided valuable insight into the obstacles to translating VHDL into a subset of VHDL.

3.4.3 Lessons Learned From Example. The lessons learned from the Hayes example were probably lessened due to the inability to run the design through the IDAS. Be that as it may, some valuable ideas did come from the Hayes modeling exercise.

It is possible to translate a physical design into a JRS styled VHDL description using the JRS procedures and gain a correct implementation -- an implementation that gives correct results when run through a VHDL simulation. The Hayes example therefore indicates that other designs should work if translated to the JRS style.

During development and implementation of the Hayes CPU, it was easy to use a perfectly allowable construct in VHDL, but then learn it was not allowed by the JRS style. This indicated a need for a detailed analysis and design of each mapping needed to transform standard VHDL into the JRS style. It also highlighted the need for an automated tool to do the translating, since humans make errors when given such a complex task.

Not much was learned regarding the translation of types and functions since the Hayes model was originally built using JRS' VHDL Style Guide. A more realistic system was needed to test mappings for types and functions and to verify the feasibility of mappings required by IDAS which were not exercised during the Hayes implementation. The system chosen for use during the remaining development of Style-V was the Floating-Point Arithmetic Shift Processor (FPASP). The FPASP was complex enough to test most of the mappings required for standard VHDL to the JRS IDAS styled subset.

3.4.4 Stylized Limitations. A great concern when going from a design coded in a general language to a more restricted form is the need to maintain all the functionality of the design. The IDAS will maintain functionality of functions which map to IDAS provided functions. However, since the IDAS allows user-defined functions which it does not understand or use, it is not clear that design functionality is maintained -- apparently, functionality would be lost in the conversion. Also, adding functions to the IDAS is not a user option -- it requires a request to JRS which entails an implementation waiting time, a possibility the request would not be accepted, and most likely, a charge for the service.

An additional concern of function mapping is how to do it. Is it even possible to determine if two sections of

code are functionally equivalent? For an example of the difficulty of this problem, consider the two simple procedures in Figure 3-10.

The procedure George performs an ADD operation on two 32-bit bit_vectors using AND and OR gates. The procedure Brad also performs an ADD on two 32-bit bit_vectors, but Brad uses the more common XOR, AND, and OR combinations to accomplish the ADD. Note that none of the variables of either procedure have a name that matches the other procedure -- just like the flexibility in a real programming language. Without carefully analyzing the logic, it is virtually impossible to map these "functionally equivalent" procedures to one another.

A naive approach might be to test the code segments for a given input or set of inputs, and this might work for a small number (say N) examples. However, an implementation that does this r ks accepting a mapping that is incorrect for input (N + Additionally, as more complexity is encompassed in a design, the number of tests increases dramatically. Consider our simple sample procedures of Figure 3-10.

Given the two 32-bit inputs (each having 2^{32} possible values) and an additional input bit (which has two possible values), the total number of possible inputs is $2^{32} * 2^{32} * 2 = 2^{32+32+1} = 2^{65} \approx 3.689 * 10^{19}$. A computer processing 100 procedure runs per second would take about

11,697,742,263 years to test all possibilities.

Style-V's problem of mapping existing functions is different than the problem of translating a function from one language into another. The translators and methods reviewed in Chapter 2 provide many examples of how a "translation" is done.

Two technologies currently exist to perform translations -- transliteration and refinement or abstraction and reimplementation. Translations by transliteration and refinement are usually line-by-line conversions of functionally well defined statements of one language into functionally well defined statements of a target language. Translations by abstraction and reimplementation involve mapping the logic of an input program into formally defined layers of abstraction and then by mapping the formally defined abstraction into a program in the target language.

Unfortunately, Style-V needs to map nonatomic functions (multiple statements with unique logic) for which meaning is not known to specific nonatomic functions for which meaning is known. Neither transliteration and refinement nor abstraction and reimplementation can solve this problem. Since it is realistically impossible to ensure a **completely correct** mapping automatically, Style-V requires human intervention to perform this task.

```
Procedure George (OP1, OP2 : in BIT_VECTOR;
        CARRY_IN : in BIT;
        SUM      : out BIT_VECTOR(31 downto 0);
        CARRY_OUT : out BIT);
variable LCL_CRY,   : BIT;
variable i          : INTEGER := 0;
 Begin  -- George
     LCL_CRY := CARRY_IN;
     for i in 0 to 31 loop
          SUM(i) := OP1(i) and OP2(i);
          SUM(i) := SUM(i) and LCL_CRY;
          if ( ((OP1(i) = '1') and (OP2(i) = '1'))
             or ((OP1(i) = '1') and (LCL_CRY = '1'))
             or ((OP2(i) = '1') and (LCL_CRY = '1'))
             )
               then LCL_CRY := '1';
               else LCL_CRY := '0';
          end if;
     end loop;
     CARRY_OUT := LCL_CRY;
End George;
```

```
Procedure Brad (IN1, IN2 : BIT_VECTOR (31 downto 0);
      IN3       : bit;
      OUT1      : out Bit;
      OUT2      : out BIT_VECTOR (31 downto 0) );
variable LC, LR, Z1, Z2 : BIT;
variable J : INTEGER;
begin
     LC := IN3;
     for J in 0 to 31 loop;
          LR := IN1(J) xor IN2(J);
          Z1 := IN1(J) and IN2(J);
          OUT2 := LR xor LC;
          Z2 := LR and LC;
          LC := Z1 or Z2;
     end loop;
     OUT1 := LC;
end Brad;
```

Figure 3-10. Two "Functionally Equivalent" Procedures

A later version of Style-V might consider using a "partial testing" method to get a "first cut" mapping which

the user could run against representative input to "verify" correctness. This would entail a fairly sophisticated mechanism for taking VHDL procedures as input, determining input for the procedures, processing input through them, comparing the output to output generated by IDAS procedures with the same number of input and output parameters, and determining if a match occurred. If the automatic mappings proved to be incorrect, the user could rerun the translator with a switch set turning the automatic mapping function off. This process alone might be a good thesis topic for some future thesis effort.

Another limitation which may not allow the complete functionality of an original design to be maintained in a translation is the limited typing of JRS styled VHDL. Apparently, the mapping of machine specific words from multilevel logic to bit logic is possible and functionality can be preserved (See Section 3.4.1.1). However, the mapping of types for GENERIC statements and the limited modes available for types may limit functionality of a translated design. Since translation to JRS IDAS styled VHDL requires some types to be mapped to a more general type, it is not clear if the same functionality will be maintained in all cases, though it is expected the new design will suffice for microcode generation purposes.

A limitation which has been solved by this thesis is the modeling of multilevel logic. This can now be accom-

plished using the JRS styled BIT logic (See Section 3.4.1.1). An additional benefit is the limitation on tri-state busses stated in the JRS VHDL Style Guide need not be a concern since multilevel logic can be modeled.

Though not necessarily a functionality concern, the restriction of tracking status using the JRS defined STATUS_TYPE and STATUS_VECTOR is a difficult translation task. This research did not find an automated way to map status variables or signals to JRS procedures and their associated STATUS_TYPE without manual intervention or possibly the need for some type of definition file in addition to the VHDL code of the design.

3.5 Modern Structured Analysis

Modern structured analysis provides a method for achieving a detailed specification of a problem. The specification is the most important part of a development effort, for every action taken after the specification is directed toward fulfilling the requirements documented in it.

Not only does modern structured analysis provide a vehicle for providing well-stated and unambiguous specifications, it also provides that information in a format that facilitates modular, functional design. Every major function is defined with its inputs and outputs during the modern structured analysis process. The first step in this process is the definition of the system context.

3.5.1 Style-V Translator System Context. When building anything, including a software system, the first step during problem analysis is to determine how the problem relates to the world around it. For a translator, and specifically for Style-V, the world represents the inputs it expects and the outputs expected of it. Inputs can come from external entities such as users or data stores, and outputs can go to the same (Davis, 1990:91).

Figure 3-11 provides a level zero DFD (context diagram) showing the system context for Style-V. This figure displays a user entity which inputs commands and responses to the system, the input files coming from a data store containing VHDL source files, status information and system requests going back to the user, and output files going to a data store. At this level of abstraction, this is the only level of detail allowed.

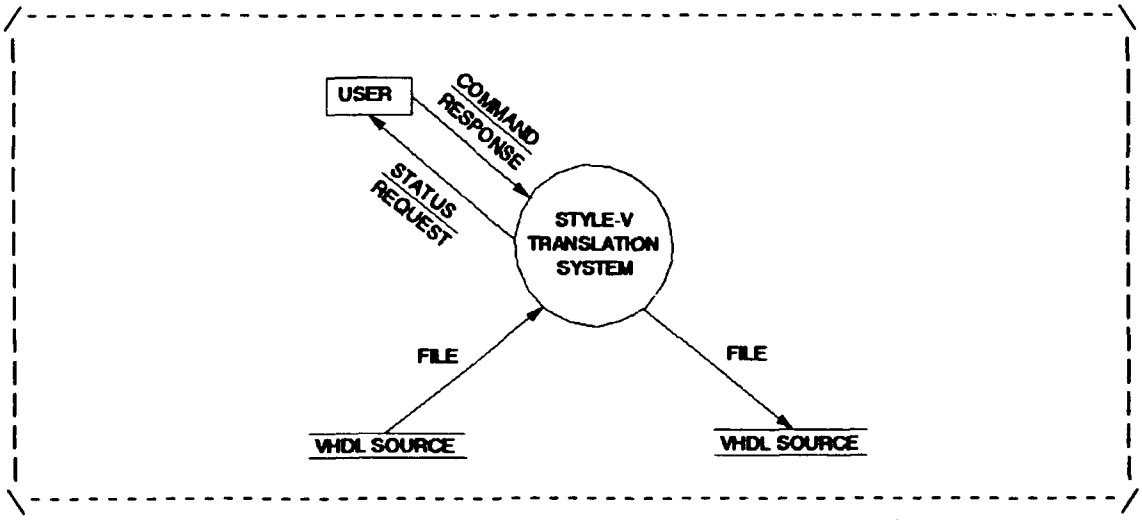


Figure 3-11. Style-V System Context Diagram

3.5.2 Style-V External Events. Once the system context is defined and understood, the next step is to completely specify the external events affecting the system. This specification takes the form of a narrative event list (Davis, 1990:91). The narrative should cover all the actions necessary to translate VHDL files from a standard form into a stylized form. Therefore, the External Event List (EEL) should describe all the external actions necessary to facilitate the mappings described by Section 3.4.1, Figure 3-6.

At first glance, it would seem that many actions would be required to facilitate the thirteen mappings identified in Figure 3-6. However, a careful look at the context diagram shows that the only entities outside the Style-V system that act upon it are the user entity and the input files of VHDL source code.

Only one input from the user would tell the system to partially or completely stylize a file, group of files, or the entire design. Then, Style-V would request the file from the VHDL source store and perform all the internal actions necessary to stylize the file. Style-V would only ask for more input if manual intervention was necessary or if a recoverable error occurred. The user would then input an appropriate response. With this overview, it is then relatively easy to derive the event list for the process. The event list is provided in Figure 3-12.

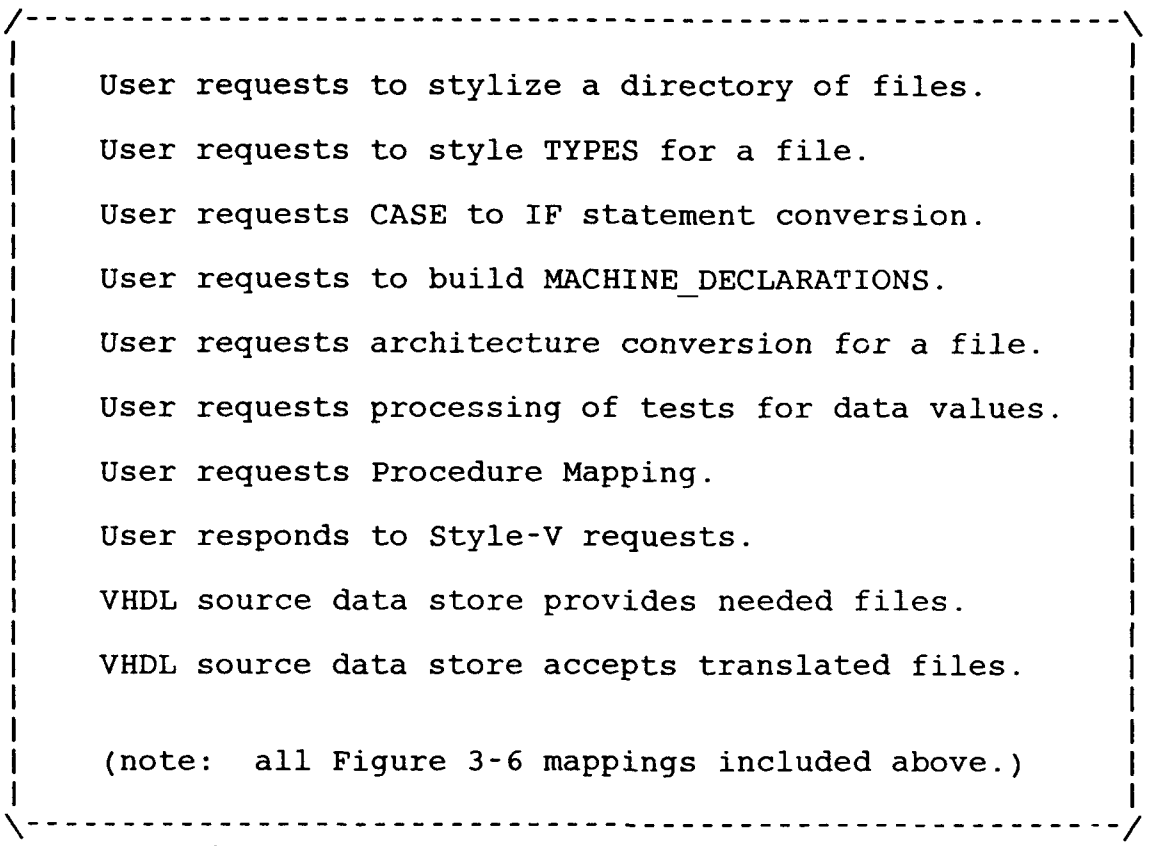


Figure 3-12. Style-V External Event List

3.5.3 Style-V Event Behaviors. When deciding how each external event affects a system, the analyst needs to determine which operations of a system affect other operations within the system. With this knowledge, the analyst can avoid proposing a course of action which will cause a conflict later in the development phase.

For the Style-V translation task, a review of the external events and the expected output of each shows that the actions required to stylize a VHDL file are orthogonal -- each action can be performed independently of the other actions and in any order, and the results will be the same.

Therefore, any of the processes required of the Style-V translator can be done without concern for which other Style-V processes have been done or will yet be done.

Identifying the external events may have been simple, but specifying all the behaviors of the system in response to those external events is not. This section contains an explanation of the actions performed by Style-V in response to external events. This explanation must be detailed enough to facilitate the design phase in Chapter 4.

In Figure 3-12, the various external events were summarized. The first event was a user's request to stylize all files in a directory. This operation entails a combination of all the separate actions identified in Figure 3-12. Therefore, a detailed description of this action will best be given after each of the separate actions is explained.

3.5.3.1 Type Conversion. A major requirement for stylizing a VHDL design is to convert the standard and user-defined types to a type allowed by the JRS VHDL subset. This involves scanning an input file for type declarations, checking those declarations to determine if they are compatible with types allowed in the VHDL subset, and converting types that are not compatible. A first pass over the file converts all types to a subset compatible types and the result is written to an intermediate file. Generic clause types are converted automatically unless no clear mapping exists, and then the user is asked to identify an acceptable

type. All of these conversions are collected in a temporary file. Since it is impossible to determine which types are used for status tracking, the user is then asked to identify which types are used to track status and a second pass over the file converts those types to JRS IDAS Status_Types. A high level model of the type conversion process is provided in Figure 3-13. A more detailed view will be described in Section 3.5.4.

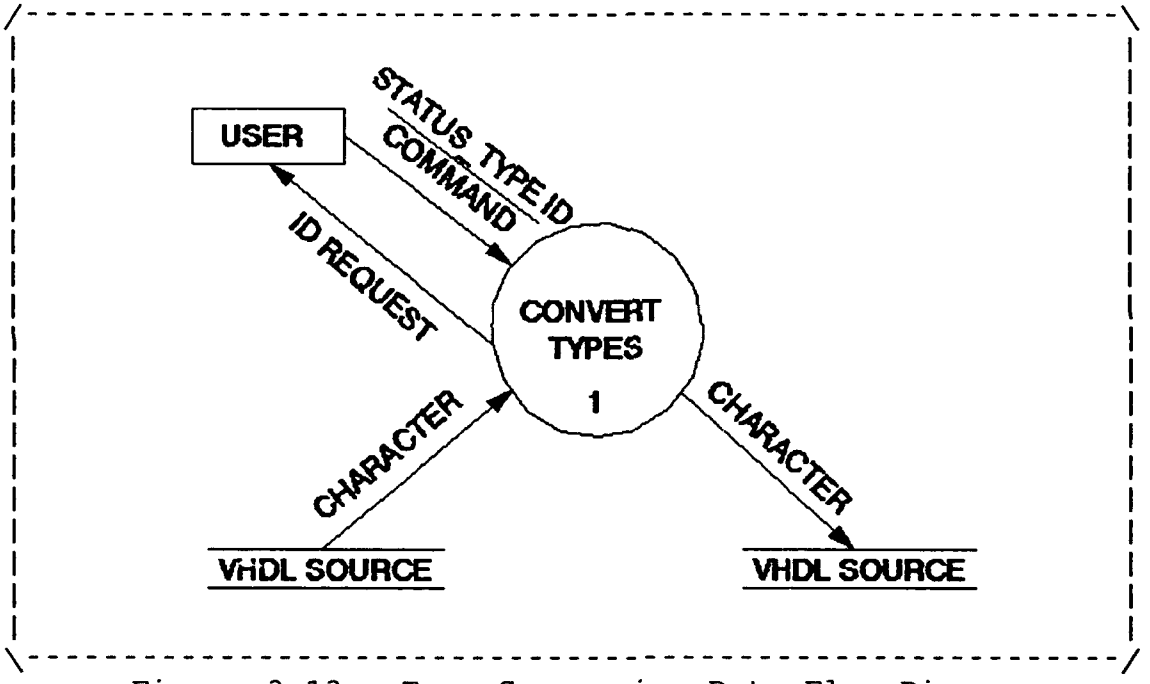


Figure 3-13. Type Conversion Data Flow Diagram

3.5.3.2 CASE to IF Conversion. Since the JRS IDAS style of VHDL does not permit the VHDL CASE statement, Style-V must translate CASE statements to equivalent IF statements. The method chosen to perform this task is transliteration and reimplementation.

The translation process consists of several steps. The first is to read an input file until the keyword CASE is recognized, convert it to IF, and write it to the output file. The second task is to read the conditional variable and store it as a string for use in the IF, ELSIF, and ELSE portions of the IF statement. The input file is read until the value after the first WHEN is read, then the string is written for the conditional variable, the symbol "=", and the value to the output file. The next word is read from the input which should be a "=>" symbol and the word THEN is written to the output file. Now the WHEN clause body should be read and passed through to the output file until the keyword WHEN is recognized again. Then the word after WHEN is read. If the word after WHEN is a value for the conditional variable and not the keyword OTHERS, the phrase ELSIF followed by the variable value is written to the output file. If the keyword OTHERS was found, the phrase ELSE is written to the output file. If an ELSIF phrase was written, when the word "=>" is recognized the word THEN is written to the output file, otherwise it is ignored. The WHEN clause phrase is passed through until the key word phrase END CASE is recognized. This process is represented by a high level description in Figure 3-14. A more detailed view will be described in Section 3.5.4.

3.5.3.3 Build Machine Declarations. The machine specific declarations for a design must be located in a

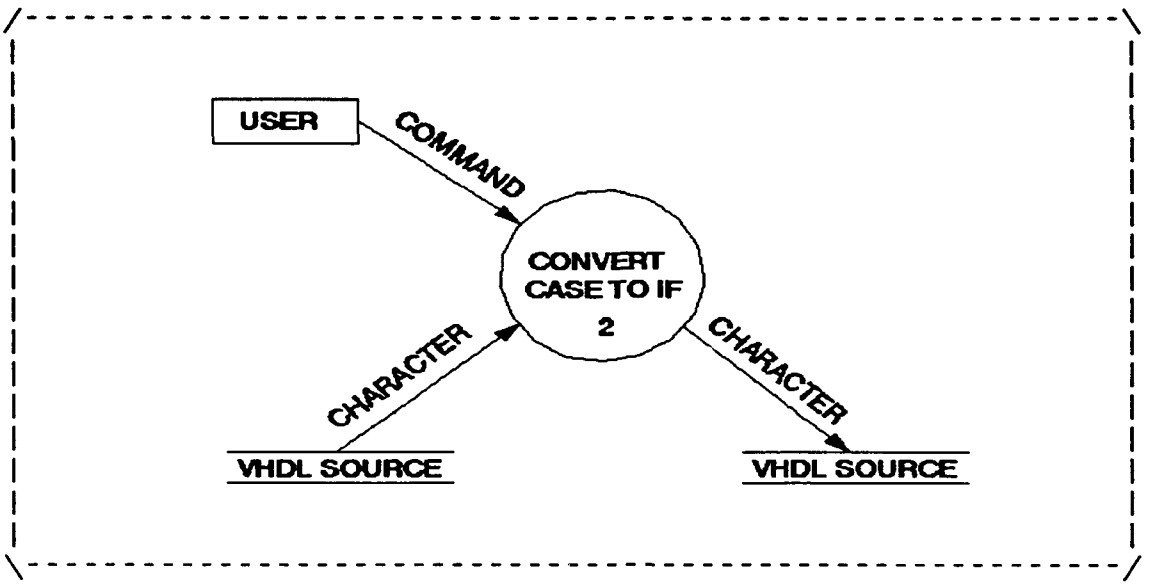


Figure 3-14. CASE to IF Conversion Data Flow Diagram

package called MACHINE_DECLARATIONS for a design specified for the JRS IDAS. To successfully build this package, three types of inf- -mation must be collected and formed into the package. The first piece consists of the CYCLE_LENGTH which is the length of time for one instruction fetch cycle. The second piece consists of the declarations of the machine which use types other than those provided in the JRS Data_Types package. The final piece consists of the declarations representing buses and their corresponding resolution functions.

Since no standard notation exists for defining clock cycles, the CYCLE_LENGTH may be difficult or even impossible to determine from the raw input files. For this data item, a query to the user is the method used by Style-V.

To successfully create MACHINE_DECLARATIONS, the user must provide a list of the file names wherein VHDL source is located. This input could be a list of file names or possibly a directory name if all files in the directory are to be processed.

When processing machine declarations from different files, Style-V must not duplicate the same type name from different files. To facilitate this process, Style-V creates a table of declarations for each file. These tables are then compared to determine if a conflict would exist in the new MACHINE_DECLARATIONS package. Also, as each file is processed, when USE clauses are encountered, the package name for the package containing the USE clause is appended to a list representing the package identified in the USE clause. By tracking which packages use a package wherein declarations are defined, if name changes due to conflicts are required, the name changes can be properly propagated.

Style-V recognizes bus declarations by determining if a resolution function is part of the declaration. This is relatively easy to determine since the input is validated VHDL code. Therefore, if a declaration has a multiword type definition, it will be a bus type. The name of the resolution function will be extracted and used later when Style-V collects all resolution functions in MACHINE_DECLARATIONS.

The processing necessary to create MACHINE_DECLARATIONS requires multiple passes over the files. A final pass

removes all declarations from the files that were written to package MACHINE_DECLARATIONS. The high level view of the process described above is pictorially described in Figure 3-15. A more detailed view will be described in Section 3.5.4.

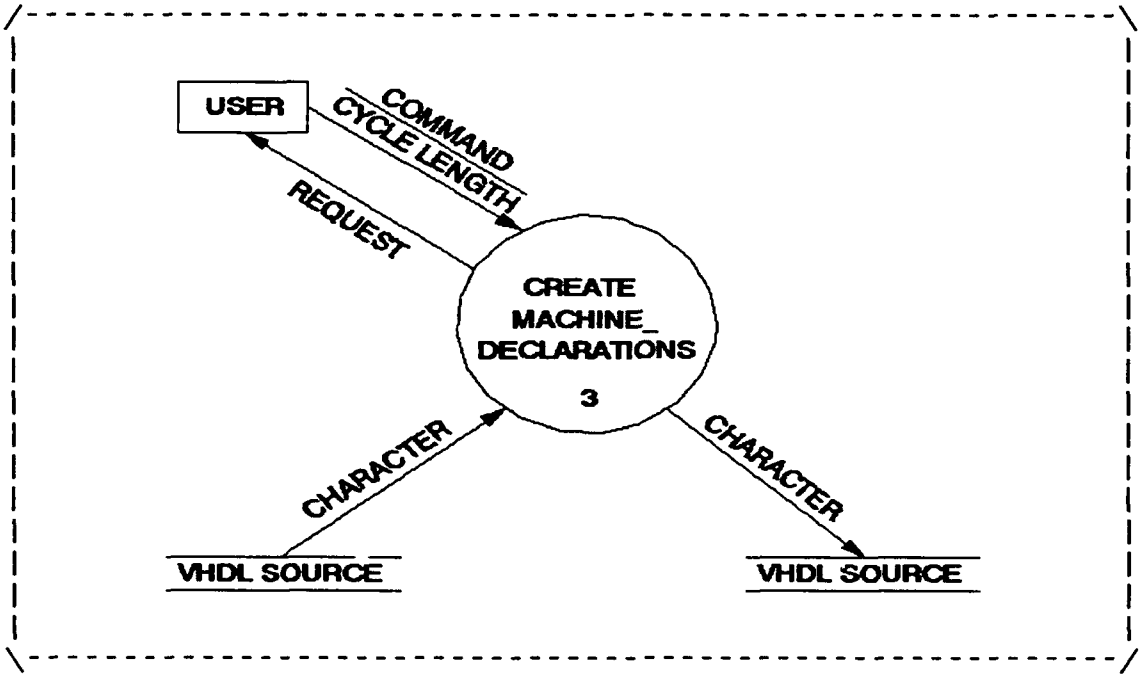


Figure 3-15. Create MACHINE_DECLARATIONS DFD

3.5.3.4 Architecture Conversion. Since JRS IDAS style VHDL does not allow multiple concurrent statements in an architecture, architectures containing BLOCK statements or multiple process statements must be converted into architectures with only one process statement. The overall description of the requirement to convert standard VHDL architectures into JRS single process architectures was given in Section 3.4.1.3. When requesting architecture

conversion, the user must specify which file or files requires conversion. The user can specify this by a file list or a directory if all files in the directory are to be processed.

For each file of the system, five steps are required to convert multiprocess architectures (possibly including block statements) into single-process architectures with no block statements. The first step is to collect statements which are not part of any block or process in the architecture and put them in the statement part of a new high level architecture which will encompass all the architectures formed by the next three steps. The second step is to convert block statements into process statements. The third step is to create an entity for each of the processes formed from the original architecture in step 2. The fourth step is to create an architecture for each of the entities created in step 3. Finally, the fifth step is to form the architectures produced for each process into a higher level architecture with the same name as the original architecture. In this way, only the internal composition has changed. The functionality and external view of the architecture remains the same. The remainder of this section discusses the activities of each step. A real world example using the FPASP design is provided in Chapter 4.

The activity of step 1 (collecting statements which are not part of any PROCESS or BLOCK statement) is a simple

matter of collecting statements which occur outside any BLOCK or PROCESS statement while parsing the architecture. The new architecture will eventually consist of the original entity description, an architecture which defines and instantiates components for each of the subarchitectures created in steps 2 through 4, and a statement part containing the statements collected which did not belong to any block or process.

The activity of step 2 (turning a BLOCK statement into a PROCESS statement) entails determining and storing the guard of the BLOCK statement. The statement starting the block is deleted. As the next lines are parsed, declarations before the keyword BEGIN are found. If declarations are found, they are kept in storage for future use. When the keyword BEGIN is reached (the block's begin), it is deleted. All statements are stored until a PROCESS statement is found. Then, as each process is encountered, the contents of any sensitivity list are stored. The keyword PROCESS is maintained, but the sensitivity list is deleted. Next, any declarations stored from the block is copied into the process. Then, the declarations are parsed until the process BEGIN is found. Next, a WAIT statement with an UNTIL clause for the stored block guard and an ON clause for the stored sensitivity list is inserted as the first statement of the process. Next, any statements copied from the BLOCK statement are placed into the PROCESS statement after

the wait. This process has not only "moved" the block to the process, but has stylized the process entry. Now, the process is parsed passing through all information until the END PROCESS phrase has been passed through. These actions are repeated for any additional processes (using their sensitivity lists). When the END BLOCK phrase is found, it is deleted.

The activity of step 3 (creating an entity for each PROCESS statement) entails identifying the ports and aliases for port portions used in the process, identifying any generic clause items referenced in the process, and forming the entity description for the process based on the collected information. To properly perform this task, the generics, ports, and aliases identified in the original entity and architecture must be known. These can be collected in lists during the initial parsing of the original entity and architecture. Only those generics, ports, and aliases used in the process need be included in the new entity for the new process architecture. Later, when the subarchitectures are combined under a new architecture with the original name, the entity generics and ports will map to the appropriate ports or port portions of the original entity (which stays intact during this entire transformation process).

The activity of step 4 (creating an architecture for each new process statement and entity) is one of the easier steps of the transformation process. The architecture

declaration is of the form "architecture BEHAVIOR of ENTITY-NAME is," and it is followed by BEGIN which is followed by the process and the architecture is completed with an "end BEHAVIOR" clause after the process statement.

The activity of step 5 (creating a new higher level architecture) consists of defining and instantiating components of the new subarchitectures and connecting them to the appropriate ports of the original entity using newly defined signals. Since the JRS style of VHDL required components to have the same name and the same ports as the entity for which they are defined, this is a relatively easy task. Entity AN_ITEM with ports IN1_PORT, IN2_PORT, and OUT1_PORT would be represented by component AN_ITEM with ports IN1_PORT, IN2_PORT, and OUT1_PORT. The component instance will have a unique instance name and use the component name (entity name) and a port map of the component (entity) ports assigned to locally defined signals of the same type as the component ports. The locally defined signals are easy to create since one each is required for each of the original entity ports (name and type come from stored record of original entity ports). After all the components representing the new subarchitectures are instantiated, the new architecture is complete.

A second task to make architectures correct for the JRS IDAS system is to ensure structural architectures have no delays associated with their signal assignments. If delay

is required, a behavioral architecture MOVE statement must be used to model the wait. Version 1 of Style-V will not provide synthesis of behavioral from structural or structural from behavioral architectures; therefore, manual intervention will be required to ascertain if a delay is absolutely necessary. If the delay is needed, the Style-V will return the file to the user for modification.

A third process needed to ensure architectures are stylized is to ensure component names and ports match exactly with the entity defining the components. This was taken care of for the components created for the new architectures. However, other components which do not meet these strict requirements may exist in other parts of the system. Therefore, a process of Style-V will need to check the files of the system for components which do not match their respective entity.

Since one entity may define several components in standard VHDL (all of which may have different names and ports defined), three actions are required to ensure components are properly defined. First, Style-V will make a copy of the entity and its associated architecture with the name of the component. Secondly, the ports of the component will be adjusted to look exactly like the entity, even though not all will be connected to "live" signals when the component is wired into an architecture. Finally, Style-V adjusts component instantiations to match their corresponding

declarations and assigns "dummy" signals to the ports not used by the original component.

Finally, one task remains to ensure architectures are in the style expected by the JRS IDAS. Any signal delay that was modeled with a WAIT statement before a signal assignment statement must be transformed into a signal assignment with an AFTER clause and no preceding WAIT statement. This requires one pass over the files with a small lookahead buffer to see if a WAIT statement precedes a signal assignment statement. Output from the buffer will go directly to the output files.

When the four processes described above are complete, (forming single-process architectures, removing delay from structural architectures, aligning component to entity mapping, and positioning wait in behavioral architectures) the architectures of the design are in the form required by the JRS IDAS. As with the main processes required for the overall stylization of standard VHDL into the JRS form, the subprocesses required to transform architectures into the JRS form are orthogonal. None depend on the results of a previous process and they can be performed in any order.

A high view of architecture processing is presented in Figure 3-16, and a conceptual view of the four main processes necessary to complete architecture processing is presented in Figure 3-17. More detail will be provided in Section 3.5.4.

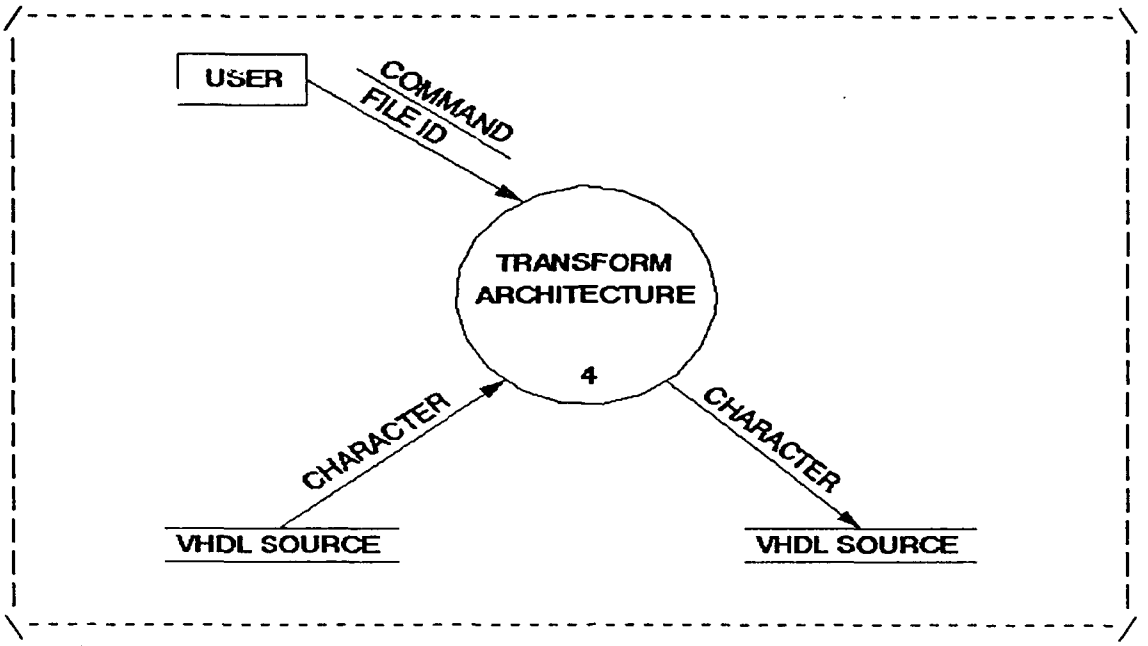


Figure 3-16. Architecture Conversion Data Flow Diagram

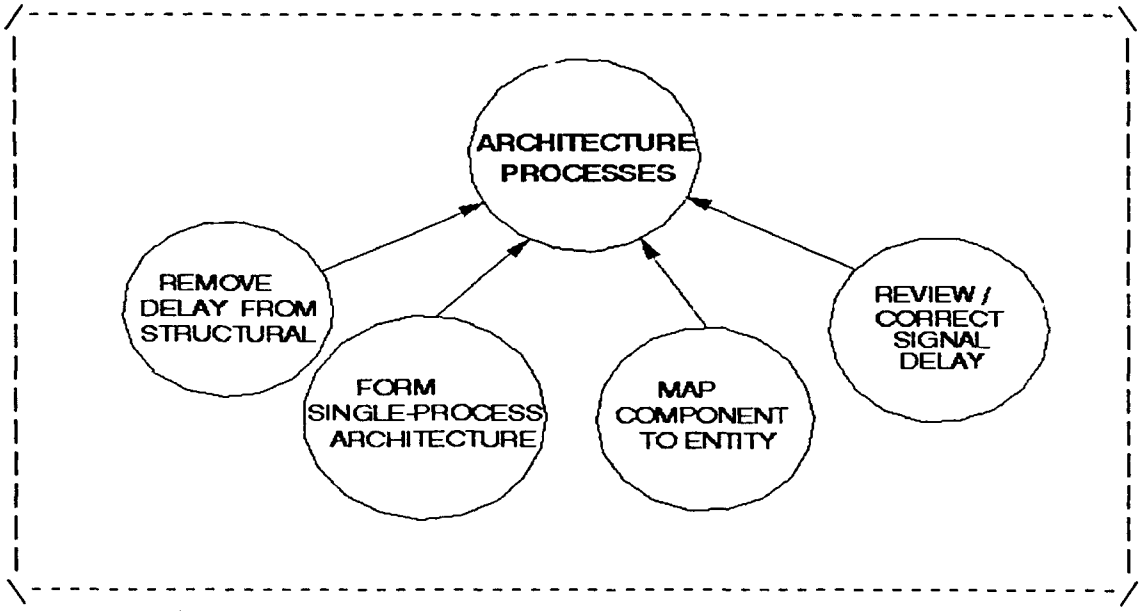


Figure 3-17. Main Architecture Processes DFD

3.5.3.5 Validate Condition Tests. To satisfy the JRS requirement that no tests for data values occur, Style-V must check each IF statement clause to ensure no such tests

are being done. Style-V must identify the control word and all aliases to it and all system clocks, since tests on these values are allowed. As stated in an earlier section, JRS specifically requires tests for a clock value to precede tests for a control value.

Identifying the clock and control signals used in the system is relatively easy, since a process can scan all files and look for declarations with type clock and control respectively. Also, any aliases that use fields of the control variables are then easily identified.

Once the clock and control variables are identified, the process need only parse all the files of the system looking for IF statement conditional checks. When a conditional check is found, it is checked to see if it contains anything other than a test for one of the defined clock or control variables. If it does, a warning is output to the user that redesign of the package in the file must be accomplished. The user has the option to continue checking for more data check violations or to abort the check at that time.

Conversely, a successful run is one in which no data type violations are found. A "SUCCESS" message is printed after each successful run. If a run was not successful, the message printed after the run will list all packages and files wherein violations occurred.

The validation of conditional tests Style-V process is given in Figure 13-18. A more detailed view will be presented in Section 3.5.4.

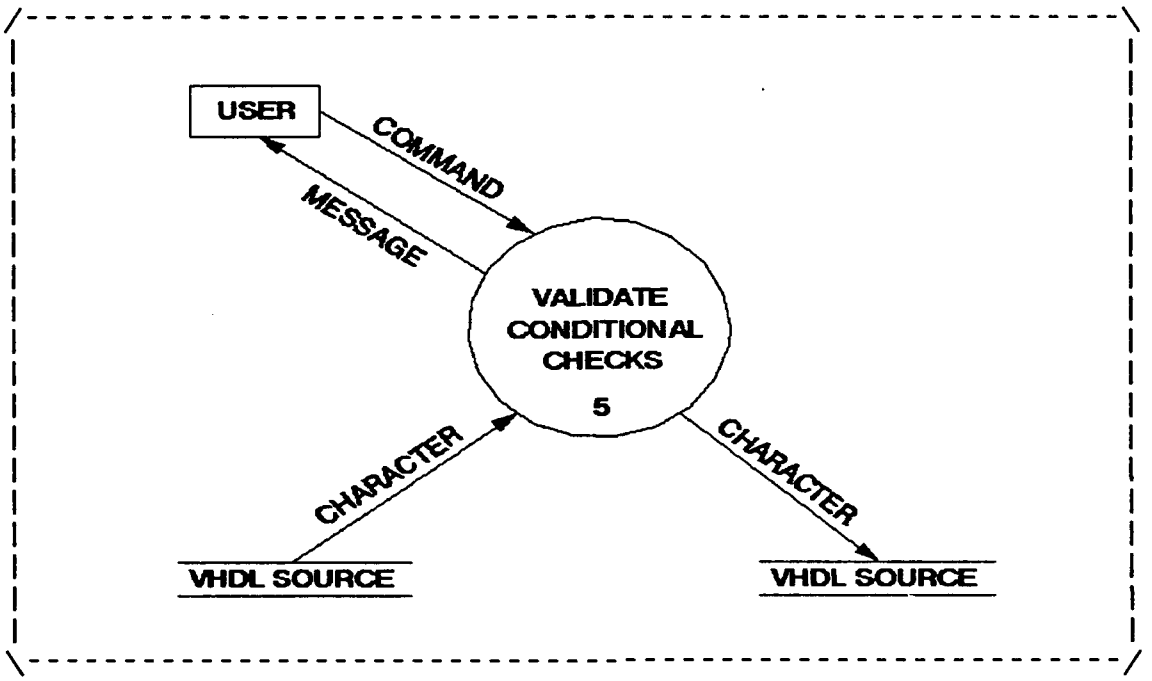


Figure 3-18. Conditional Test Validation DFD

3.5.3.6 Map Procedures. Probably the most difficult task faced by Style-V is the mapping of user defined functions and procedures to JRS IDAS procedures. Section 3.4.4 contains an overview of this problem and provides and example to show the complexity of the task. The term "function" used in this section represents either functions or procedures.

Based on the facts presented in Section 3.4.4, Style-V uses an interactive session with the user to map functions to JRS IDAS procedures. This requires the user to have the

IDAS function declarations (and possibly even bodies) available for comparison with their user-defined functions.

The user has two main tasks. The first is to identify the JRS IDAS procedures which map to their functions and procedures. Second, if a function does not map to any combination of IDAS functions, the user can either redesign that function or tell Style-V to use the function anyway. As noted in an earlier section, including functions not defined by IDAS is allowed, but the microcode generator will not use them. Finally, if a user defined function maps to a combination of IDAS functions, the user will need to identify the order of processing and the parameter assignments.

Once the user has provided the procedure mappings, Style-V comments out the existing user defined function or procedure declarations and their respective bodies. Also, all calls to the previous user defined procedures are converted to calls to appropriate JRS IDAS functions. Finally, a USE clause is added to each file of the design so the calls to the JRS IDAS procedures will work.

The high level representation of the procedure mapping process is represented in Figure 3-19. A more detailed description is provided in Section 3.5.4.

3.5.3.7 Perform Complete Stylization. A complete stylization entails performing all the processes described in Sections 3.5.3.1 through 3.5.3.6. Figure 3-20 represents this user option and is displayed to indicate no specific

order is required when performing the stylization process due to the orthogonality of the processes.

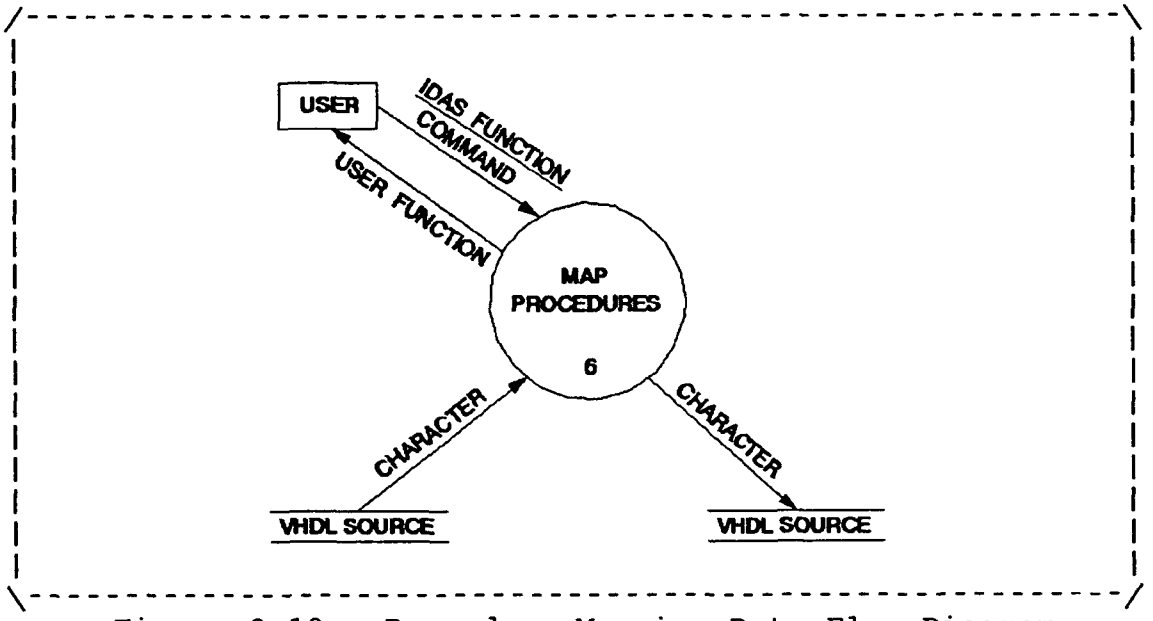


Figure 3-19. Procedure Mapping Data Flow Diagram

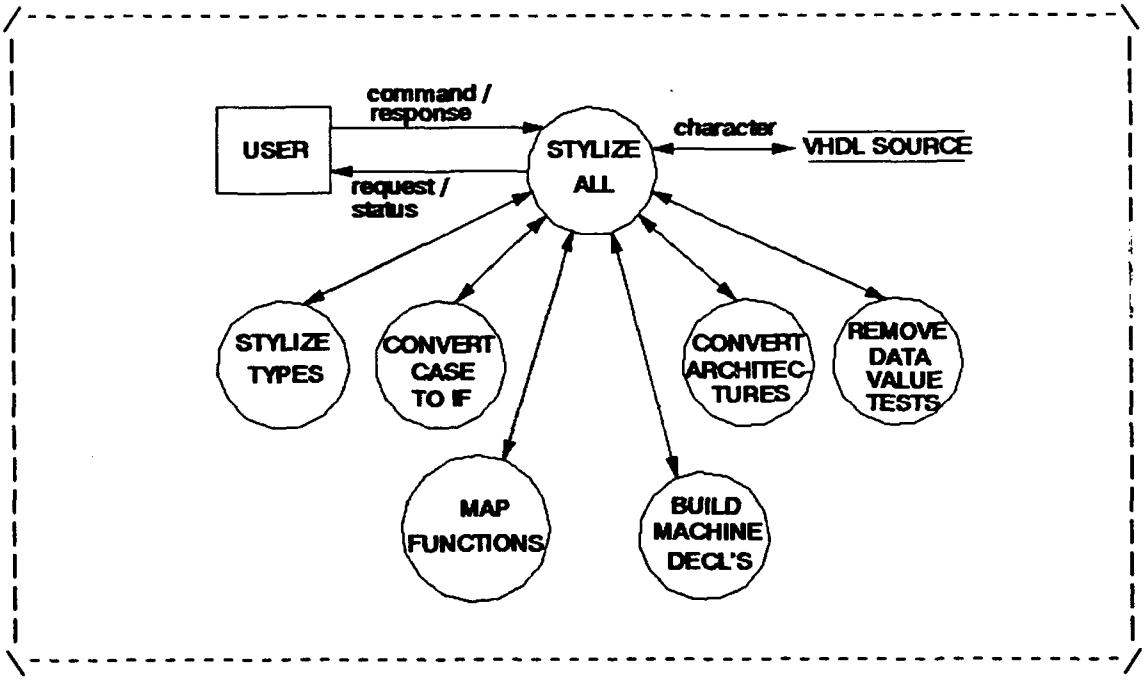


Figure 3-20. Option for Complete Stylization of a Design

3.5.4 Leveling of Style-V DFDs. Now that the high level DFDs for Style-V are defined, it is possible to decompose each Style-V process, capturing the decomposition in a lower level DFD. This method is backwards to Ward and Mellor's method described by Davis where the lowest level DFDs are described first (DAVIS, 1990:91). The reason is that their method works with known tasks from an existing system. Since Style-V does not yet exist, the top-down approach of this thesis identifies the tasks of the system by incremental decomposition and captures the results in lower level DFDs.

The level 1 DFDs for the Style-V translator were provided in Section 3.5.3. This section provides the first level of decomposition following level 1. The level 2 diagrams for Type Conversion, CASE_TO_IF Conversion, Create MACHINE DECLARATIONS, Architecture Conversion, Conditional Test Validation, and Procedure Mapping are provided in this section.

The level 2 DFD for type conversion is presented in Figure 3-21. The lexical and semantic processing of the type conversion process are more clearly visible at this level. The Scan_For_Declarations process scans for declaration statements. When they are found they are passed to a declaration converter process. All other input is passed to an intermediate file. After the Declaration_Converter processed declarations to the same intermediate file. Then,

when all of the declaration conversion is done, a Status_Type_Converter process finishes the type conversion process and writes the output files.

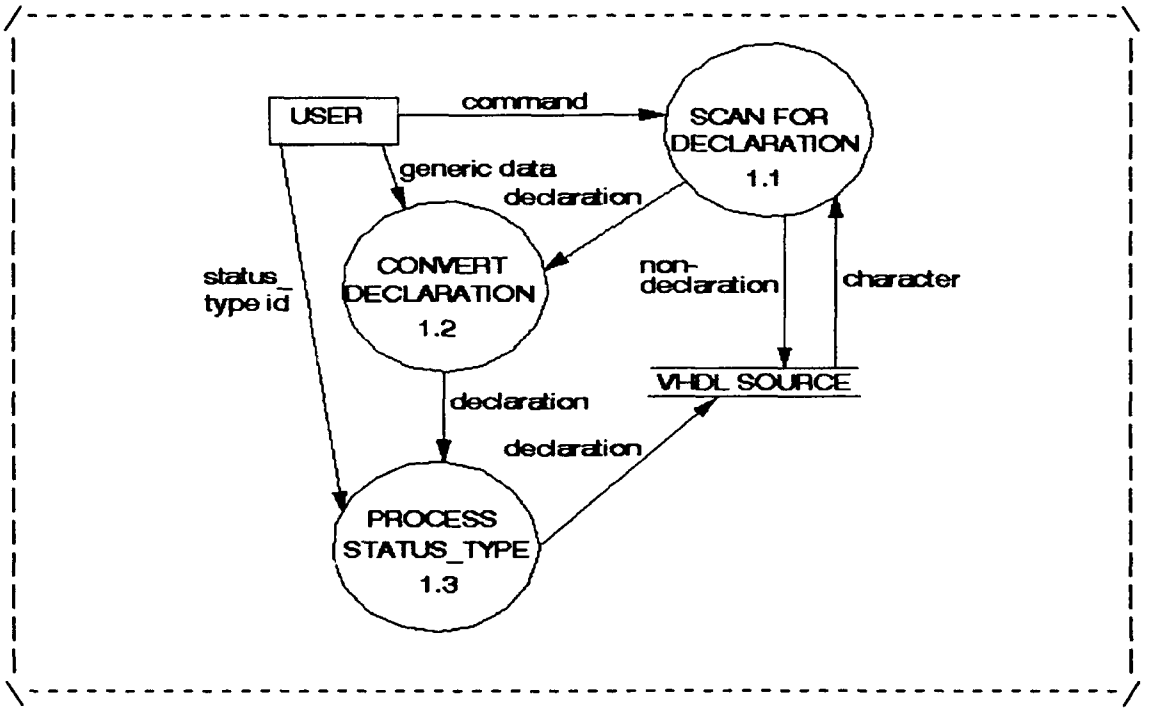


Figure 3-21. Type Conversion DFD -- LEVEL 2

The level 2 DFD for CASE_To_IF_Conversion is provided in Figure 3-22. The processes represented in this DFD were described in Section 3.5.3.2. This DFD describes the data flow and processes necessary to implement the behavior of the CASE-To-IF Conversion DFD found in Figure 3-14. A medium level of detail describing the actions necessary for converting VHDL CASE statements to VHDL IF statements is represented in the DFD of Figure 3-22.

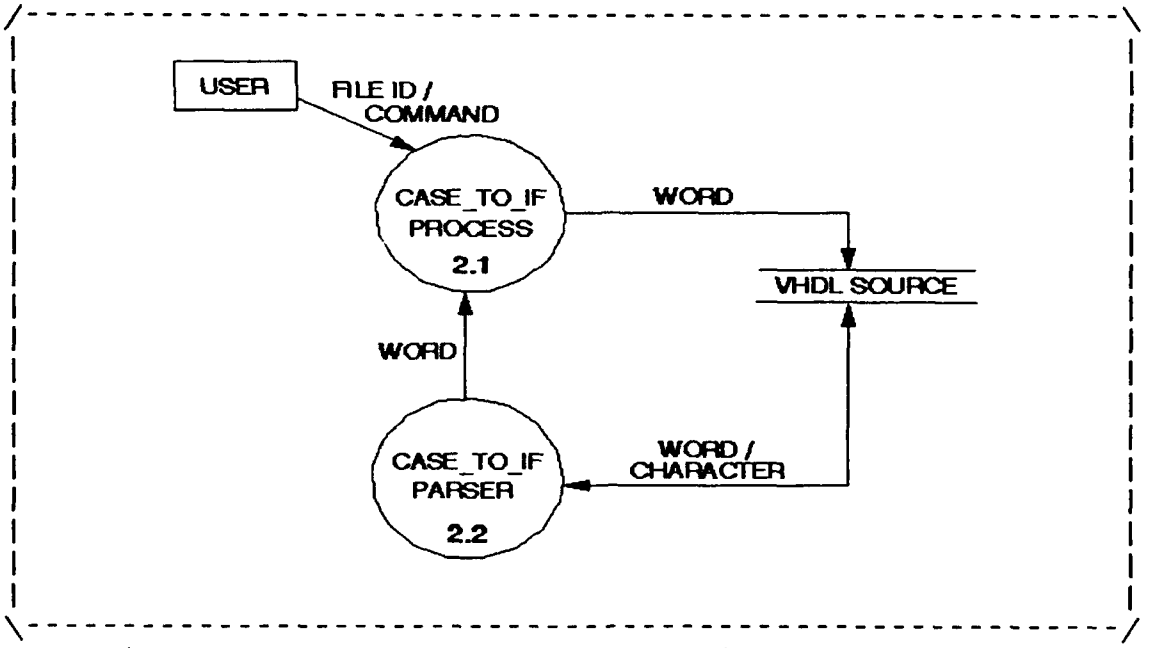


Figure 3-22. CASE_to_IF Conversion DFD -- Level 2

The level 2 DFD for collecting machine specific declarations in package MACHINE_DECLARATIONS is provided in Figure 3-23. This DFD shows the data flow and processes necessary to implement the behavior of the process in the Create MACHINE DECLARATIONS DFD of Figure 3-15. The processes for this level of detail were described in Section 3.5.3.3.

The level 2 DFD for architecture processing is provided in Figure 3-24. This DFD shows the subprocess required to process a standard VHDL architecture into a JRS IDAS styled VHDL architecture.

The level 2 DFD for validating conditional variables is provided in Figure 3-25. The processes necessary for checking whether a control variable is a clock_type or

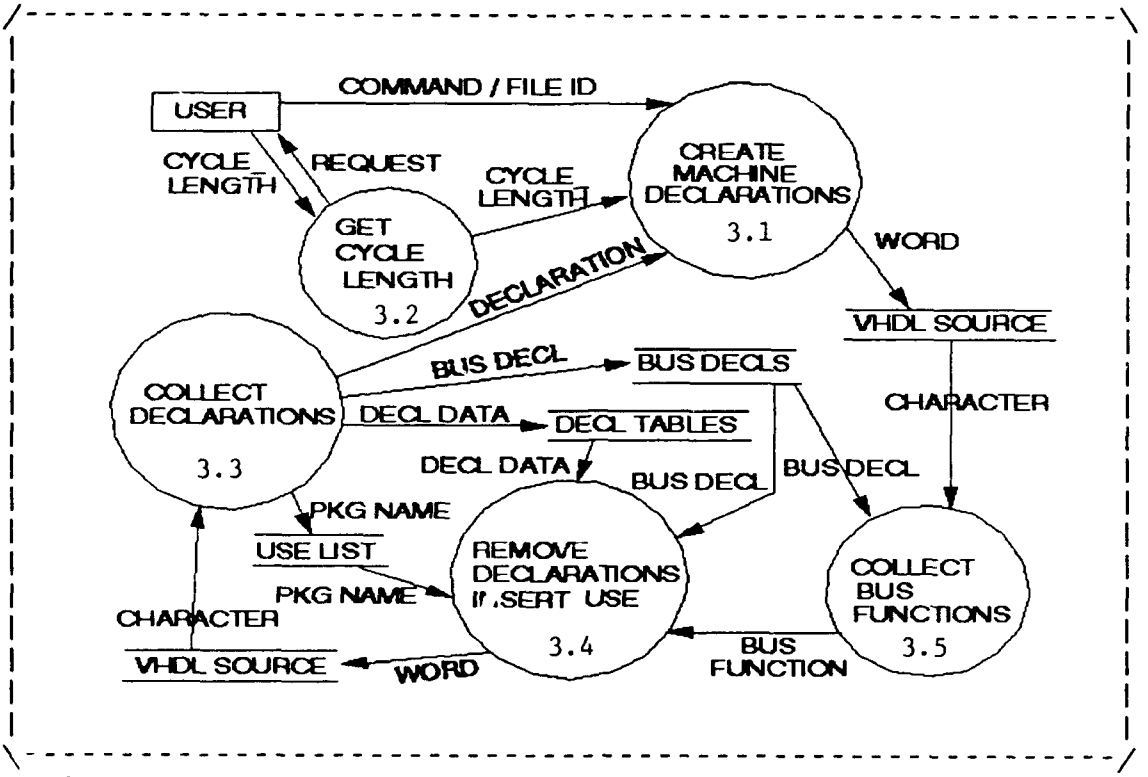


Figure 3-23. Create MACHINE_DECLARATIONS DFD -- Level 2

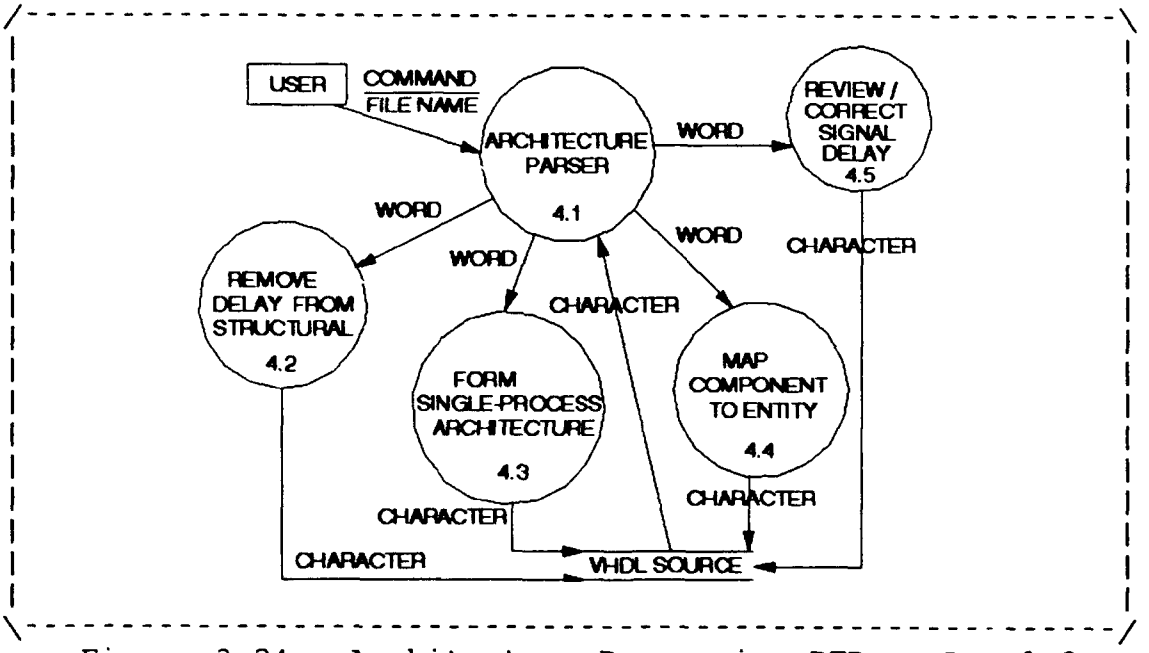


Figure 3-24. Architecture Processing DFD -- Level 2

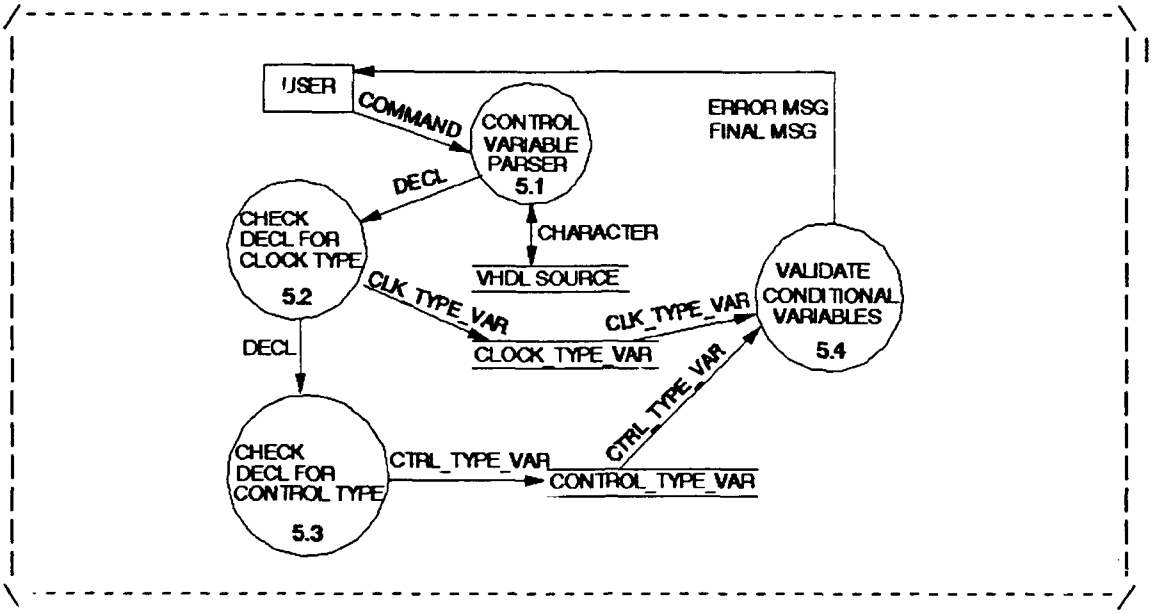


Figure 3-25. Validate Conditional Variables -- Level 2

control_type are identified, as well as the process that reports to the user if an illegal conditional variable is found.

The level 2 DFD for mapping procedures is provided in Figure 3-26. This DFD shows the user interaction required to perform mappings between JRS IDAS and user-defined procedures. It also shows the process of updating the source files once a mapping has been accomplished.

3.5.5 Style-V Data Dictionary. The data dictionary (DD) for the Style-V system was built as the DFDs were constructed. By doing this, the meaning of each term in the context in which it was used was preserved. The DD gives the item name and a description of the item.

Common notation and short English phrases are used for item descriptions. The common notation consists of item or

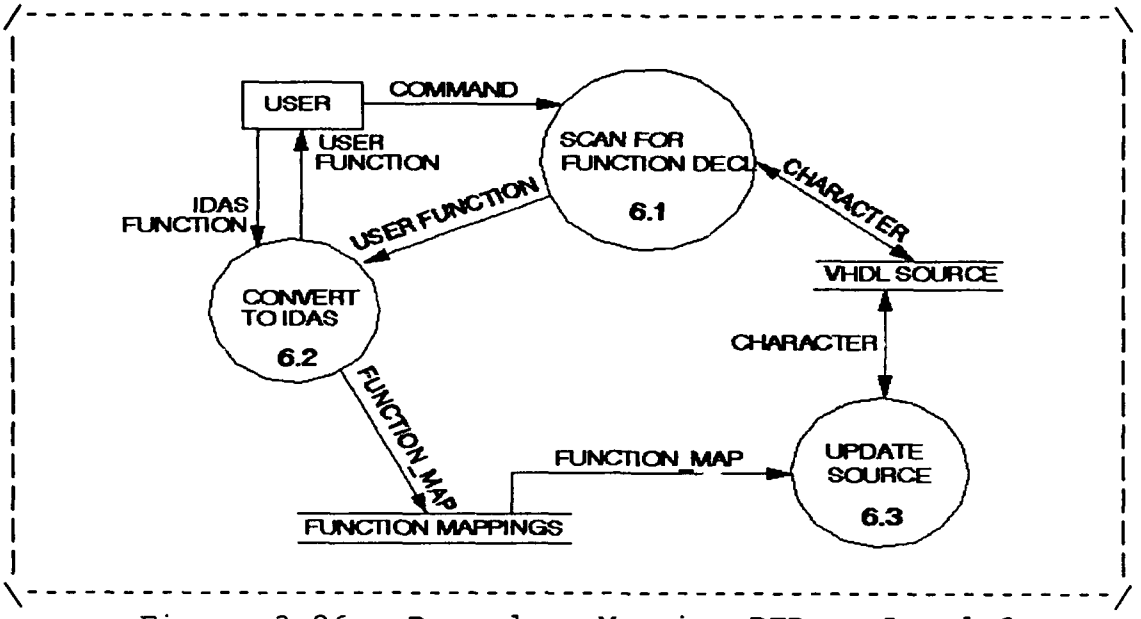


Figure 3-26. Procedure Mapping DFD -- Level 2

range identifiers and the use of option and repeat operators. An example of a range identifier is a..z which stands for the small alphabetic letters between a and z inclusive. An example of a repeat operator is $(a..z)_1$ which represents one or more small alphabetic letters. The optional operator (|) is placed between items to signify that any of the items in the optional group can be picked as the value of the item. An example of optional definition is (a | b | 9B), signifying that the item being defined is either the letter a, the letter b, or the phrase 9B. Grouping is done inside brackets which can be operated on by repeat notation, or grouping can be accomplished by using a different item defined elsewhere in the DD as part of the definition. Recursive and circular definitions should only be used when

an item can go to a null string -- the definition must represent a finite sequence of substitutions.

The following alphabetical listing is the DD for the Style-V Translator:

architecture	:	collection of statements forming a VHDL architecture
ARCHITECTURES	:	$(\text{architecture})_0$
bit_type	:	type mark allowed in JRS styled VHDL
character	:	(a..zA..Z0..9_)
component	:	a VHDL design item
COMPONENTS	:	$(\text{component})_0$
conditional	:	guard | sensitivity_list
CONDITIONALS	:	$(\text{conditional})_0$
confirm_flag	:	(Y | y | N | n)
count	:	integer
COUNT_TOTAL	:	a data store to accumulate a total
declaration	:	$(\text{name},)_0$ (name) (decl_tail)
decl_name	:	name portion of a declaration
decl_tail	:	portion of a VHDL declaration statement encompassed by ":" and ";"
ENTITIES	:	$(\text{entity})_0$
entity	:	a VHDL design unit
extension	:	(.) (extension_tail)
extension_tail	:	$[\ (\textit{name})\ (\textit{extension})_0\ ]_0$
file_name	:	(name) $[\text{extension}]_0$
flag	:	$(\text{character})_1$

generic_type : a type designator in a generic clause

guard : conditional value for process control

heading : VHDL code of the form ARCHITECTURE (BEHAVIORAL | STRUCTURAL) of ARCHITECTURE_NAME

integer : $(0..9)_1$

INTERMEDIATE : a temporary store for process results

letter : (a..zA..Z)

name : (letter | number) $(character)_0$

nondeclaration : a group of words, not a declaration

number : (0..9)

ORIG_ENTITY : high-level architecture entity store

port_list : a list of entity port declarations

possible_constant : a word which may be a constant name

possible_declaration : a word, possibly a declaration part

possible_variable : a word which may be a variable name

sensitivity_list : signal values for process control

special_character : nonblank noncharacter single element

status_function_call : convert function for a status type

status_type_decl : a declaration for a status variable

total : accumulated value of a data store

type_mark : a declaration type designator

unmapped_generic : type not mappable to JRS style

VHDL SOURCE : file storage of VHDL source code

word : $[$ character | special character $]_1$

WORDS : a group / store of zero or more words

The data dictionary described above provides a developer with a frame of reference when designing the modules to implement the behaviors identified during analysis. This design phase is covered in the next chapter.

3.6 Review

The focus of Chapter 3 has been the analysis of the requirements for the Style-V Translator to convert a standard VHDL design into a styled VHDL design acceptable to the JRS IDAS system.

Section 3.1 provided a review of notations and methods for analyzing a system. A possible notation is a data flow diagram (DFD), data dictionary (DD), control flow diagram (CFD), entity-relationship diagram (ERD), or some variant of these. The analysis methods which use these notations are Structured Requirements Definition, Structured Analysis and Design Technique, Structured Analysis and System Specification, Modern Structured Analysis, Problem Statement Language / Problem Statement Analyzer (PSL/PSA)^{TM}, and Object-Oriented Problem Analysis. Two other simpler analysis methods used for small problems are the method of listing inputs and outputs and the method of listing major functions.

Section 3.2 discussed the choice of Modern Structured Analysis as the method for the analysis of Style-V. Basically, the translations and mappings done by Style-V are functionally intensive, so a functional decomposition was chosen.

Section 3.3 provided a starting place to consider the functions of a translator system. The lexical, syntactic, and semantic processes which compose a translation system were described. Armed with the knowledge of the functionality required of a translator and knowing that the input was already syntactically correct, it was then possible to ensure the main processes of Style-V provided lexical and semantic processing capabilities. Each of the main Style-V processes has a lexical function which reads input files and forms tokens understandable to the remaining functions of the process. The functions of the processes that do most of the work of Style-V (like converting types) are semantic functions as they modify the input files but in a way that maintains the original meaning -- as much as possible.

Section 3.4 was a detailed look at the problem faced by Style-V. The differences between the VHDL subset defined by JRS and standard VHDL were compared. The differences could be characterized as type, declaration, structural, statement, and other. An example stylized machine was discussed and lessons learned from the example were provided. Section 3.4 ended with a discussion of the limitations of the JRS style of VHDL and the difficulties posed by these limitations.

Finally, Section 3.5 was the Modern Structured Analysis of the Style-V system. It started with a view of the system context. Then, the external events affecting the system

were identified. Next, the Style-V processes which responded to the external events were documented at a high level of abstraction. The next step was to decompose and level the data flow diagrams into a more detailed representation of the system. During this leveling process, the true magnitude of the Style-V creation task was realized and the thesis was scoped to concentrate on four of the six main processes of Style-V. During the creation of the data flow diagrams, a data dictionary was produced to document the data moving in the system.

Now that the Style-V Translator system has been analyzed, Chapter 4 will document the manual demonstrations of the concepts described above. The processes of converting VHDL CASE statements to IF statements, converting free-form VHDL types into the more restricted JRS IDAS types, converting multiple process architectures into single process architectures, and converting user-defined procedures into JRS IDAS procedures will be demonstrated.

IV. Demonstration of Concept

4.0 Introduction

It is not possible to complete an implementation of Style-V during one thesis cycle -- the problem is just too large. Therefore, instead of following Chapter 3 with a design chapter, Chapter 4 provides documentation of manual implementations of some of the mappings required for Style-V. The manual demonstrations show the feasibility of the concepts derived during the analysis described in Chapter 3. The implementation of other Style-V processes will use similar techniques to those this chapter describes. Also, this thesis effort has laid the ground work for any effort which would further implement Style-V.

Section 4.1 discusses the selection of concepts for manual implementation. Section 4.2 provides a description of a manual implementation of the mappings using portions of the VHDL design of a substantial integrated circuit now being developed by the Air Force -- the FPASP. Section 4.3 discusses the lessons learned from the manual implementation process. Finally, Section 4.4 provides a review of this chapter.

4.1 Selection of Concepts

Four of the mappings were chosen for manual demonstration of the translation concepts for mapping standard VHDL

to the JRS subset. The concepts of CASE statement conversion, type conversion, architecture conversion, and procedure mapping were chosen as being representative of the tasks of Style-V. Demonstration of these mappings shows the unfeasibility of a fully automated implementation and demonstrates that only a partially automated solution is possible. The remainder of this section describes the rationale for choosing these mappings.

CASE statement conversion was chosen as one of the tasks for manual implementation due to the prevalence of CASE statements in VHDL designs. Conversion of CASE statements also affects the implementation of behaviors as described in the JRS Style Guide, since system behaviors are to be captured in behavioral IF statements (VHDL Style, 1989:11,13). The conversion of CASE statements also represents a category of the translation tasks dealing with local modifications of input files -- global knowledge of the system is not required. Other mappings falling in this "local look" category are converting modes BUFFER and LINKAGE to mode INOUT, using GUARD and SENSITIVITY LISTS to build WAIT statements, moving signal delay to an AFTER clause, and removing structural architecture signal delay.

Another mapping required to translate standard VHDL into JRS styled VHDL is type conversion. JRS allowed types are quite limited when compared with the rich type structures allowed by standard VHDL. The ability to map user-

defined types which use bit and bit_vector logic is basically a direct translation to the JRS bit types. However, most VHDL designs use multilevel logic when representing modern integrated circuits. This multilevel logic is usually represented by a base type which is an enumerated type showing the representation of signal states. An example is the use of the enumerated type ('0', '1', 'X', 'Z') to represent the states zero, one, don't care, and high impedance.

Successful mapping of these multilevel logic types to JRS bit types represented quite a challenge. Tasks similar to the type conversion task but not part of the FPASP manual demonstration are generic clause conversion and status type conversion.

Architectural conversion from multiple concurrent statement architectures (those with block or multiple process statements) to single process architectures is described and the manual demonstration documented. The ability to perform architecture conversion was dependent on having a global view of an entire architecture and the entity which defined the interfaces of the architecture. From these, new lower level entities and architectures could be constructed. The goal of architecture conversion was to maintain a system view of the architecture which was the same as the original architecture. To obtain this view, the multiple internal processes had to be converted to single process

subarchitectures as components of a new architecture with the same external view as the original. Since the problem of mapping multiple concurrent statement architectures to single process architectures deals with limited global knowledge, it is somewhat similar to the problems of gathering machine declarations into one file and matching components to the entities they instantiate -- two problems not chosen for manual implementation using the FPASP.

Procedure mapping was chosen due to the criticality of the requirement to be able to map user-defined procedures to JRS IDAS defined procedures. This thesis worked toward developing a translator for the microcode generation portion of the JRS IDAS. The microcode generator of the IDAS does not recognize any user-defined procedures, but can only generate microcode for behaviors modeled using JRS provided procedures. Therefore, it is absolutely critical to map user-defined functions and procedures to JRS provided procedures. The ability to map functions required a high level knowledge of the logic of the functions being mapped. Other than the transformation of types, which is also included as a manual implementation, no other mappings are similar to the task of mapping procedure calls.

4.2 Manual Implementations

This section provides a detailed explanation of the manual implementation of the four mappings described in

Section 4.1. Section 4.2.1 describes the CASE statement conversion using the FPASP upper data path as an example. Section 4.2.2 describes the type translations carried out on the types of the FPASP. Section 4.2.3 describes the architecture conversion process using the same upper data path as used for the CASE statement conversion. Finally, Section 4.2.4 describes the procedure mappings carried out using the ALU functions description of the upper data path of the FPASP.

4.2.1 CASE Statement Conversion. Since a VHDL CASE statement is a more compact notation to perform the function of an IF-THEN-ELSIF-ELSE structure, conversion of the CASE statement to an IF statement is not difficult. All the information to form the IF statements is readily available in the CASE statement structure.

Consider the general form of the CASE statement in Figure 4-1, and compare it with an equivalent IF statement in Figure 4-2. The conditional variable (cond_var) of the CASE statement is compared against the conditional value (cond_val) in the WHEN portions of the CASE statement to determine which alternative is executed. Using an IF-THEN-ELSIF-ELSE structure, the cond_var is compared with the cond_val until it is equal and that determines which VHDL statements are executed. Therefore, the only knowledge required to translate from CASE to IF-THEN-ELSIF-ELSE structure consists of the conditional variable, the conditional

values, and the VHDL statements which are executed for a given conditional value.

```
CASE cond_var IS
     WHEN cond_val1 => VHDL statements
     WHEN cond_val2 => VHDL statements
     WHEN cond_val3 => VHDL statements
       .      .     .    .      .
       .      .     .    .      .
       .      .     .    .      .
     WHEN cond_valn => VHDL statements
     WHEN others    => VHDL statements
END CASE;
```

Figure 4-1. General CASE Statement Structure

The documentation provided by JRS on the style of VHDL acceptable to IDAS did not clearly describe the allowed structure for an IF statement. The approach of this thesis was to convert CASE statements to the IF-THEN-ELSIF-ELSE structure and run some translated code through the IDAS to verify that was an acceptable structure. Should the IDAS have rejected the IF-THEN-ELSIF-ELSE structure, it would have been a simple matter to change the translator to produce only IF-THEN statements as the translation of the VHDL CASE statement. Since the portion of the IDAS required for this thesis did not work, it was not possible to validate the CASE to IF conversion hypothesis.

The process of generating only IF-THEN structures instead of IF-THEN-ELSIF-ELSE structures would simply entail

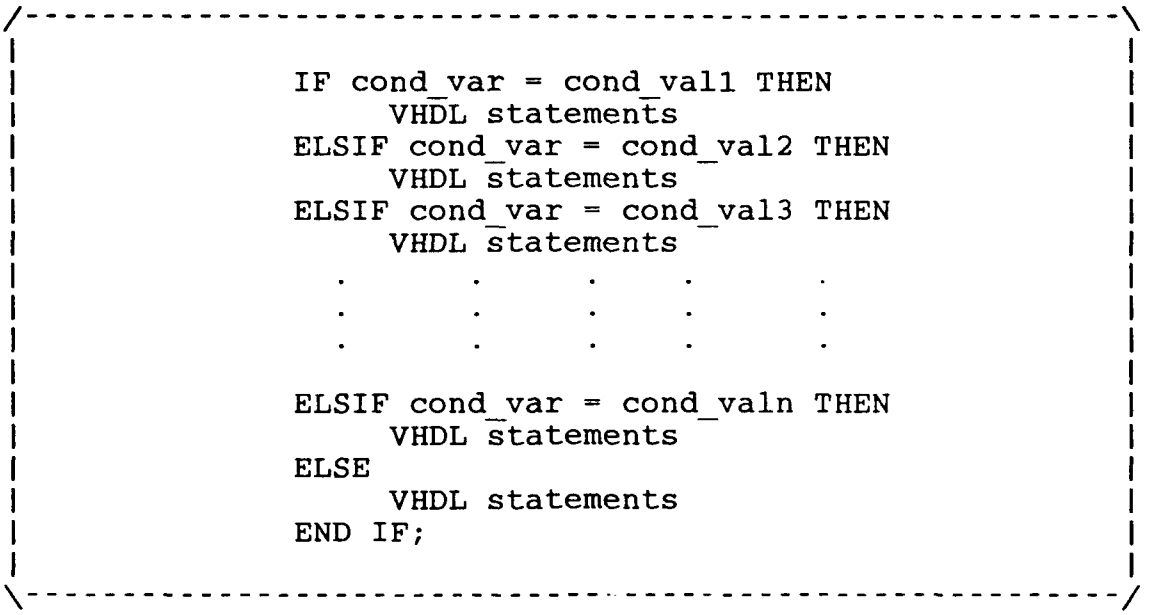


Figure 4-2. General IF Statement Structure

generating an END IF followed by an IF each time an ELSIF or an ELSE would be generated. Following the IF, should a conditional value be present, the condition of the IF would check the equality of the cond_var and the current cond_val. On the other hand, should no cond_val be present (processing the ELSE clause), the IF condition would be to check if the cond_var was not equal to any of the conditional values used in the previous IF statements during the processing of the current CASE statement. This means a simple list of the conditional values must be maintained if CASE statements are translated to IF-THEN statements only. See Figure 4-3 for a simple example.

The CASE to IF-THEN-ELSIF-ELSE process was the only Style-V process taken past the analysis and preliminary

```
CASE X IS
     WHEN 1       =>  statements_1
     WHEN 2       =>  statements_2
     WHEN others  =>  statements_3
END CASE;
```

```
IF (X = 1) THEN
     statements_1
END IF;
IF (X = 2) THEN
     statements_2
END IF;
IF not ((X = 1) or (X = 2)) THEN
     statements_3
END IF;
```

Figure 4-3. CASE to IF-THEN Example

design phases to actual implementation. This module successfully converted the CASE statements of the architectures extracted from the upper data path of the FPASP design.

The CASE_to_IF module was run on the new architectures UPPER_REGISTERS, UPPER_ALU_SHIFTER, FUNCTION_ROM, and LITERAL_INSERTER which were created by the architecture conversion process (described in Section 4.2.3). However, as noted before, these conversions are orthogonal and the same results would occur should the CASE to IF conversion be done before the creation of new architectures.

Implementation of the CASE_to_IF module produced successful results. Sample results from running it against FPASP code comprise Section 1 of Appendix C. The CASE_to_IF

pseudo code, C code, and tests and results are located in Appendix D.

4.2.2 Type Conversion. As described in Chapter 3, type conversion posed several challenges. The JRS types are limited to bit types -- those based on bit logic consisting of 1's and 0's. This restriction does not allow types that represent multilevel logic, such as the enumerated type BUS_BIT used by the FPASP to represent four-level logic. The type BUS_BIT uses '0' to represent zero, '1' to represent one, 'X' to represent don't care, and 'Z' to represent high impedance.

Package and package body FPASP4_2_TYPEDEF are included in Appendix B and contain the type definitions for the FPASP VHDL description. This code was examined and an algorithm devised to translate the four-level logic of the FPASP to a two-level (bit) logic representation. The translated FPASP4_2_TYPEDEF package specification and body are located in Appendix C.

The first review of the declarations package FPASP4_2_TYPEDEF revealed that most of the types were defined with a base type of BUS_BIT. As described above, BUS_BIT was an enumerated type ('X','0','1','Z'). Since this type of enumerated type is not allowed in the JRS IDAS style of VHDL, a conversion to a bit type was necessary.

Since only '0's and '1's are allowed by the IDAS, the first approach considered was to convert 'X' and 'Z' to

either '0' or '1'. The problem with this type of conversion is that some of the functions defined for the FPASP depend on the ability to model multilevel logic. A further discussion of this problem is in Section 3.4.1.1 and a sample section of procedure code that would always cause "incorrect" procedure execution was provided in Figure 3-7.

Since a simplistic conversion was not possible, another method was necessary. The next step was to consider representing each bit of the BUS_BIT type by a bit_vector of length two. A '0' would be represented by the bit_vector "00", a '1' by "01", a 'X' by "10", and a 'Z' by "11". The translation of any type with a base type of BUS_BIT would then entail using multilevel bit vectors. For example, the BUS_BIT_VECTOR "01XZ" would be represented as a vector of bit vectors ("00","01","10","11"). This type of translation may work for some tools; however, the JRS IDAS Move procedure which is used to move data around a design can only process bit_vectors -- no multidimensional bit vectors.

Knowing that the IDAS required data to be in the form of bit_vectors, the next logical step was to consider using two bits to represent one logical bit. Using this approach, a 0 would be represented by 00, a 1 by 01, a X by 10, and a Z by 11. Then, the example BUS_BIT_VECTOR "01XZ" would become the BIT_VECTOR "00011011". This solution was suitable for the IDAS MOVE procedures. However, another concern arose. The other IDAS procedures which would replace ALU-

type operations will use bit vectors from the system and produce results from them, but these procedures expect true two-level logic bit vectors, so a conversion function must translate the two-level logic representation of four-level logic into true two-level logic before entering any of the ALU-type IDAS procedures and back again to four-level logic represented in two-level logic after the procedures to maintain compatibility with the rest of the FPASP chip.

To lessen the amount of "name" changing the translator would do, the base type name was maintained and the new representation for the type introduced. Since representing the four-level logic elements as bit vectors would cause IDAS Move problems, the BUS_BIT type became ('0', '1') -- a type compatible with type BIT. Should this have not been acceptable to IDAS, changing the type BUS_BIT to subtype BUS_BIT of BIT would have worked as an alternate option and not affected the remaining translations.

Now, since each BUS_BIT in the BUS_BIT_VECTORS is represented by two bits, the specified length of each BUS_BIT_VECTOR declaration must be doubled. For example, BUS_BIT_VECTOR(31 downto 0) would become BUS_BIT_VECTOR(63 downto 0) and BUS_BIT_VECTOR(1 to 7) would become BUS_BIT_VECTOR(1 to 14). In general, the new lower index has the same value as the original lower index and the formula for calculating the upper index is:

```
Upper_Index := High_Index + High_Index - Low_Index + 1.
```

The decision on which index to change is determined by the word between the indexes. If the word "downto" is between the indexes, the left index is changed, otherwise the right index is changed.

Another concern is constant declarations. Not only does the index range need adjustment, but the literal representing the constant must be adjusted also. However, since the logical bit of each true bit is simply a double bit, the replacement can be made by replacing each bit of a constant literal with the corresponding double bit. For example, the constant literal "ZZXX10" would become "111110100100".

From the above discussion, it is easy to see that only local knowledge is required to change type declarations. Once the base type is identified and changed, changing the other declarations is simply a matter of text replacement or index calculation and replacement. Changing procedure and function declarations is done in the same way. However, within procedure and function bodies, the calculations and modifications are more difficult.

Not only do loop ranges need to be calculated in a similar way to declaration ranges, but the assignment logic inside a loop must account for assigning two bits instead of one. For example, if a pretranslation assignment statement was:

```
XX(I) := '1',
```

the new assignment statement would be:

```
XX(I*2 to I*2+1) := "01",
```

or possibly:

```
XX(I*2+1 downto I*2) := "01".
```

Also, the logic of the loop would need to be modified to skip execution of the loop on every other pass because two bits were processed on the previous pass. Figure 4-5 provides an example of the FPASP code and the modified loop resulting from translation.

Another consideration evident from Figure 4-5 is how to translate simple constructs into more complex constructs. The calculation TEMP(I) := A(I) and B(I) of Figure 4-5 was simple due to the definition of the "and" operator for two single element BUS_BITs. However, since the translation now uses two-element BUS_BIT_VECTORS to represent an actual bit, the previous "and" function which performed an and of two BUS_BITs became one with two parameters of type BUS_BIT_VECTOR (1 downto 0) and a return type of BUS_BIT_VECTOR. However, this caused a compile error in the VHDL analyzer due to an ambiguous overloading of this "and" function with the more general "and" function with BUS_BIT_VECTOR parameters and return type.

The solution taken in the manual implementation was to expand the more general "and" function to perform all calculations necessary to produce the "and" result. The truth

table from the FPASP design which had to be implemented for the "and" operation is provided in figure 4-4.

OPERAND 2 BIT

AND	X	Z	0	1
X	X	X	0	X
Z	X	X	0	X
0	0	0	0	0
1	X	X	0	1

OPERAND 1 BIT

Figure 4-4. FPASP General "and" Truth Table

Another approach which would be simpler to implement for Style-V would be to rename the "and" function which had parameters BUS_BIT_VECTOR (1 downto 0) and to call the newly named function from the now simplified translation of the general "and" function. Figure 4-6 provides the more simplified translation of the general "and" function, and Figures 4-7 and 4-8 show the translation of the BUS_BIT function to the BUS_BIT_VECTOR (1 downto 0) function is basically a one-for-one textual substitution -- an easy translation.

BEFORE

```
function "and"(A, B : BUS_BIT_VECTOR)
                    return BUS_BIT_VECTOR is
variable TEMP : BUS_BIT_VECTOR(31 downto 0);
begin
     for I in A'LOW to A'HIGH loop
               TEMP(I) := A(I) and B(I);
     end loop;
     return TEMP(A'RANGE);
end "and";
```

AFTER

```
function "and"(A, B : BUS_BIT_VECTOR)
                    return BUS_BIT_VECTOR is
variable TEMP : BUS_BIT_VECTOR(63 downto 0);
variable SKIP : BIT := '0';

begin
     for I in A'HIGH-A'LOW downto 0 loop
       if SKIP = '0' then
          if (A(I downto I-1) = "00") or
             (B(I downto I-1) = "00") then
                    TEMP(I downto I-1) := "00";
          elsif (A(I downto I-1) = "11") or
                (A(I downto I-1) = "10") then
                    TEMP(I downto I-1) := "10";
          elsif (A(I downto I-1) = "01") and
                (B(I downto I-1) = "01") then
                    TEMP(I downto I-1) := "01";
          else TEMP(I downto I-1) := "10";
          end if;
          SKIP := '1';
       else
          SKIP := '0';
       end if;
     end loop;
     return TEMP(A'high-A'low downto 0);
end "and";
```

Figure 4-5. Sample "Hard" Loop Translation

BEFORE

```
function "and"(A, B : BUS_BIT_VECTOR)
                    return BUS_BIT_VECTOR is
variable TEMP : BUS_BIT_VECTOR(31 downto 0);
begin
     for I in A'LOW to A'HIGH loop
               TEMP(I) := A(I) and B(I);
     end loop;
     return TEMP(A'RANGE);
end "and";
```

AFTER

```
function "and"(A, B : BUS_BIT_VECTOR)
                    return BUS_BIT_VECTOR is
variable TEMP : BUS_BIT_VECTOR(63 downto 0);
variable SKIP : BIT := '0';

begin
     for I in A'HIGH-A'LOW downto 0 loop
       if SKIP = '0' then
          TEMP(I downto I-1) :=
               A(I downto I-1) and2 B(I downto I-1);
          SKIP := '1';
       else
          SKIP := '0';
       end if;
     end loop;
     return TEMP(A'high-A'low downto 0);
end "and";
```

Figure 4-6. Simplified "and" Translation

BEFORE

```
function "and"(A, B : BUS_BIT) return BUS_BIT is
begin
     case A is
          when 'X' =>
            if B = '0'
              then  -- 'and' is dependent on B.
                return '0';
            else  -- Undefined.
                return 'X';
            end if;
          when 'Z' =>
            if B = '0'
              then  -- 'and' is dependent on B.
                return '0';
            else  -- Undefined.
                return 'X';
            end if;
          when '0' => return '0';
                      -- Doesn't matter what B is.
          when '1' =>  -- 'and' is dependent on B.
            if B = 'Z'
              then  -- Undefined.
                return 'X';
            else
                return B;
            end if;
     end case;
end "and";
```

Figure 4-7. "and" With BUS_BIT Type

AFTER

```
function "and2"(A, B : BUS_BIT_VECTOR(1 downto 0))
                              return BUS_BIT_VECTOR is
 begin
   case A is
     when "10" =>
       if B = "00"
         then  -- 'and' is dependent on B.
           return "00";
       else  -- Undefined.
           return "10";
       end if;
     when "11" =>
       if B = "00"
         then  -- 'and' is dependent on B.
           return "00";
       else  -- Undefined.
           return "10";
       end if;
     when "00" => return "00";
          -- Doesn't matter what B is.
     when "01" =>  -- 'and' is dependent on B.
       if B = "11"
         then  -- Undefined.
           return "10";
       else
           return B;
       end if;
   end case;
 end "and2";
```

Figure 4-8. "and" With BUS_BIT_VECTOR (1 downto 0) Type

Some type of data structure which contained the original declaration and the translated declaration would need to exist to facilitate the renaming of a less general function as described above. As the translator worked through a file, it would not know a more general function was coming which would have a type conflict with a newly translated function. A table implemented as a linked list of two-field

records would serve this purpose. The link list would provide quick order 1 time storage of new data, and since the number of procedures would be relatively small for most designs (n < 1000), the order n search time required to find a less general function would not be significant. In fact, since most "like named" functions generally occur in the same vicinity of a design, and since the less general function is usually defined before the more general ones, the search would be much less than n on average, though the worst case would still be on the order of n.

In summary, the type conversion process required to translate the FPASP with four-level BUS_BIT representation into bit logic required implementing a translation scheme to use bit logic to represent four-level logic, translating all declarations using the new representation, and translating any functions or procedures operating on the types of the system to use the new bit logic types. Some translations, such as declaration ranges and literal values was straight forward and consisted mainly of textual substitution. Other translations, such as function logic (for example, loops), was more difficult and required knowledge of how using two bits to represent one actual bit affected the system.

4.2.3 Architecture Conversion. Besides type restrictions, another major restriction of JRS IDAS for VHDL designs is the requirement to have only one process statement for a behavioral architecture. In standard VHDL, a designer

can model any number of concurrent processes and blocks in an architecture.

Section 3.4.1.3 discussed structural differences between JRS IDAS styled VHDL and standard VHDL, including a discussion of architectural differences. A detailed discussion of the algorithm for converting a multiple concurrent statement architecture to multiple single process architectures was given in Section 3.5.3.4. This section provides a review of an implementation of the architecture conversion algorithm on one entity and architecture of the FPASP.

The FPASP4_UDATAPATH entity and architecture were chosen for decomposition because they presented most of the translation challenges identified in the architecture conversion algorithm. For the FPASP4_UDATAPATH entity, one port had mode BUFFER, which is not an allowed mode for JRS IDAS styled VHDL. The entity had a generic clause, but the only type used in the generic clause was INTEGER, so no translation was necessary for the generic types. Finally, several BUS types were defined as entity ports.

Challenges represented by the FPASP4_UDATAPATH architecture were aliases of slices of entity ports, statements outside of any process or block, and processes within blocks. The successful manual conversion of this architecture shows the feasibility of converting architectures in general using the algorithm defined in this thesis. Refer to Appendix B for the original FPASP4_UDATAPATH entity and

architecture descriptions and Appendix C for the translated FPASP4_UDATAPATH entity and architectures.

The conversion process began by determining which statements did not fall within any process or block. These statements were set aside to be used in the final architecture which instantiated all the subarchitectures created in the next steps. These statements were saved to put in a begin-end structure for the new FPASP4_UDATAPATH architecture. This completed a first pass of the FPASP4_UDATAPATH architecture.

The second conversion step was to convert the block statements into process statements. Since each block consisted of a process statement, this was a simple matter of performing the following thirteen steps:

1. scanning for the block statement,
2. scanning the block guard (if any),
3. saving the guard for latter use,
4. removing the guard statement,
5. scanning for the process statement,
6. scanning the process sensitivity list (if any),
7. saving the sensitivity list for latter use,
8. removing the sensitivity list from the process statement,
9. passing through any declarations,
10. scanning the "begin" for the process,
11. creating and inserting a WAIT statement for the

process with an UNTIL clause for the saved guard conditions and an ON clause for the saved sensitivity list,

12. passing all information through until an "end process" phrase was scanned,

13. finally, deleting the "end block" phrase which followed the "end process" phrase.

The above processing would continue for the remainder of the FPASP4_UDATAPATH architecture. This processing comprises the second pass through the FPASP4_UDATAPATH architecture. The results of these second step actions would be a sequence of process statements with a WAIT statement with UNTIL or ON clauses as the first statement of the process -- a requirement of JRS styled VHDL. Figure 4-9 provides an abbreviated representation of the results of this process.

Once the block and process statements of the FPASP4_UDATAPATH were converted to a sequence of process statements with the JRS required WAIT statement, new entities for each process statement were created. The name of each of these new entities could have been any legal name allowed by VHDL which did not duplicate existing names in the design, but using labels of block or process statements of the original architecture made the printed output of the translation easier to follow.

To form the ports of each entity, each process statement was scanned to determine which ports (or aliases of

```
Architecture BEHAVIOR of FPASP4_UDATAPATH is

     -- declarations
     alias  x : ...
     alias  y : ...
          ...
begin
     UPPER_REGISTERS: process
     begin
          WAIT UNTIL guard1 ON sensitivity list1;
          {statements}
     end process;
     UPPER_ALU_SHIFTER: process
     begin
          WAIT UNTIL guard2 ON sensitivity list2
          {statements}
     end process;
     UINSERT:  process
     begin
          WAIT UNTIL guard3 ON sensitivity list3;
          {statements}
     end process;
     FUNCTION_ROM:  process
     begin
          WAIT UNTIL guard4 ON sensitivity list4;
          {statements}
     end process;
end BEHAVIOR;
```

Figure 4-9. Architecture With Process Statements Converted

port slices) were used in the process. By using only these to form the ports of the new entity, creation of unused ports was avoided. The port and alias declarations of the original entity and architecture were used to form the port declarations for the new entity. The mode was also maintained from the original entity for each port declaration in each new entity. An exception was the IDAS requirement to convert mode BUFFER to mode INOUT for one of the ports.

Another consideration was the use of generics. If a variable which matched a generic was found in a process statement, a generic clause was created for the new entity representing the process. In the FPASP implementation, only one process used the generics defined by the original entity, so only the entity for that process contained a generic clause.

After all entities representing the process statements in the design were created, it was time to create the new subarchitectures. This is an easy step at this point in the translation process. The first action required to construct each new architecture was to create the architecture header. The architecture header consisted of the keyword "architecture", followed by the architecture name (BEHAVIOR), followed by the keyword "of", followed by the entity name, followed by the keyword "is". The body of the architecture was simply the keyword "begin" followed by the process statement followed by the phrase "end BEHAVIOR;". A representation of the new architecture is provided in Figure 4-10.

Now that each process statement was converted into an entity and architecture pair, the final step of architecture conversion was to create the new FPASP4_UDATAPATH architecture. Each new entity was represented by a component with the same name as the entity and with the same port declarations as the entity -- as required for JRS styled VHDL.

```
Architecture BEHAVIOR of UPPER_REGISTERS is

begin

     UPPER_REGISTERS: process
     begin
          WAIT UNTIL guard1 ON sensitivity list1;
          {statements}
     end process;

end BEHAVIOR;
```

Figure 4-10. Representation of New Architecture

Each component was then instantiated and the ports assigned to the port or port slice of the original FPASP4_UDATAPATH entity. Finally, the statements which were outside any process statement of the original architecture were included in a begin-end block. Figure 4-11 provides an outline representation of the results of this process. Conversion of the FPASP4_UDATAPATH architecture was now complete.

4.2.4 Procedure Mapping. As explained in Section 3.4.4 the task of mapping one procedure to another is quite difficult. One way would be to use exhaustive testing, but two problems hinder using this solution for Style-V. The test driver for testing the procedures and comparing the results does not exist and would require a major development effort to produce. Secondly, given the facts described in Section 3.4.4, even simple procedures could not be "fully" tested in an acceptable amount of time.

```
Architecture BEHAVIOR of FPASP4_UDATAPATH is
     component UPPER_REGISTERS declaration
     component ALU_SHIFTER declaration
     component LITERAL_INSERTER declaration
     component FUNCTION_ROM declaration
begin
     component UPPER_REGISTERS instantiation
     component ALU_SHIFTER instantiation
     component LITERAL_INSERTER instantiation
     component FUNCTION_ROM instantiation
     statements not included in any process
end BEHAVIOR;
```

Figure 4-11. New Architecture After Conversion

The process chosen for Style-V was manual intervention, since no automated method seems possible at this time. The remainder of this section describes how a sampling of FPASP functions were mapped to JRS IDAS functions and describes a case where a mapping was not possible.

The FPASP procedure MOVNUL had one input bus type and one output bus type. This procedure mapped easily to the IDAS MOVE procedure which could support these ports while providing the same functionality. This same process was used on six other functions with good success. Figure 4-12 provides the FPASP procedure declarations and the

declarations for the IDAS procedure to which they mapped for three of the successful mappings. However, one FPASP procedure could not be mapped to an IDAS procedure or combination of IDAS procedures.

The FPASP procedure SR could not be mapped to any IDAS procedure. The declaration of this "shift" function and the "shift" functions provided by the JRS IDAS are provided in Figure 4-13. The function SR accepts a word (ALU_IN), assigns the rightmost bit to an out bit (SH_OUT), shifts the remaining bits one bit to the right, and assigns an input bit (SH_IN) to the leftmost position of the word to form the result (RESULT). None of the IDAS shift procedures provided this capability.

The translation method of transliteration and refinement will not work for procedure mapping since those methods use only local knowledge to perform translations. Additionally, the translation method of abstraction and reimplementation will probably fail to be capable of performing the procedure mappings, since:

> the key to the increase in abstraction ... is the ability to **recognize** the net effects of a computation. This in turn depends on the abstraction component having a significant amount of knowledge about what kinds of computations can be performed. (Waters, 1986:14)

The information provided by Waters indicates no current technology (transliteration and reimplementation orabstraction and reimplementation) provides a solution to the problem of function mapping faced by Style_V.

```
FPASP
MOVNUL (MOV_IN   :
          in DR32_RESOLVE BUS_BIT_VECTOR(31 downto 0);
        RESULT   :
          out DR32_RESOLVE BUS_BIT_VECTOR(31 downto 0));

IDAS
MOVE (In_Port   : in Data_Vector;
      Out_Port  : out Data_Vector;
      Name      : "MOVNUL" );
```

```
FPASP
ORUL (LEFT,RIGHT   :
        in DR32_RESOLVE BUS_BIT_VECTOR(31 downto 0);
      RESULT       :
        out DR32_RESOLVE BUS_BIT_VECTOR(31 downto 0);
      ZERO         : out BUS_BIT  );

IDAS
LOGICAL_OR (In_Port1   : in Data_Vector;
            In_Port2   : in Data_Vector;
            Out_Port   : out Data_Vector;
            Status     : out Status_Type;
            Name       : "ORUL"   );
```

```
FPASP
ADCUL (LEFT,RIGHT   :
         in DR32_RESOLVE BUS_BIT_VECTOR(31 downto 0);
       CARRY_IN     : in BUS_BIT;
       RESULT       :
         out DR32_RESOLVE BUS_BIT_VECTOR(31 downto 0);
       CARRY        : out BUS_BIT;
       OVERFLOW     : out BUS_BIT;
       SIGN         : out BUS_BIT;
       ZERO         : out BUS_BIT;

IDAS
ADDC (In_Port1   : in Data_Vector;
      In_Port2   : in Data_Vector;
      In_Port3   : in Data;
      Out_Port   : out Data_Vector;
      Status     : out Status_Type;
      Name       : "ADCUL"   );
```

Figure 4-12. FPASP to IDAS Procedure Mapping Examples

FPASP SHIFT RIGHT PROCEDURE

```
procedure SR (ALU_IN : in DR32_RESOLVE
                        BUS_BIT_VECTOR (31 downto 0);
              SH_IN  : in BUS_BIT;
              RESULT : out DR32_RESOLVE
                        BUS_BIT_VECTOR (31 downto 0);
              SH_OUT : BUS_BIT    );
```

JRS IDAS SHIFT RIGHT PROCEDURES

```
procedure SHIFT_RIGHT_1
              (In_Port   : in Data_Vector;
               Out_Port  : out Data_Vector;
               Name      : in String );
-------------------------------------------------
procedure SHIFT_RIGHT_1_BIT_IN
              (In_Port   : in Data_Vector;
               In_Bit    : in Data;
               Out_Port  : out Data_Vector;
               Name      : in String );
-------------------------------------------------
procedure SHIFT_RIGHT_1_ONEFILL
              (In_Port   : in Data_Vector;
               Out_Port  : out Data_Vector;
               Name      : in String );
```

Figure 4-13. FPASP to IDAS NonMappable Example

4.3 Lessons Learned From Examples

The following sections provide a summary of the lessons learned during the manual implementation of Style-V translations on the FPASP design. The difficulties experienced during these implementations and the lessons learned from them indicate care should be used when developing the remaining modules of Style-V. Lessons are also good for

insight into the types of processes necessary for developing a successful translator.

4.3.1 Lessons From CASE Conversion. The main lesson learned from the CASE to IF analysis and implementation was that modifications which require only local knowledge in the source code are algorithmically simple to design and implement. Little temporary storage and no complicated data structures are required for processing translations similar to CASE to IF conversion.

4.3.2 Lessons From Type Conversion. The main lesson from the type conversion exercise was that a simple approach is sometimes inappropriate, and it may be plain wrong. A strong analysis should always be conducted to determine how a course of action affects the entire system. In the case of type conversion, it would have been wrong to simply compress multilevel logic into bit logic by assigning the values '0' or '1' to some other previously defined value. A thorough analysis showed the best way was to represent each actual bit with two logical bits.

Another translation lesson learned from type conversion is that decisions early in a process affect design decisions throughout the translation. For instance, the decision to represent an actual multilevel logic bit by two bits of bit logic required modification of the functions and procedures which operated on types. Then, the modification of the type functions led to other lessons.

One of the lessons was that range bounds and loop ranges all needed adjusting, and the logic of loops had to be changed to account for a two-bit representation of one actual bit. This realization confirms that type conversion is a major task in the translation effort. All functions and procedures of a design must be tuned to handle the changed types, otherwise the original procedures will fail because of the new logical representation of data.

During the tuning of procedures, other conflicts may occur. For overloaded procedures and functions, it may happen that a less general function may have the same basic types of parameters and return types as a more general function with the same name. Since this is not allowed, a data structure holding the previous and current declarations of functions must be maintained. In this manner, any conflicting less general function may be renamed and retained for use by the more general function. Calls from the system to the function would now go to the more general function for processing.

Finally, being restricted to using only JRS IDAS provided functions and procedures for ALU-type operations meant a translation function had to precede and follow any call to IDAS functions. This was due to the fact that the IDAS functions were originally written to process data_vectors which are two-level logic representations of data. Therefore, the four-level logic representations had to be

converted to two-level logic before the IDAS procedure and back to four-level logic after the IDAS procedure. Any logic levels other than '0' or '1' would be lost during this process; however, that was not expected to be a problem as data going through an ALU-type function should be stable at values of '0's and '1's.

4.3.3 Lessons From Architecture Conversion. The main lesson of the architecture conversion exercise was to understand a problem before looking at solution alternatives. Sometimes an apparent simple solution may be simply wrong. For the architecture conversion problem, the simplest and first considered option was to combine all blocks and processes in an architecture into one big process statement. However, careful analysis showed that this would destroy the parallelism natural with the multiple processes and blocks. The correct solution was then discovered which entailed creating new subarchitectures for each process. By doing this, the parallelism of the original blocks and processes was maintained.

Another lesson was that using multiple passes for certain types of processing problems makes the task easier to complete. In the case of architecture conversion, one pass was necessary to determine the declared ports and aliases and to collect the statements which are outside any process or block for later inclusion in the final architecture. A second pass is used to create a sequence of process

statements with the JRS required WAIT statement. A third pass creates an entity for each of the process statements. The fourth pass creates the architecture for each process. Finally, a fifth pass creates the new architecture which replaces the original architecture.

4.3.4 Lessons From Procedure Mapping. Procedure mapping is a difficult task. It requires an abstract, logical review of a subject procedure and a detailed knowledge of the procedures which are available to which it can be mapped. Only after reasoning about the functions performed, can one hope to successfully map the procedures.

Another lesson was that the procedures provided by IDAS are a proper subset of the functions possible for a VHDL design -- not all functions are included in the subset. This causes great problems when a nonmappable procedure is encountered during a manual mapping process -- an indication that redesign is required.

4.4 Review

This chapter has provided a demonstration of some of the concepts of translating standard VHDL into the JRS IDAS subset of VHDL. The manual implementations of CASE statement conversion, type conversion, architecture conversion, and procedure mapping were accomplished to demonstrate the feasibility of building Style-V to translate standard VHDL to the JRS subset. The lessons learned from these manual implementations were also discussed. The important points

which this chapter points out are that some translations like CASE statement and architecture conversion are easy, some like type conversion are difficult, and yet others like procedure mapping are presently impossible for an automated system.

V. Results, Conclusions, and Recommendations

5.0 Introduction

Chapter 1 of this thesis identified the need to translate VHDL written in the IEEE standard language to a subset of the VHDL language defined for a microcode generating tool. After stating the problem, Chapter 1 described the goals of the research effort -- to design a translator to fulfill the need. Another key section of Chapter 1 was the section discussing the assumptions about the translator, research, and IDAS tool.

Unfortunately, one of the key assumptions stated in Chapter 1 proved to be incorrect. The assumption was made that the IDAS tool would accept stylized VHDL and an Ada program and produce microcode for the VHDL design. However, despite the efforts of local experts, including some calls to the IDAS maker, JRS Research Laboratories, the IDAS would not process the files created to test the concept of generating microcode. This meant that any translation could not be tested against the IDAS tool. When this situation became apparent, the focus of the research effort turned to performing manual simulations to demonstrate the feasibility of the concepts required for a successful translation.

Chapter 2 consisted of a literature review to determine the current technologies for translating computer languages. The essential activities of lexical analysis and parsing were researched since any translator would need to perform these functions. Also, several examples of translators were reviewed.

Chapter 3 was a requirements analysis of the translator system (called Style-V). This analysis consisted of a domain analysis, language comparison, and Modern Structured Analysis. Domain analysis determined what composed a translator system. Once the components of the translator were identified, the analysis turned to the specific problem of translating the standard VHDL into a subset.

A careful comparison of the standard VHDL to the styled VHDL form was conducted. The results of the domain analysis and the VHDL comparison formed the basis for Modern Structured Analysis.

Modern Structured Analysis was used to determine the processes required in Style-V. The system context, external events, and event behaviors were defined and analyzed to determine "how" Style-V would perform the translation task. Enough detail was achieved to successfully perform desktop simulations of Style-V processes.

Chapter 4 documented the performance of the desktop simulations to demonstrate the feasibility of producing the Style-V translator. Not all processes were demonstrated,

but a representative subset of the processes provided enough information to determine if a Style-V translator could be fully automated. Chapter 4 also included the lessons learned from the manual implementations.

This chapter provides a summary of the results of this thesis effort, some conclusions based on the results, and recommendations based on the conclusions. The original goal of fully developing the Style-V translator was not realistic and the scope of the thesis was adjusted. In the final analysis, the problem of translating standard VHDL into the subset required by JRS for the IDAS tool was too large to complete in one thesis cycle. This thesis has produced an analysis of the problem and demonstrated how some translation tasks can be automated and why others cannot.

5.1 Results

Once the decision was made to design Style-V, one of the first efforts was to construct a simple example CPU to verify advertised IDAS capabilities. A successful demonstration of the IDAS would show a design could be written in the stylized form. It would also show the assumption that the IDAS would work was correct. However, the result of this effort showed the assumption was incorrect. Another result was the VHDL code for the sample system. The VHDL code for this system is included in Appendix A.

The next results of this research were those obtained from the analysis process. The domain analysis of the system provided an understanding of the parts composing any translator system. The decomposition of the translator system into its constituent parts is documented in Figures 3-1 through 3-5.

The comparison of IEEE standard VHDL to the stylized form required by the IDAS tool resulted in a set of mappings for the Style-V translator to satisfy. These mappings are identified in Figure 3-6. It was during the evaluation of these mappings that it became apparent that more than one thesis cycle would be required to complete a Style-V design and implementation.

After identifying the mappings required between standard and stylized VHDL, the Modern Structured Analysis of the system resulted in a description of Style-V in sufficient detail to complete the manual simulations. The products of the analysis consisted of the level one and level two data flow diagrams and a data dictionary for processes necessary to translate all defined mappings. The data flow diagrams and data dictionary are included Chapter 3.

During completion of the Chapter 3 processes, the amount of time required to simulate all mappings was deemed to be greater than the amount available. At that point, the mappings CASE_to_IF, Type Conversion, Architecture

Conversion, and Procedure Mapping were selected for manual implementation.

The manual implementation of CASE statement conversion resulted in a successful algorithm for converting CASE statements to IF-THEN-ELSIF-ELSE statements or IF-THEN statements. The CASE statement to IF statement mapping was the only mapping chosen for full implementation. The prototype CASE_to_IF conversion module is included in Appendix D of this thesis. However, since the IDAS did not perform as expected, it was not possible to test if the output of the module was acceptable or if further adjustments were required.

Architecture conversion was more challenging than CASE statement conversion. Like the CASE statement conversion, only one input file is manipulated at any one time. However, several passes are required over this file to produce subarchitectures. The results of the architecture conversion manual implementation were an algorithm to accomplish it, a new architecture at the same level in the system as the original architecture, and four architectures one level beneath the new architecture. The algorithm was provided in Chapter 4 and the resulting architectures are in Appendix C.

Type conversion proved to be quite a challenge, even for a manual implementation. One of the results of the effort was the use of bit logic to represent multilevel logic. Another result was a translated type definition

package for the Floating Point Application Specific Processor. The results of the translation of all the declarations and over fifty functions and procedures in this type definition package processed successfully through VHDL analysis.

The final task completed for the demonstration of concept effort was to attempt mapping procedures from the Floating Point Application Specific Processor (FPASP) to the IDAS. Eight of the FPASP processes were selected for mapping and seven mapped successfully. However, no mapping of IDAS procedures was found to satisfy the requirements of one FPASP procedure -- SR which shifted a bit out the right side of a word and a bit in the left side. The potential for this problem was identified in Sections 3.4.4 and 3.5.3.6, and the manual implementation confirmed the analysis. The manual implementation results are documented in Sections 4.2.4 and 4.3.4.

5.2 Conclusions and Recommendations

Regardless of the IDAS shortcomings, the design of the simple CPU was successful. The design simulates well through a VHDL system and may be useful in a course on VHDL or as an example system for future research. Since the VHDL code for the simple CPU runs in a VHDL system, a future researcher need not repeat this work, saving many hours of development effort.

When the IDAS failed to accept the simple CPU, a limitation of the tool was manifested. This revelation alerted

the holders of the tool that portions of it need further attention. They can now work with the vendor to have the tool completed. Unfortunately, this was not possible during this thesis effort.

The results of domain analysis provided a good foundation for understanding the parts of a translator system and how they interact. The techniques described in this thesis can be used to perform a domain analysis on other systems. Specifically, anyone beginning a research effort on translation systems need not repeat this work.

Modern Structured Analysis provided an appropriate means for defining the Style-V translator. This research adopted the view that translating languages is a functional problem; therefore, a functional approach to the analysis and design of the system was appropriate. The data flow diagrams and data dictionary techniques used for defining Style-V provide a good level of detail to facilitate further design. The techniques used for Style-V are equally applicable to like tasks. Future research efforts should consider using these same techniques.

CASE statement to IF statement conversion proved to be relatively simple to accomplish. Since only local knowledge was required to perform the translation (from the CASE keyword to the END CASE phrase), complex data structures and file manipulation was not required. In fact, only a single pass over a file was required to perform CASE statement to

IF statement conversion. The methods used during the manual and actual implementations of CASE statement conversion would be appropriate for other "single-pass" type problems.

For architecture conversion, use of data structures is required to successfully perform architecture conversion. For this effort, linked lists would suffice since a small number of elements would be in any one list. Adding items to the linked list requires order one time. Since all items in a list are usually used at the same time, it would take order n time to retrieve all n items. Researchers should study their problems and select the best data structures for solving them.

Once the use of bit logic for multilevel logic was determined, conversion of declarations and constants was relatively simple -- it was basically a textual substitution process. However, the conversion of functions (including procedures) was quite a different matter. The logic of internal loops had to be changed to properly manipulate the multiple bit representations of single logical bits. Therefore, loop indexes, assignment statement logic, and addition of "skip_this_time" logic was necessary. Also, type conflicts on previously compatible overloaded functions were possible. These problems were difficult to solve by hand and a full implementation of type conversion will require a great deal of further analysis and design. This task alone might warrant a future thesis effort.

The most difficult task of the Style-V was the mapping of user defined procedures into IDAS provided procedures. For cases when the functionality of each procedure is known and a combination of IDAS procedures (possibly one) can perform the user defined procedure's functionality, the mapping is possible. However, since the IDAS procedures are limited, it is not possible to map all possible procedures to the IDAS procedures. This thesis provided one example of a procedure which could not map to IDAS procedures.

In general, the automated mapping of procedures is a "hard" task. This thesis has shown that exhaustive testing is not a realistic answer to proving two procedures are functionally equivalent. With some risk, equivalence classes could reduce the number of tests required to ascertain if two procedures are functionally "equal" to one another. However, a program which would take two procedures with unrestricted numbers and types of parameters as input, which would produce meaningful test data, and which would determine if the procedures are equal is also a "hard" task. These tasks may be good thesis topics for those in the Artificial Intelligence specialty.

5.3 Epilogue

The restrictions of the IDAS were numerous. Other tools may force less serious restrictions. For those tools, the methods described in this thesis may prove sufficient.

Appendix A. Hayes CPU VHDL Design

This appendix contains the VHDL code representing the design of the simple CPU described by Hayes (Hayes, 1988:315). The design represented a simple model coded in the style of VHDL required by the JRS IDAS tool. Besides showing the potential for using JRS styled VHDL to create a design, this code may be useful for a course on VHDL. Future research efforts on VHDL translation may take advantage of this code and save many hours of development effort.

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  AC.VHD    (ACCUMULATOR COMPONENT)
--
--  AFFECTS:
--            BY:  DR.VHD  (DATA REGISTER COMPONENT)
--                 ALU.VHD  (ARITHMETIC LOGIC UNIT COMPONENT)
--                 CU.VHD  (CONTROL SIGNALS)
--                 CLOCK.VHD  (CLOCK)
--
--            ON:  ALU.VHD  (ARITHMETIC LOGIC UNIT COMPONENT)
--                 DR.VHD  (DATA REGISTER COMPONENT)
--
--  PURPOSE:  MODEL OF THE ACCUMULATOR OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--        14 OCT 91 (Ver 5)  Moved delay from behavioral IF stmts
--                           and eliminated wire delay.
------------------------------------------------------------------
------------------------------------------------------------------
library UTIL;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;


entity ACCUMULATOR is
    generic ( REG_DELAY  : time := 5 ns ;
              BUS_DELAY  : time := 1 ns ;
              WIRE_DELAY : time := 1 ns ;
              ALU_DELAY  : time := 7 ns ) ;
    port( DATA_FM_BUS : in DATA_VECTOR(15 downto 0) ;
          DATA_FM_ALU : in DATA_VECTOR(15 downto 0) ;
          DATA_TO_ALU : out DATA_VECTOR(15 downto 0) ;
          DATA_TO_BUS : out DATA_VECTOR(15 downto 0) ;
          C12         : in CONTROL ;
          C6          : in CONTROL ;
          C5          : in CONTROL ;
          C2          : in CONTROL ;
          C1          : in CONTROL ;
          C0          : in CONTROL ;
```

```
          CLK          : in CLOCK ) ;

end ACCUMULATOR;


architecture BEHAVIOR of ACCUMULATOR is

begin -- architecture ACCUMULATOR(BEHAVIOR)

   process
        variable AC_CONTENTS : data_vector(15 downto 0);
   begin

        wait until CLK = '1';
        if C6 = '1' then wait for REG_DELAY + BUS_DELAY;
        end if;

        if (C0 = '1' or C1 = '1' or C2 = '1') then
                behaviors.READ_REGISTER
                        (reg => AC_CONTENTS ,
                         out_port => AC_CONTENTS ,
                         name => "ACtoALU" );
                DATA_TO_ALU <= transport AC_CONTENTS after REG_DELAY;
               --wait for REG_DELAY + WIRE_DELAY + BUS_DELAY + WIRE_DELAY
                --        + ALU_DELAY + WIRE_DELAY ;
                wait for ALU_DELAY + REG_DELAY + BUS_DELAY;
                behaviors.WRITE_REGISTER
                        (in_port => DATA_FM_ALU ,
                         reg => AC_CONTENTS ,
                         name => "ALUtoAC" );
                DATA_TO_ALU <= transport AC_CONTENTS after REG_DELAY;
        end if;

        if C5 = '1' then
                behaviors.READ_REGISTER
                        (reg => AC_CONTENTS ,
                         out_port => AC_CONTENTS ,
                         name => "ACtoDR" );
                DATA_TO_BUS <= transport AC_CONTENTS after REG_DELAY;
        elsif CLK = '1' and C5 = '0' then
                DATA_TO_BUS <= transport b"0000000000000000";
        end if;

        if C6 = '1' then
             --wait for REG_DELAY + WIRE_DELAY + BUS_DELAY + WIRE_DELAY;
                behaviors.WRITE_REGISTER
                        (in_port => DATA_FM_BUS ,
                         reg => AC_CONTENTS ,
                         name => "ACfromDR" );
                DATA_TO_ALU <= transport AC_CONTENTS after REG_DELAY;
        end if;
```

```
        if CLK = '1' and C12 = '1' then
                behaviors.SHIFT_RIGHT_LOGICAL_1
                        (in_port => AC_CONTENTS ,
                         out_port => AC_CONTENTS ,
                         name => "RSHIFT_AC_FM_ALU" );
                DATA_TO_ALU <= transport AC_CONTENTS after REG_DELAY;
        end if;

    end process;

end BEHAVIOR;
```

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  AR.VHD    (ADDRESS REGISTER COMPONENT)
--
--  AFFECTS:
--            BY:  DR.VHD  (DATA REGISTER COMPONENT)
--                 PC.VHD  (PROGRAM COUNTER COMPONENT)
--                 CU.VHD  (CONTROL SIGNALS)
--                 CLOCK.VHD  (CLOCK)
--
--            ON:  MEMORY.VHD  (MAIN MEMORY COMPONENT)
--
--  PURPOSE:  MODEL OF THE ADDRESS REGISTER OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--        15 Oct 91 (Ver 5)  Removed wire delay.  Moved wait out
--              of behavioral IF statements.  Incorporated delay
--              in signal assignment statements.  Moved CLK test
--              to WAIT statement and removed it from the IFs.
------------------------------------------------------------------
------------------------------------------------------------------
library UTIL;
use UTIL.DATA_TYPES.all;
use JTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;


entity ADDRESS_REGISTER is

    generic ( REG_DELAY  : time := 5 ns;
              BUS_DELAY  : time := 1 ns );
    port( ADDR_FM_BUS : in DATA_VECTOR(15 downto 0) ;
          ADDR_TO_MEM : out ADDRESS_VECTOR(15 downto 0) ;
          C10         : in CONTROL ;
          C7          : in CONTROL ;
          C4          : in CONTROL ;
          C3          : in CONTROL ;
          CLK         : in CLOCK ) ;

end ADDRESS_REGISTER;
```

```
architecture BEHAVIOR of ADDRESS_REGISTER is

begin -- architecture ADDRESS_REG(BEHAVIOR)

   process
        variable AR_CONTENTS : address_vector(15 downto 0) ;
   begin
        wait until CLK = '1';
        wait for REG_DELAY + BUS_DELAY ;

        if C7 = '1' or C10 = '1' then
                behaviors.WRITE_REGISTER
                        (in_port => ADDR_FM_BUS ,
                         reg => AR_CONTENTS ,
                         name => "ADDR_FM_BUS" );
                AR_CONTENTS := AR_CONTENTS and b"0001111111111111";
        end if;

        ADDR_TO_MEM <= transport AR_CONTENTS;

   end process;

end BEHAVIOR;
```

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  ALU.VHD    (ARITHMETIC LOGIC UNIT COMPONENT)
--
--  AFFECTS:
--            BY:  AC.VHD  (ACCUMULATOR COMPONENT)
--                 DR.VHD  (DATA REGISTER COMPONENT)
--                 CU.VHD  (CONTROL SIGNALS)
--                 CLOCK.VHD  (CLOCK)
--
--            ON:  AC.VHD  (ACCUMULATOR COMPONENT)
--                 SR.VHD  (STATUS REGISTER COMPONENT)
--
--  PURPOSE:  MODEL OF THE ARITHMETIC LOGIC UNIT OF THE SIMPLE
--            CPU DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--       15 OCT 91 (Ver 5)  Eliminated wire delay.  Moved test
--                          for CLK = '1' to WAIT statement from IF.
--                          Moved wait from behavioral IF statements.
------------------------------------------------- ----------------
------------------------------------------------------------------

library UTIL;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;


entity ARITHMETIC_LOGIC_UNIT is

    generic ( REG_DELAY  : time := 5 ns ;
              BUS_DELAY  : time := 1 ns ;
              ALU_DELAY  : time := 7 ns ;
              CLK_PERIOD : time := 100 ns ) ;
    port' DATA_FM_AC  : in DATA_VECTOR(15 downto 0) ;
          DATA_FM_BUS : in DATA_VECTOR(15 downto 0) ;
          DATA_TO_AC  : out DATA_VECTOR(15 downto 0) ;
          ZERO_STAT   : out STATUS ;
          C2          : in CONTROL ;
          C1          : in CONTROL ;
          C0          : in CONTROL ;
```

```
          CLK          : in CLOCK ) ;

end ARITHMETIC_LOGIC_UNIT;


architecture BEHAVIOR of ARITHMETIC_LOGIC_UNIT is

begin    -- architecture ARITHMETIC_LOGIC_UNIT(BEHAVIOR)

   process
      variable ADD16_RESULT : data_vector(15 downto 0);
      variable ADD16_STATUS : status_type;
      variable AND16_RESULT : data_vector(15 downto 0);
      variable AND16_STATUS : status_type;
      variable NOT16_RESULT : data_vector(15 downto 0);
      variable NOT16_STATUS : status_type;
      variable IDLE_RESULT  : data_vector(15 downto 0);
      variable IDLE_STATUS  : status_type;
   begin

      wait until CLK = '1';
      wait for REG_DELAY + BUS_DELAY;

      if C0 = '1' then
        -- "add" behavior
        behaviors.ADD
                (in_port1 => DATA_FM_AC,
                 in_port2 => DATA_FM_BUS,
                 out_port => ADD16_RESULT,
                 status => ADD16_STATUS,
                 name => "ADD16" );
        DATA_TO_AC <= transport ADD16_RESULT after ALU_DELAY;
        if ADD16_STATUS(zero) = one then
                ZERO_STAT <= transport '1' after ALU_DELAY;
        else
                ZERO_STAT <= transport '0' after ALU_DELAY;
        end if;
      end if;

      if C1 = '1' then
        -- "and" behavior
        behaviors.LOGICAL_AND
                (in_port1 => DATA_FM_AC,
                 in_port2 => DATA_FM_BUS,
                 out_port => AND16_RESULT,
                 status => AND16_STATUS,
                 name => "AND16" );
        DATA_TO_AC <= transport AND16_RESULT after ALU_DELAY;
        if AND16_STATUS(zero) = one then
                ZERO_STAT <= transport '1' after ALU_DELAY;
        else
                ZERO_STAT <= transport '0' after ALU_DELAY;
```

```
      end if;
    end if;

    if C2 = '1' then
      -- "comp" behavior
      behaviors.LOGICAL_NOT
              (in_port => DATA_FM_AC,
               out_port => NOT16_RESULT,
               status => NOT16_STATUS,
               name => "NOT16" );
      DATA_TO_AC <= transport NOT16_RESULT after ALU_DELAY;
      if NOT16_STATUS(zero) = one then
              ZERO_STAT <= transport '1' after ALU_DELAY;
      else
              ZERO_STAT <= transport '0' after ALU_DELAY;
      end if;
    end if;
    if not ( C0 = '1' or C1 = '1' or C2 = '1' ) then
      -- idle state "behavior"
      behaviors.MOVE
              (in_port => DATA_FM_AC,
               out_port => IDLE_RESULT,
               status => IDLE_STATUS,
               name => "ALU_IDLE" );
      DATA_TO_AC <= transport IDLE_RESULT after ALU_DELAY;
      if IDLE_STATUS(zero) = one then
              ZERO_STAT <= transport '1' after ALU_DELAY;
      else
              ZERO_STAT <= transport '0' after ALU_DELAY;
      end if;
    end if;


  end process;

end BEHAVIOR;
```

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  CLOCK.VHD  (CLOCK COMPONENT)
--
--  AFFECTS:
--            BY:  none
--
--            ON:  AC.VHD  (ACCUMULATOR COMPONENT)
--                 ALU.VHD  (ARITHMETIC LOGIC UNIT COMPONENT)
--                 AR.VHD  (ADDRESS REGISTER COMPONENT)
--                 CU.VHD  (CONTROL UNIT COMPONENT)
--                 DR.VHD  (DATA REGISTER COMPONENT)
--                 IR.VHD  (INSTRUCTION REGISTER COMPONENT)
--                 PC.VHD  (PROGRAM COUNTER COMPONENT)
--                 MEMORY.VHD (MEMORY COMPONENT)
--
--  PURPOSE:  MODEL OF THE SYSTEM CLOCK OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  DATE:  24 JULY 1991
------------------------------------------------------------------
------------------------------------------------------------------

library Util;
use Util.DATA_TYPES.all;
use Util.BEHAVIORS, Util.BEHAVIORS.all;

entity sys_clock is
  port
    (ck     :  in  clock;
     sysclk :  out clock);
end sys_clock;


architecture BEHAVIOR of SYS_CLOCK is
begin
  process
    VARIABLE phase0_out_pulse :   clock;
    VARIABLE reset_ck_out_pulse :   clock;
  begin
    wait on ck;
    if ck = '1' then
      BEHAVIORS.pulse
        (system_clock => ck,
```

```
        out_pulse => phase0_out_pulse,
        Name => "phase0");
      sysclk <= transport phase0_out_pulse after  0 ns;
    end if;
    if ck = '0' then
      BEHAVIORS.pulse
        (system_clock => ck,
        out_pulse => reset_ck_out_pulse,
        Name => "reset_ck");
      sysclk <= transport reset_ck_out_pulse after  0 ns;
    end if;
  end process;
end BEHAVIOR;
```

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  CU.VHD  (CONTROL UNIT COMPONENT)
--
--  AFFECTS:
--            BY:  IR.VHD  (INSTRUCTION REGISTER COMPONENT)
--                 SR.VHD  (STATUS REGISTER COMPONENT)
--                 CLOCK.VHD  (CLOCK)
--
--            ON:  CU.VHD  (CONTROL SIGNALS)
--
--  PURPOSE:  MODEL OF THE CONTROL UNIT OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--        15 Oct 91 (Ver 5)  Removed redundant CLK test from IFs.
------------------------------------------------------------------
------------------------------------------------------------------

library UTIL;
use UTIL.DATA_TYPES.all;

entity CONTROL_UNIT is

    generic ( CU_DELAY    : time := 15 ns ) ;

    port( CU_INSTR        : in DATA_VECTOR(15 downto 0) ;
          ZSTAT_IN        : in STATUS ;
          CONTROLS        : out CONTROL_VECTOR(12 downto 0) ;
          CLK             : in CLOCK ) ;

end CONTROL_UNIT ;


architecture BEHAVIOR of CONTROL_UNIT is

begin      -- architecture CONTROL_UNIT(BEHAVIOR)

   process
   begin
        wait until CLK = '0';
```

```
------------------------------------------------------
--   PROCESS FETCH CYCLE
------------------------------------------------------

-- Process AR <- PC
CONTROLS <= transport b"0010000000000" after CU_DELAY;
wait until CLK'event and CLK = '0';
-- Process READ M
CONTROLS <= transport b"0000000001000" after CU_DELAY;
wait until CLK'event and CLK = '0';
-- Process PC <- PC + 1 and IR <- DR(OP)
CONTROLS <= transport b"0101000000000" after CU_DELAY;
wait until CLK'event and CLK = '0';

------------------------------------------------------
--   PROCESS EXECUTE CYCLE
------------------------------------------------------

-- Process LOAD Instruction
if CU_INSTR(15 downto 13) = b"000" then
   -- Process AR <- DR(ADR)
   CONTROLS <= transport b"0000010000000" after CU_DELAY;
   wait until CLK'event and CLK = '0';
   -- Process READ M
   CONTROLS <= transport b"0000000001000" after CU_DELAY;
   wait until CLK'event and CLK = '0';
   -- Process AC <- DR
   CONTROLS <= transport b"0000001000000" after CU_DELAY;
end if;

--Process STORE Instruction
if CU_INSTR(15 downto 13) = b"001" then
   -- Process AR <- DR(ADR)
   CONTROLS <= transport b"0000010000000" after CU_DELAY;
   wait until CLK'event and CLK = '0';
   -- Process DR <- AC
   CONTROLS <= transport b"0000000100000" after CU_DELAY;
   wait until CLK'event and CLK = '0';
   -- Process WRITE M
   CONTROLS <= transport b"0000000010000" after CU_DELAY;
end if;

-- Process ADD Instruction
if CU_INSTR(15 downto 13) = b"010" then
   -- Process AR <- DR(ADR)
   CONTROLS <= transport b"0000010000000" after CU_DELAY;
   wait until CLK'event and CLK = '0';
   -- Process READ M
   CONTROLS <= transport b"0000000001000" after CU_DELAY;
   wait until CLK'event and CLK = '0';
   -- Process AC <- AC + DR
   CONTROLS <= transport b"0000000000001" after CU_DELAY;
```

```
        end if;


        -- Process AND Instruction
        if CU_INSTR(15 downto 13) = b"011" then
           -- Process AR <- DR(ADR)
           CONTROLS <= transport b"0000010000000" after CU_DELAY;
           wait until CLK'event and CLK = '0';
           -- Process READ M
           CONTROLS <= transport b"0000000001000" after CU_DELAY;
           wait until CLK'event and CLK = '0';
           -- Process AC <- AC /\ DR
           CONTROLS <= transport b"0000000000010";
        end if;

        -- Process JUMP Instruction
        if CU_INSTR(15 downto 13) = b"100" then
           CONTROLS <= transport b"0000100000000" after CU_DELAY;
        end if;

        -- Process JUMPZ Instruction
        if CU_INSTR(15 downto 13) = b"101" and ZSTAT_IN = '1' then
           CONTROLS <= transport b"0000100000000" after CU_DELAY;
        end if;

        -- Process COMP Instruction
        if CU_INSTR(15 downto 13) = b"110" then
           CONTROLS <= transport b"0000000000100" after CU_DELAY;
        end if;

        -- Process RSHIFT Instruction
        if CU_INSTR(15 downto 13) = b"111" then
           CONTROLS <= transport b"1000000000000" after CU_DELAY;
        end if;

   end process;

end BEHAVIOR;
```

```
------------------------------------------------------------------
--  HAYES' SIMPLE CPU INSTANTIATED ARCHITECTURE
------------------------------------------------------------------
--  FILE:  CPU588_5.VHD     (588 CPU INSTANTIATED ARCHITECTURE)
--
--  AFFECTS:  COMPLETE HAYES' SIMPLE CPU (ALL COMPONENTS)
--
--  PURPOSE:  PROVIDE A MECHANISM TO DEFINE THE SIGNALS AND
--            COMPONENTS EXISTING IN THE SIMPLE CPU DESIGN.  THIS
--            ARCHITECTURE WAS USED TO DESCRIBE THE HAYES' SIMPLE CPU
--            (HAYES 1988:315).
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION: 5
--
--  DATE:  24 Sep 91
--              revised 13 Oct 91 to use library HAYES_CPU instead
--              of library PROC588.
--
--         15 Oct 91  (Ver 5)  Created a new higher level test bench.
------------------------------------------------------------------
------------------------------------------------------------------
library UTIL;
library HAYES_VER5;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;
use HAYES_VER5.MACHINE_DECLARATIONS.all;


entity CPU588 is
  port ( SYS_CLK : in CLOCK );
end CPU588;

architecture BEHAVIOR of CPU588 is

        ------------------------
        -- SIGNAL DECLARATIONS
        ------------------------

        signal CTRL_BUS     : CONTROL_VECTOR(12 downto 0) ;
        signal ALUsigAC     : DATA_VECTOR(15 downto 0) ;
        signal ACsigALU     : DATA_VECTOR(15 downto 0) ;
        signal ACsigBUS     : DATA_VECTOR(15 downto 0) ;
        signal DRsigBUS     : DATA_VECTOR(15 downto 0) ;
        signal MEMsigBUS    : DATA_VECTOR(15 downto 0) ;
        signal PCsigBUS     : ADDRESS_VECTOR(15 downto 0) ;
        signal BUSsig       : DATA_VECTOR(15 downto 0) ;
        signal ARsigMEM     : ADDRESS_VECTOR(15 downto 0) ;
        signal IRsigCU      : ADDRESS_VECTOR(15 downto 0) ;
```

```
signal CLOCKsig     : CLOCK ;
signal CLKpulseIN   : CLOCK ;
signal STATUSsig    : STATUS ;

-------------------------
-- COMPONENT DECLARATIONS
-------------------------

component ACCUMULATOR
port (DATA_FM_BUS   : in DATA_VECTOR(15 downto 0) ;
      DATA_FM_ALU   : in DATA_VECTOR(15 downto 0) ;
      DATA_TO_ALU   : out DATA_VECTOR(15 downto 0) ;
      DATA_TO_BUS   : out DATA_VECTOR(15 downto 0) ;
      C12           : in CONTROL ;
      C6            : in CONTROL ;
      C5            : in CONTROL ;
      C2            : in CONTROL ;
      C1            : in CONTROL ;
      C0            : in CONTROL ;
      CLK           : in CLOCK );
end component;

for all: ACCUMULATOR use entity
                  hayes_ver5.ACCUMULATOR(BEHAVIOR);

component ARITHMETIC_LOGIC_UNIT
port (DATA_FM_AC    : in DATA_VECTOR(15 downto 0) ;
      DATA_FM_BUS   : in DATA_VECTOR(15 downto 0) ;
      DATA_TO_AC    : out DATA_VECTOR(15 downto 0) ;
      ZERO_STAT     : out STATUS ;
      C2            : in CONTROL ;
      C1            : in CONTROL ;
      C0            : in CONTROL ;
      CLK           : in CLOCK );
end component;

for all: ARITHMETIC_LOGIC_UNIT use entity
                hayes_ver5.ARITHMETIC_LOGIC_UNIT(BEHAVIOR);


component ADDRESS_REGISTER
port (ADDR_FM_BUS   : in DATA_VECTOR(15 downto 0) ;
      ADDR_TO_MEM   : out ADDRESS_VECTOR(15 downto 0) ;
      C10           : in CONTROL ;
      C7            : in CONTROL ;
      C4            : in CONTROL ;
      C3            : in CONTROL ;
      CLK           : in CLOCK );
end component;

for all: ADDRESS_REGISTER use entity
                     hayes_ver5.ADDRESS_REGISTER(BEHAVIOR);
```

```
component SYS_CLOCK
port (CK     : in CLOCK ;
      SYSCLK : out CLOCK );
end component;

for all: SYS_CLOCK use entity hayes_ver5.SYS_CLOCK(BEHAVIOR);

component CONTROL_UNIT
port (CU_INSTR : in ADDRESS_VECTOR(15 downto 0) ;
      ZSTAT_IN : in STATUS ;
      CONTROLS : out CONTROL_VECTOR(12 downto 0) ;
      CLK      : in CLOCK );
end component;

for all: CONTROL_UNIT use entity
                      hayes_ver5.CONTROL_UNIT(BEHAVIOR);

component DATABUS
port (DATA_FM_AC  : in DATA_VECTOR(15 downto 0) ;
      DATA_FM_DR  : in DATA_VECTOR(15 downto 0) ;
      DATA_FM_MEM : in DATA_VECTOR(15 downto 0) ;
      ADDR_FM_PC  : in ADDRESS_VECTOR(15 downto 0) ;
      DATA_TO_AC  : out DATA_VECTOR(15 downto 0) ;
      DATA_TO_ALU : out DATA_VECTOR(15 downto 0) ;
      DATA_TO_AR  : out DATA_VECTOR(15 downto 0) ;
      DATA_TO_DR  : out DATA_VECTOR(15 downto 0) ;
      DATA_TO_IR  : out DATA_VECTOR(15 downto 0) ;
      DATA_TO_MEM : out DATA_VECTOR(15 downto 0) ;
      DATA_TO_PC  : out DATA_VECTOR(15 downto 0) ;
      CONTROLS    : in CONTROL_VECTOR(12 downto 0) ;
      CLK         : in CLOCK ) ;
end component;

for all: DATABUS use entity hayes_ver5.DATABUS(BEHAVIOR);

component DATA_REGISTER
port (DATA_FM_BUS  : in DATA_VECTOR(15 downto 0) ;
      DATA_TO_BUS  : out DATA_VECTOR(15 downto 0) ;
      C11          : in CONTROL ;
      C8           : in CONTROL ;
      C7           : in CONTROL ;
      C6           : in CONTROL ;
      C5           : in CONTROL ;
      C4           : in CONTROL ;
      C3           : in CONTROL ;
      C1           : in CONTROL ;
      C0           : in CONTROL ;
      CLK          : in CLOCK );
end component;

for all: DATA_REGISTER use entity
```

```
                        hayes_ver5.DATA_REGISTER(BEHAVIOR);

        component INSTRUCTION_REGISTER
        port (DATA_FM_BUS   : in DATA_VECTOR(15 downto 0) ;
              INST_TO_CU    : out ADDRESS_VECTOR(15 downto 0) ;
              C11           : in CONTROL ;
              CLK           : in CLOCK );
        end component;

        for all: INSTRUCTION_REGISTER use entity
                        hayes_ver5.INSTRUCTION_REGISTER(BEHAVIOR);

        component MAIN_MEMORY
        port (DATA_FM_BUS   : in DATA_VECTOR(15 downto 0) ;
              DATA_TO_BUS   : out DATA_VECTOR(15 downto 0) ;
              ADDR_FM_AR    : in ADDRESS_VECTOR(15 downto 0) ;
              C4            : in CONTROL ;
              C3            : in CONTROL ;
              CLK           : in CLOCK );
        end component;

        for all: MAIN_MEMORY use entity
                        hayes_ver5.MAIN_MEMORY(BEHAVIOR);

        component PROGRAM_COUNTER
        port (DATA_FM_BUS   : in DATA_VECTOR(15 downto 0) ;
              ADDR_TO_BUS   : out ADDRESS_VECTOR(15 downto 0) ;
              C10           : in CONTROL ;
              C9            : in CONTROL ;
              C8            : in CONTROL ;
              CLK           : in CLOCK );
        end component;

        for all: PROGRAM_COUNTER use entity
                        hayes_ver5.PR   RAM_COUNTER(BEHAVIOR);


begin     -- architecture CPU588(BEHAVIOR)

        ------------------------------
        -- COMPONENT CONFIGURATIONS
        ------------------------------

        AC_588: ACCUMULATOR
           port map (
                DATA_FM_BUS => BUSsig,
                DATA_FM_ALU => ALUsigAC ,
                DATA_TO_ALU => ACsigALU ,
                DATA_TO_BUS => ACsigBUS ,
                C12 => CTRL_BUS(12) ,
                C6 => CTRL_BUS(6) ,
```

```
        C5 => CTRL_BUS(5) ,
        C2 => CTRL_BUS(2) ,
        C1 => CTRL_BUS(1) ,
        C0 => CTRL_BUS(0) ,
        CLK => CLOCKsig );

ALU_588: ARITHMETIC_LOGIC_UNIT
   port map (
        DATA_FM_AC => ACsigALU ,
        DATA_FM_BUS => BUSsig ,
        DATA_TO_AC => ALUsigAC ,
        ZERO_STAT => STATUSsig ,
        C2 => CTRL_BUS(2) ,
        C1 => CTRL_BUS(1) ,
        C0 => CTRL_BUS(0) ,
        CLK => CLOCKsig );

AR_588: ADDRESS_REGISTER
   port map (
        ADDR_FM_BUS => BUSsig ,
        ADDR_TO_MEM => ARsigMEM ,
        C10 => CTRL_BUS(10) ,
        C7 => CTRL_BUS(7) ,
        C4 => CTRL_BUS(4) ,
        C3 => CTRL_BUS(3) ,
        CLK => CLOCKsig );

CLK_588: SYS_CLOCK
   port map (
        CK => CLKpulseIN ,
        SYSCLK => CLOCKsig );

CU_588: CONTROL_UNIT
   port map (
        CU_INSTR => IRsigCU ,
        ZSTAT_IN => STATUSsig ,
        CONTROLS => CTRL_BUS ,
        CLK => CLOCKsig );

DB_588: DATABUS
   port map (
        DATA_FM_AC => ACsigBUS ,
        DATA_FM_DR => DRsigBUS ,
        DATA_FM_MEM => MEMsigBUS ,
        ADDR_FM_PC => PCsigBUS ,
        DATA_TO_AC => BUSsig ,
        DATA_TO_ALU => BUSsig ,
        DATA_TO_AR => BUSsig ,
        DATA_TO_DR => BUSsig ,
        DATA_TO_IR => BUSsig ,
        DATA_TO_MEM => BUSsig ,
        DATA_TO_PC => BUSsig ,
```

```
            CONTROLS => CTRL_BUS ,
            CLK => CLOCKsig ) ;


      DR_588: DATA_REGISTER
         port map (
            DATA_FM_BUS => BUSsig ,
            DATA_TO_BUS => DRsigBUS ,
            C11 => CTRL_BUS(11) ,
            C8 => CTRL_BUS(8) ,
            C7 => CTRL_BUS(7) ,
            C6 => CTRL_BUS(6) ,
            C5 => CTRL_BUS(5) ,
            C4 => CTRL_BUS(4) ,
            C3 => CTRL_BUS(3) ,
            C1 => CTRL_BUS(1) ,
            C0 => CTRL_BUS(0) ,
            CLK => CLOCKsig );

      IR_588: INSTRUCTION_REGISTER
         port map (
            DATA_FM_BUS => BUSsig ,
            INST_TO_CU => IRsigCU ,
            C11 => CTRL_BUS(11) ,
            CLK => CLOCKsig );

      MEMORY_588: MAIN_MEMORY
         port map (
            DATA_FM_BUS => BUSsig ,
            DATA_TO_BUS => MEMsigBUS ,
            ADDR_FM_AR  => ARsigMEM ,
            C4 => CTRL_BUS(4) ,
            C3 => CTRL_BUS(3) ,
            CLK => CLOCKsig );

      PC_588: PROGRAM_COUNTER
         port map (
            DATA_FM_BUS => BUSsig ,
            ADDR_TO_BUS => PCsigBUS ,
            C10 => CTRL_BUS(10) ,
            C9 => CTRL_BUS(9) ,
            C8 => CTRL_BUS(8) ,
            CLK => CLOCKsig );


      process
      begin
            wait on SYS_CLK;
            CLKpulseIN <= transport SYS_CLK;
      end process;

end BEHAVIOR;
```

```
-------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
-------------------------------------------------------------------
--  FILE:  DB.VHD    (DATA BUS COMPONENT)
--
--  AFFECTS:
--            BY:  all components (except Control Unit, CU.VHD)
--
--            ON:  all components (except Control Unit, CU.VHD)
--
--  PURPOSE:  MODEL OF THE DATA BUS COMPONENT OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--       14 OCT 91 (Ver 5)  Moved wait out of behavioral IF
--                          statements and put signal asgnmt
--                          delay in sig asgnmt statements.
-------------------------------------------------------------------
-------------------------------------------------------------------
library UTIL;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;


entity DATABUS is

    generic ( REG_DELAY  : time := 5 ns;
              BUS_DELAY  : time := 1 ns;
              MEM_DELAY  : time := 10 ns );
    port( DATA_FM_AC  : in DATA_VECTOR(15 downto 0) ;
          DATA_FM_DR  : in DATA_VECTOR(15 downto 0) ;
          DATA_FM_MEM : in DATA_VECTOR(15 downto 0) ;
          ADDR_FM_PC  : in ADDRESS_VECTOR(15 downto 0) ;
          DATA_TO_AC  : out DATA_VECTOR(15 downto 0) ;
          DATA_TO_ALU : out DATA_VECTOR(15 downto 0) ;
          DATA_TO_AR  : out DATA_VECTOR(15 downto 0) ;
          DATA_TO_DR  : out DATA_VECTOR(15 downto 0) ;
          DATA_TO_IR  : out DATA_VECTOR(15 downto 0) ;
          DATA_TO_MEM : out DATA_VECTOR(15 downto 0) ;
          DATA_TO_PC  : out DATA_VECTOR(15 downto 0) ;
          CONTROLS    : in CONTROL_VECTOR(12 downto 0) ;
          CLK         : in CLOCK ) ;
```

```
end DATABUS;


architecture BEHAVIOR of DATABUS is

begin -- architecture DATABUS(BEHAVIOR)

   process
        variable DB_CONTENTS : DATA_VECTOR(15 downto 0) ;
   begin
        wait until CLK = '1' ;
        if CONTROLS(3) = '1' then                          --ver5
                wait for MEM_DELAY;
        else wait for REG_DELAY;
        end if;

        if CONTROLS(0) = '1' or CONTROLS(1) = '1' or
           CONTROLS(4) = '1' or CONTROLS(6) = '1' or
           CONTROLS(7) = '1' or CONTROLS(8) = '1' or
           CONTROLS(11) = '1' then
                behaviors.MOVE (
                                in_port => DATA_FM_DR ,
                                out_port => DB_CONTENTS ,
                                name => "DR bus_to ALU"
                                ) ;
        end if;

        if CONTROLS(3) = '1' then
                behaviors.MOVE (
                                in_port => DATA_FM_MEM ,
                                out_port => DB_CONTENTS ,
                                name => "MEM to BUS"
                                ) ;
        end if;

        if CONTROLS(5) = '1' then
                behaviors.MOVE (
                                in_port => DATA_FM_AC ,
                                out_port => DB_CONTENTS ,
                                name => "AC to BUS"
                                ) ;
        end if;

        if CONTROLS(10) = '1' then
                behaviors.MOVE (
                                in_port => ADDR_FM_PC ,
                                out_port => DB_CONTENTS ,
                                name => "PC to BUS"
                                ) ;
        end if;
```

```
        DATA_TO_DR <= transport DB_CONTENTS after BUS_DELAY;
        DATA_TO_AR <= transport DB_CONTENTS after BUS_DELAY;
        DATA_TO_AC <= transport DB_CONTENTS after BUS_DELAY;
        DATA_TO_IR <= transport DB_CONTENTS after BUS_DELAY;
        DATA_TO_PC <= transport DB_CONTENTS after BUS_DELAY;
        DATA_TO_ALU <= transport DB_CONTENTS after BUS_DELAY;
        DATA_TO_MEM <= transport DB_CONTENTS after BUS_DELAY;

   end process;

end BEHAVIOR;
```

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  DR.VHD    (DATA REGISTER COMPONENT)
--
--  AFFECTS:
--            BY:  AC.VHD  (ACCUMULATOR COMPONENT)
--                 MEMORY.VHD  (MAIN MEMORY COMPONENT)
--                 CU.VHD  (CONTROL SIGNALS)
--                 CLOCK.VHD  (CLOCK)
--
--            ON:  MEMORY.VHD  (MAIN MEMORY COMPONENT)
--                 AR.VHD  (ADDRESS REGISTER COMPONENT)
--
--  PURPOSE:  MODEL OF THE DATA REGISTER OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--       15 OCT 91 (Ver 5)  Deleted wire delay.  Moved wait from
--                          behavioral IFs.  Moved test for CLK = '1'
--                          from IFs to first WAIT statement.
------------------------------------------------------------------
------------------------------------------------------------------
library UTIL;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;


entity DATA_REGISTER is

    generic ( REG_DELAY  : time := 5 ns;
              BUS_DELAY  : time := 1 ns;
              MEM_DELAY  : time := 10 ns ) ;
    port( DATA_FM_BUS : in DATA_VECTOR(15 downto 0) ;
          DATA_TO_BUS : out DATA_VECTOR(15 downto 0) ;
          C11         : in CONTROL ;
          C8          : in CONTROL ;
          C7          : in CONTROL ;
          C6          : in CONTROL ;
          C5          : in CONTROL ;
          C4          : in CONTROL ;
          C3          : in CONTROL ;
```

```
          C1          : in CONTROL ;
          C0          : in CONTROL ;
          CLK         : in CLOCK ) ;

end DATA_REGISTER;



architecture BEHAVIOR of DATA_REGISTER is

begin     -- architecture DATA_REGISTER(BEHAVIOR)

   process
        variable DR_CONTENTS : DATA_VECTOR(15 downto 0);
   begin

        wait until CLK = '1';
        if C3 = '1' then wait for MEM_DELAY + BUS_DELAY;
        elsif C5 = '1' then wait for REG_DELAY + BUS_DELAY;
        end if;

        if (C0 = '1' or C1 = '1' or C4 = '1' or
           C6 = '1' or C7 = '1' or C8 = '1' or C11 = '1')  then
                behaviors.READ_REGISTER
                        ( out_port => DR_CONTENTS ,
                          reg => DR_CONTENTS ,
                          name => "DRtoBUS" );
                DATA_TO_BUS <= transport DR_CONTENTS after REG_DELAY;
        elsif CLK = '1' and not (C0 = '1' or C1 = '1' or C4 = '1' or
                   C6 = '1' or C7 = '1' or C8 = '1' or C11 = '1')  then
                DATA_TO_BUS <= transport b"0000000000000000"
                                                        after REG_DELAY;
        end if;

        if C3 = '1' then
                behaviors.WRITE_REGISTER
                        (reg => DR_CONTENTS ,
                         in_port => DATA_FM_BUS ,
                         name => "DRgetsMEM" );
        end if;

        if C5 = '1' then
                behaviors.WRITE_REGISTER
                        (reg => DR_CONTENTS ,
                         in_port => DATA_FM_BUS ,
                         name => "DRgetsAC" ) ;
        end if;

   end process;

end BEHAVIOR;
```

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  IR.VHD  (INSTRUCTION REGISTER COMPONENT)
--
--  AFFECTS:
--            BY:  DR.VHD  (DATA REGISTER COMPONENT)
--                 CU.VHD  (CONTROL SIGNALS)
--                 CLOCK.VHD  (CLOCK)
--
--            ON:  CU.VHD  (CONTROL UNIT COMPONENT)
--
--  PURPOSE:  MODEL OF THE INSTRUCTION REGISTER OF THE SIMPLE
--            CPU DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--        15 OCT 91 (Ver 5)  Moved test for CLK = '1' to WAIT
--                           statement from IF statement.  Eliminated
--                           wire delay.  Moved wait out of behavioral
--                           IF statement.
------------------------------------------------------------------
------------------------------------------------------------------

library UTIL;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;


entity INSTRUCTION_REGISTER is

    generic ( REG_DELAY  : time := 5 ns;
              BUS_DELAY  : time := 1 ns );
    port( DATA_FM_BUS : in DATA_VECTOR(15 downto 0) ;
          INST_TO_CU  : out ADDRESS_VECTOR(15 downto 0) ;
          C11         : in CONTROL ;
          CLK         : in CLOCK ) ;

end INSTRUCTION_REGISTER;


architecture BEHAVIOR of INSTRUCTION_REGISTER is
```

```
begin     -- architecture INSTRUCTION_REGISTER(BEHAVIOR)

   process
        variable IR_CONTENTS : DATA_VECTOR(15 downto 0);
   begin
        wait until CLK = '1';
        wait for REG_DELAY + BUS_DELAY;

        if C11 = '1' then
                behaviors.WRITE_REGISTER
                        (in_port => DATA_FM_BUS ,
                         reg => IR_CONTENTS ,
                         name => "IRgetsDR" );
                behaviors.READ_REGISTER
                        (reg => IR_CONTENTS ,
                         out_port => IR_CONTENTS ,
                         name => "CUgetsIR" );
                INST_TO_CU <= transport IR_CONTENTS after REG_DELAY;
        else INST_TO_CU <= transport b"0000000000000000";
        end if;

   end process;

end BEHAVIOR;
```

```
--------------------------------------------------------------------
--    MACHINE DECLARATIONS FOR HAYES' CPU
--------------------------------------------------------------------
--  FILE:  MACHDECL.VHD     (MACHINE DECLARATION PACKAGE)
--
--  AFFECTS:
--            BY:  none
--
--            ON:  AC.VHD  (ACCUMULATOR COMPONENT)
--                 ALU.VHD  (ARITHMETIC LOGIC UNIT COMPONENT)
--                 AR.VHD  (ADDRESS REGISTER COMPONENT)
--                 DR.VHD  (DATA REGISTER COMPONENT)
--                 IR.VHD  (INSTRUCTION REGISTER COMPONENT)
--                 MEMORY.VHD  (MAIN MEMORY COMPONENT)
--                 PC.VHD  (PROGRAM COUNTER COMPONENT)
--
--  PURPOSE:  SUPPLY THE MACHINE DEPENDENT TYPE DECLARATIONS AND
--            ANY RESOLUTION FUNCTIONS NEEDED FOR THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  HAYES' SIMPLE CPU
--            WAS USED AS A TEST CASE FOR USING THE JRS IDAS TO
--            AUTOMATICALLY GENERATE MICROCODE FOR THE DESIGNED
--            HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  24 SEP 91  (Ver 4)
--         15 OCT 91  (Ver 5)  Moved memory initialization function to a
--                      separate package since Package Machine_Declara-
--                      tions only 1) defines a constant cycle_length,
--                      2) defines machine specific bit_vectors, and
--                      3) defines bus subtypes and their resolution
--                      functions.
--------------------------------------------------------------------
--------------------------------------------------------------------

library UTIL;
use UTIL.DATA_TYPES.all;

package MACHINE_DECLARATIONS is

  constant CLK_PERIOD : time := 100 ns;

  type CONTROL_12_DOWNTO_0_VECTOR is
          array(integer range <>) of CONTROL_VECTOR(12 downto 0);

end MACHINE_DECLARATIONS;
```

```
------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------
--  FILE:  MEMORY5.VHD  (MAIN MEMORY COMPONENT)
--
--  AFFECTS:
--            BY:  DR.VHD  (DATA REGISTER COMPONENT)
--                 AR.VHD  (ADDRESS REGISTER COMPONENT)
--                 CU.VHD  (CONTROL SIGNALS)
--
--            ON:  DR.VHD  (DATA REGISTER COMPONENT)
--
--  PURPOSE:  MODEL OF THE MEMORY PORTION OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 Version 4
--         revised 24 SEP 91 to use library HAYES_CPU instead
--                 of library PROC5v8.
--         14 Oct 91  Version 5:  Moved delay from behavioral IF
--                      statements.  Eliminated wire delay.
--                      moved test for CLK = '1' to WAIT from IF.
--                      Changed working library from HAYES_CPU to
--                      HAYES_VER5.
------------------------------------------------------------
------------------------------------------------------------
library UTIL;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;
use UTIL.ATTRIBUTE_DECLARATIONS.all;
library HAYES_VER5;
use hayes_ver5.MACHINE_DECLARATIONS;
use hayes_ver5.MACHINE_DECLARATIONS.all;
use hayes_ver5.MEMORY_LOADER;
use hayes_ver5.MEMORY_LOADER.all;

entity MAIN_MEMORY is

    generic ( REG_DELAY  : time := 5 ns;
              BUS_DELAY  : time := 1 ns;
              MEM_DELAY  : time := 10 ns );
    port( DATA_FM_BUS  : in DATA_VECTOR(15 downto 0) ;
          DATA_TO_BUS  : out DATA_VECTOR(15 downto 0) ;
```

```
        ADDR_FM_AR    : in ADDRESS_VECTOR(15 downto 0) ;
        C4            : in CONTROL ;
        C3            : in CONTROL ;
        CLK           : in CLOCK );

    attribute STARTING_ADDRESS of MAIN_MEMORY: entity is 0 ;
    attribute ENDING_ADDRESS of MAIN_MEMORY:  entity is 8192 ;

end MAIN_MEMORY;


architecture BEHAVIOR of MAIN_MEMORY is

begin     -- architecture MAIN_MEMORY(BEHAVIOR)

   process
        variable TEMP_MEM  : data_15_downto_0_vector(8192 downto 0);
        variable MEM_SPACE : data_15_downto_0_vector(8192 downto 0)
                                                  := LOAD_MEMORY(TEMP_MEM);
        variable MEM_CONTENTS : data_vector(15 downto 0) ;
   begin
        wait until CLK = '1';

        if C4 = '1' then wait for REG_DELAY + BUS_DELAY;
        end if;

        if C3 = '1' then
                behaviors.READ
                        (memory => MEM_SPACE ,
                         address_port => ADDR_FM_AR ,
                         data_port => MEM_CONTENTS ,
                         name => "READmem" );
                DATA_TO_BUS <= transport MEM_CONTENTS after MEM_DELAY;
        elsif C3 = '0' then
                DATA_TO_BUS <= transport b"0000000000000000" ;
        end if;

        if C4 = '1' then
                behaviors.WRITE
                        (memory => MEM_SPACE ,
                         address_port => ADDR_FM_AR ,
                         data_port => DATA_FM_BUS ,
                         name => "WRITEmem" );
        end if;

   end process;

end BEHAVIOR;
```

```
--------------------------------------------------------------------
--    MACHINE DECLARATIONS FOR HAYES' CPU
--------------------------------------------------------------------
--  FILE:  LOADMEM4.VHD     (MEMORY LOADER PACKAGE)
--
--  AFFECTS:
--            BY:  none
--
--            ON:  MEMORY.VHD  (MAIN MEMORY COMPONENT)
--
--  PURPOSE:  LOAD THE INITIAL VALUES FOR THE MEMORY OF THE CPU
--            DESCRIBED BY (HAYES 1988:315).  HAYES' SIMPLE CPU
--            WAS USED AS A TEST CASE FOR USING THE JRS IDAS TO
--            AUTOMATICALLY GENERATE MICROCODE FOR THE DESIGNED
--            HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  1  (SEPARATED FROM MACHDECL4.VHD TO STYLIZE THE
--                MACHINE_DECLARATIONS PACKAGE FOR IDAS INPUT)
--
--  DATE:  23 SEP 91
--------------------------------------------------------------------
--------------------------------------------------------------------

library UTIL;
use UTIL.DATA_TYPES.all;

package MEMORY_LOADER is

  function LOAD_MEMORY (MEM_SPACE : in DATA_15_DOWNTO_0_VECTOR)
                                        return data_15_downto_0_vector;

end MEMORY_LOADER;


package body MEMORY_LOADER is


   function LOAD_MEMORY (MEM_SPACE : in DATA_15_DOWNTO_0_VECTOR)
                                 return data_15_downto_0_vector is
        variable M_SPACE : data_15_downto_0_vector(MEM_SPACE'range)
                                                        := MEM_SPACE;
   begin

        -- Initialize 8K Memory
        for i in M_SPACE'range loop
               M_SPACE(i) := b"0000000000000000";
        end loop;

        -- LOAD AC with contents of location 100 (=4)
```

```
        M_SPACE(0)   := b"0000000001100100";
        M_SPACE(100) := b"0000000000000100";

        -- ADD contents of location 101 (=2) to contents of AC (=4)
        M_SPACE(1)   := b"0100000001100101";
        M_SPACE(101) := b"0000000000000010";

        -- STORE contents of AC in location 102  (should be = 6)
        M_SPACE(2)   := b"0010000001100110";

        -- AND contents of AC (=6) with contents of location 103 (=2)
        M_SPACE(3)   := b"0110000001100111";
        M_SPACE(103) := b"0000000000000010";

        -- STORE contents of AC in location 104
        M_SPACE(4)   := b"0010000001101000";

        -- JUMP to location 10
        M_SPACE(5)   := b"1000000000001010";

        -- LOAD AC with contents of location 7 (=0)
        M_SPACE(10)  := b"0000000000000111";

        -- JUMPZ to location 15
        M_SPACE(11)  := b"1010000000001111";

        -- COMP contents of AC
        M_SPACE(15)  := b"1100000000000000";

        -- STORE AC in location 105 (should = 2**16)
        M_SPACE(16)  := b"0010000001101001";

        -- RSHIFT contents of AC
        M_SPACE(17)  := b"1110000000000000";

        -- STORE AC in location 106 (should = 2**15)
        M_SPACE(18)  := b"0010000001101010";

        return M_SPACE;

    end LOAD_MEMORY;


end MEMORY_LOADER;
```

```
------------------------------------------------------------------
--  HAYES' CPU  --  MODELED PROCEDURALLY IN VHDL
------------------------------------------------------------------
--  FILE:  PC5.VHD  (PROGRAM COUNTER COMPONENT)
--
--  AFFECTS:
--            BY:  DR.VHD  (DATA REGISTER COMPONENT)
--                 CU.VHD  (CONTROL SIGNALS)
--                 CLOCK.VHD  (CLOCK)
--
--            ON:  AR.VHD  (ADDRESS REGISTER COMPONENT)
--
--  PURPOSE:  MODEL OF THE PROGRAM COUNTER OF THE SIMPLE CPU
--            DESCRIBED BY (HAYES 1988:315).  THIS MODEL
--            COMBINED WITH MODELS OF THE OTHER COMPONENTS OF
--            HAYES' SIMPLE CPU WAS USED AS A TEST CASE FOR
--            USING THE JRS IDAS TO AUTOMATICALLY GENERATE MICRO-
--            CODE FOR THE DESIGNED HARDWARE.
--
--  AUTHOR:  CAPT DENNIS A. RUMBLEY
--           AFIT/ENG
--
--  VERSION:  5
--
--  DATE:  4 SEP 91 (Ver 4)
--       15 OCT 91 (Ver 5)    Eliminated wire delay.  Moved wait
--                      from behavioral IF statements.  Moved test
--                      for CLK = '1' to WAIT statement from IF.
------------------------------------------------------------------
------------------------------------------------------------------
library UTIL;
use UTIL.DATA_TYPES.all;
use UTIL.BEHAVIORS;
use UTIL.BEHAVIORS.all;


entity PROGRAM_COUNTER is

    generic ( REG_DELAY  : time := 5 ns;
              BUS_DELAY  : time := 1 ns );
    port( DATA_FM_BUS : in DATA_VECTOR(15 downto 0) ;
          ADDR_TO_BUS : out ADDRESS_VECTOR(15 downto 0) ;
          C10         : in CONTROL ;
          C9          : in CONTROL ;
          C8          : in CONTROL ;
          CLK         : in CLOCK ) ;

end PROGRAM_COUNTER;


architecture BEHAVIOR of PROGRAM_COUNTER is
```

```
begin     -- architecture PROGRAM_COUNTER(BEHAVIOR)

   process
        variable PC_CONTENTS : address_vector(15 downto 0);
   begin
        wait until CLK = '1';
        if C8 = '1' then wait for REG_DELAY + BUS_DELAY;
        end if;

        if C8 = '1' then
                behaviors.WRITE_REGISTER
                        (in_port => DATA_FM_BUS ,
                         reg => PC_CONTENTS ,
                         name => "PCgetsDR" );
        end if;

        if C9 = '1' then
                behaviors.INC
                        (in_port => PC_CONTENTS ,
                         out_port => PC_CONTENTS ,
                         name => "incPC" );
        end if;

        if C10 = '1' then
                behaviors.READ_REGISTER
                        (reg => PC_CONTENTS ,
                         out_port => PC_CONTENTS ,
                         name => "PCtoAR" );
                ADDR_TO_BUS <= transport PC_CONTENTS after REG_DELAY;
        else
                ADDR_TO_BUS <= transport b"0000000000000000"
                                                    after REG_DELAY;
        end if;

   end process;

end BEHAVIOR;
```

```
------------------------------------------------------------------
--  TEST BENCH for HAYES SIMPLE CPU
------------------------------------------------------------------
--  PURPOSE:  Provide the test bench entity for the simple
--      CPU described by (Hayes 1988:315).  The test bench
--      provides the clock and process duration processes for
--      control of this VHDL model.  The test bench is not
--      and need not be JRS styled VHDL.
--
--  FILE:  HAYES_TEST.VHD
--
--  AUTHOR: Capt Dennis A. Rumbley
--          AFIT Masters Student
--          AFIT/ENG  GCS-91D
--
--  DATE:  15 OCT 91
--
--  VERSION:  0
------------------------------------------------------------------
library UTIL;
use UTIL.Data_Types.all;
library HAYES_VER5;
use HAYES_VER5.MACHINE_DECLARATIONS.all;


entity HAYES_CPU is
end HAYES_CPU;

architecture TEST_BENCH of HAYES_CPU is

    signal SYS_CLOCK_SIG : CLOCK;

    component CPU588_COMP
        port ( SYS_CLK : in CLOCK );
    end component;

    for all: CPU588_COMP use entity HAYES_VER5.CPU588(BEHAVIOR);

begin   -- architecture HAYES_CPU(TEST_BENCH)

CPU588: CPU588_COMP
        port map (SYS_CLK => SYS_CLOCK_SIG );


make_clock:  process
begin
        SYS_CLOCK_SIG <= not SYS_CLOCK_SIG
                after hayes_ver5.MACHINE_DECLARATIONS.CLK_PERIOD/2;
        wait for hayes_ver5.MACHINE_DECLARATIONS.CLK_PERIOD/2 ;
end process ;

start_stop: process
```

```
begin
        wait for 7000 ns ;
        assert false
        report "End of Simulation"
        severity error;
end process;

end TEST_BENCH;
```

Appendix B
FPASP Design Code

The Floating Point Application Specific Processor (FPASP) is a United States Air Force research and development effort. The design is therefore proprietary to the United States Air Force. The code of this appendix is part of that design, and therefore must be restricted from distribution to the general public.

The FPASP code which would normally be found in this appendix is maintained in Volume II of this thesis. The controlling office for release of the FPASP information in Volume II is:

Rome Laboratory

RL/OCTS

Griffiss AFB, NY

Appendix C
Stylized FPASP Design Code

The Floating Point Application Specific Processor (FPASP) is a United States Air Force research and development effort. The design is therefore proprietary to the United States Air Force. The translated code of this appendix used part of that design, and therefore must be restricted from distribution to the general public.

The translated FPASP code which would normally be found in this appendix is maintained in Volume II of this thesis. The controlling office for release of the results of Style-V translated FPASP code in Volume II is:

Rome Laboratory

RL/OCTS

Griffiss AFB, NY

Appendix D
CASE_TO_IF Conversion Module

This appendix contains the pseudo code and C code for the CASE_TO_IF conversion module of the Style-V translator and some test results. Sample output is located in Section 1 of Appendix C.

SECTION 1: PSEUDO CODE

```
/* Pseudo code for the CASE statement to IF statement conversion
   function required for the STYLE-V translator.  This module provides
   one of the conversions necessary to convert IEEE standard VHDL into
   the JRS stylized VHDL for input into the Integrated Design Automation
   System.
   ------------------------------------------------------------------------
   It may be possible to implement the following functions using the LEX
   tool.
*/


/*
  ************************************************************************
  *  The following function processes an input file to the point where
  *  a CASE statement is found.  Control is then passed to a special
  *  function designed to convert the CASE statement to an IF statement.
  *  Once the conversion of the CASE statement is complete, processing
  *  returns to this function.  Each CASE statement is processed
  *  in this manner for the entire file.  The resulting output file
  *  is named NC_orig_file_name to identify the changed file.  Control
  *  then passes to the calling function.  For example, this function
  *  will handle all statements represented as !! below and would call
  *  CONV_CASE_TO_IF to handle the CASE statements.  (Note: comments are
  *  passed through and do not affect the translation.)
  *
  *     BEGIN FILE
  *     !!
  *     !!
  *     !!
  *     CASE
  *        $$
  *        $$
  *        CASE
  *           $$
  *           $$
  *        END CASE
  *        $$
  *     END CASE
  *     !!
  *     !!
  *     CASE
  *        $$
  *        $$
  *     END CASE
  *     !!
  *     END FILE
  ************************************************************************
 */
 /*
```

```
function CASE_TO_IF_CONV (in_file)
        open in_file
        create out_file
        move_white_space (in_file, out_file)
       read_word (in_file, the_word)
        while not eof in_file
                if the_word = "case" or "CASE"
                        then CONV_CASE_TO_IF
                        else write_to_file (out_file, the_word)
                end if
                move_white_space (in_file, out_file)
                read_word (in_file, the_word)
        end while not eof in_file
        close in_file
        close out_file
end CASE_TO_IF_CONV


function CONV_CASE_TO_IF
        ---------------------
        --  "case " => "if "
        ---------------------
        write_to_file (out_file, "if ")
        -------------------------------------
        -- Pass conditional argument through
        -------------------------------------
        move_white_space (in_file, out_file)
        read_word (in_file, the_word)
        COND_ARG := the_word
        write_to_file (out_file, COND_ARG)
        ------------ ------------------------
        --  Pass nothing through for "is "
        -------------------------------------
        move_white_space (in_file, out_file)
        read_word (in_file, the_word)
        if the_word not= "is " or "IS "
                then print error message
                     set global error flag
                     exit CONV_CASE_TO_IF
        end if
        -------------------------------------
        --  "when " becomes "= "
        -------------------------------------
        move_white_space (in_file, out_file)
        read_word (in_file, the_word)
        if the_word = "when " or "WHEN "
                then write_to_file ("= ")
                else print error message
                     set global error flag
                     exit CONV_CASE_TO_IF
        end if
        -------------------------------------
```

```
 -- Pass conditional value through
 -------------------------------------
 move_white_space (in_file, out_file)
 read_word (in_file, the_word)
write_to_file (out_file, the_word)
 -------------------------------------
 -- "=> " becomes "then "
 -------------------------------------
 move_white_space (in_file, out_file)
 read_word (in_file, the_word)
 if the_word = "=> "
         then write_to_file (out_file, "then ")
         else print error message
              set global error flag
              exit CONV_CASE_TO_IF
 end if
 ---------------------------------------------
 -- Process when clause statements to out file
 ---------------------------------------------
 prev_word := the_word
              (** needed to exit case processing **)
 move_white_space (in_file, out_file)
 read_word (in_file, the_word)
 while the_word not= "when " or "WHEN "
         if the_word not= "case" or "CASE"
                 then write_to_file (out_file, the_word)
                 else if prev_word = "end " or "END "
                         then exit CONV_CASE_TO_IF
                         else CONV_CASE_TO_IF
         end if
         prev_word := the_word
         move_white_space (in_file, out_file)
         read_word (in_file, the_word)
 end while the_word not= "when " or "WHEN "
 ---------------------------------------
 -- Pass nothing when next "when " seen
 ---------------------------------------
 loop  (* process when statements until done *)
     prev_word := the_word
     --------------------------------
     -- Check if next word is "others"
     --------------------------------
     move_white_space (in_file, out_file)
     read_word (in_file, the_word)
     if the_word not= "others " or "OTHERS "
         then write_to_file (out_file, "elsif ")
              write_to_file (out_file, COND_ARG)
              write_to_file (out_file, " = ")
              write_to_file (out_file, the_word)
              -------------------------------------
              -- "=> " becomes "then "
              -------------------------------------
```

```
                    move_white_space (in_file, out_file)
                    read_word (in_file, the_word)
                    if the_word = "=> "
                       then write_to_file (out_file, "then ")
                       else print error message
                            set global error flag
                             exit CONV_CASE_TO_IF
                    end if
                else write_to_file (out_file, "else "
                    move_white_space (in_file, out_file)
                    read_word (in_file, the_word)
                    if the_word not= "=> "
                       then print error message
                            set global error flag
                            exit CONV_CASE_TO_IF
                    end if
            end if
            ------------------------------------------
            --  Process when clause statements
            ------------------------------------------
            prev_word := the_word
            move_white_space (in_file, out_file)
            read_word (in_file, the_word)
            while the_word not= "when " or "WHEN "
                if the_word not= "case" or "CASE"
                        then write_to_file (out_file, the_word)
                        else if prev_word = "end " or "END "
                                then exit CONV_CASE_TO_IF
                                else CONV_CASE_TO_IF
                end if
                prev_word := the_word
                move_white_space (in_file, out_file)
                read_word (in_file, the_word)
            end while the_word not= "when " or "WHEN "
        end loop  (* process when stmts until done *)

end CONV_CASE_TO_IF


function READ_WORD (in_file, the_word)
        the_word := ""
        get_char (in_file, the_char)
        while the_char = "-"
                (**check for comment**)
                temp_char := the_char
                get_char (in_file, the_char)
                if the_char = "-"
                        then the_word := the_word + "--"
                             while not end_of_line
                                get_char (in_file, the_char)
                                the_word := the_word+the_char
                             end while
```

```
                                write_word (out_file, the_word)
                                get_char (in_file, the_char)
                            else the_word := the_word + "-"
                    end if
            end while
           while not eof and the_char not= blank or tab or new-line
                    the_word := the_word+the_char
                    get_char (in_file, the_char)
            end while
            the_word := the_word+blank
    end READ_WORD

    function MOVE_WHITE_SPACE (in_file, out_file)
            get_char (in_file, the_char)
            while not eof and the_char = blank or tab or new-line
                    put_char (out_file, the_char)
                    get_char (in_file, the_char)
            end while
            if eof (in_file)
                    then terminate successfully
                    else (* move file pointer back one character *)
                    return_char (in_file, the_char)
    end MOVE_WHITE_SPACE

    function WRITE_TO_FILE (out_file, the_word)
            write_word (out_file, the_word)
            move_white_space (in_file, out_file)
    end WRITE_TO_FILE

*/
```

SECTION 2: C CODE

```
/*****************************************************************************
*****************************************************************************
************************* C A S E T O I F ***********************************
*****************************************************************************
*****************************************************************************

  ****************************************************************************
  *  PURPOSE:  Copy an input file to an output file except that all case
  *     statements of the input file will be converted to equivalent
  *     if-elsif-else statements.  This program will handle all files
  *     that match the following abstract model.  Note that CASETOIF
  *     handles simple and nested case statements.
  *
  *     BEGIN FILE
  *     !!
  *     !!
  *     !!
  *     CASE
  *        $$
  *        $$
  *        CASE
  *           $$
  *           $$
  *        END CASE
  *        $$
  *     END CASE
  *     !!
  *     !!
  *     CASE
  *        $$
  *        $$
  *     END CASE
  *     !!
  *     END FILE
  ****************************************************************************
*/
        #include <io.h>
        #include <stdio.h>
        #include <string.h>

        /* GLOBAL VARIABLES */
        int pgm_success = 1;
        char *the_word = "";

        /*************************************************************
         *  FUNCTION:  FATAL_CASE_WORD_ERROR
         *************************************************************
         *  PURPOSE:  Print error message for fatal error when an
```

```
 *            incorrect word for the context is found.
 *            Also, sets the global pgm_success flag to 0.
 *
 *  CALLED BY:  conv_case_to_if
 *
 *  CALLS:      none
 *
 *  PARAMETERS:
 *              error_word -- a character string which was
 *                      the offending word.
 *
 *  LOCAL VARIABLES:
 *              in_str[132] : string used to load input param
 *                      string.  Used to limit the number of
 *                      pointer strings because of overwrite
 *                      problems when using pointer strings.
 *
 *  GLOBAL VARIABLES:
 *              pgm_success -- used to show a condition has
 *                      occurred which causes abnormal
 *                      program termination.
 *
 *  AUTHOR:     Capt Dennis A. Rumbley
 *              AFIT Masters Student
 *              AFIT/ENG, GCS-91D
 *
 *  DATE:       20 Sep 91
 *
 *  VERSION:    1
 ***************************************************************
*/
void fatal_case_word_error (char *error_word)
{
        char in_str[132] = "";
        strcpy(in_str, error_word);
        puts
        ("*** Now entering FATAL_CASE_WORD_ERROR function ***");
        printf("\n\tWord other than %s", in_str, " found ");
        printf("in textual context of CASE statement!");
        printf("\n\t***FATAL ERROR***");
        printf("\n\n");
        pgm_success = 0;  /* abnormal terminate */
        puts
        ("*** Now leaving FATAL_CASE_WORD_ERROR function ***");
}       /* end fatal_case_word_error */


/*****************************************************************
* FUNCTION:  READ_WORD_OR_PASS (in_file, out_file)
*****************************************************************
*
* PURPOSE:  It gets the next word from the input file.  A word
```

```
*     starts with a letter or quote and can contain underscores.
*     For this program READ_WORD_OR_PASS also recognizes a
*     special word "=>".  Any nonword or whitespace found is
*     passed to the output file.
*
*  CALLED BY:    conv_case_to_if
*                case_to_if_conv
*
*  CALLS:        none
*
*  LOCAL VARIABLES:
*                the_char  : a character var used to read a
*                        char-at-a-time from the input file.
*                a_word    : a character array used to collect
*                        chars to form words from the input.
*                index     : the index for the a_word char array.
*
*  GLOBAL VARIABLES:
*                the_word  : used to update the current word
*                        for use by the calling functions.
*
*  AUTHOR:       Capt Dennis A. Rumbley
*                AFIT Masters Student
*                AFIT/ENG, GCS-91D
*
*  DATE:         20 Sep 91
*
*  VERSION:      1
*****************************************************************
*/
void read_word_or_pass (FILE *in_file, FILE *out_file)
{
        /*  variable declarations */
        char the_char = ' ';
        char a_word[132] = "";
        int index = 0;

        /*  processing statements */
        strcpy (the_word, "");
        the_char = fgetc (in_file);

   /* read characters until a valid character to start a word */
        while (!feof (in_file) &&
               (the_char < 'A' ||
               (the_char > 'Z' && the_char < 'a') ||
               the_char > 'z'))
        {
                /* return string literals */
                if (the_char == '"')
                {
                        a_word[index++] = '"';
                        the_char = fgetc (in_file);
```

```
                while (the_char != '"')
                {
                        a_word[index++] = the_char;
                        the_char = fgetc (in_file);
                }
                a_word[index] = '"';
                strcpy (the_word, a_word);
                the_char = fgetc (in_file);
                break;
        }
        /* look for process comment state */
        if (the_char == '-')
        {
                the_char = fgetc (in_file);
                if (the_char == '-')
                {
                        fputs ("--", out_file);
                        the_char = fgetc (in_file);
                         hile (!feof (in_file)
                               && the_char != '\n')
                        {
                             fputc (the_char, out_file);
                             the_char = fgetc (in_file);
                        }
                }
                else fputc ('-', out_file);
        }
        /* look for becomes symbol state */
        if (the_char == '=')
        {
                the_char = fgetc (in_file);
                if (the_char == '>')
                {
                        strcpy (a_word, "=>");
                        the_char = fgetc (in_file);
                        break;
                }
                else fputc ('=', out_file);
        }
       /* if not comment, becomes, or string lit pass */
        if ((the_char != '-') && (the_char != '=') &&
            (the_char != '"'))
        {
                fputc (the_char, out_file);
                the_char = fgetc (in_file);
        }
} /* end while char not legal to start a word */

/* now process a regular word state */
while (!feof (in_file) &&
      ((the_char >= 'A' && the_char <= 'Z') ||
       (the_char >= 'a' && the_char <= 'z') ||
```

```
                the_char == '_'))
        {
                a_word[index++] = the_char;
                the_char = fgetc (in_file);
        }

        /* restore file to read-a-word state */
        if (!feof (in_file))
                ungetc (the_char, in_file);

        /* copy local value to global variable */
        strcpy (the_word, a_word);

}  /* end read_word_or_pass */


/*********************************************************
*  FUNCTION:  READ_COND_VALUE (INPUT_FILE, OTHERS_FLAG)
*********************************************************
* PURPOSE:  It gets the next word from the input file.  A word
*     starts with a letter or quote and can contain underscores.
*     For this program READ_WORD_OR_PASS also recognizes a
*     special word "=>".  Any nonword or whitespace found is
*     passed to the output file.
*
*  LOCAL VARIABLES:
*               a_char  : a character var used to read a
*                       char-at-a-time from the input file.
*               a_word[132]  : a character array used to collect
*                       chars to form words from the input.
*               other_word[7] : a character array used to check
*                       if the input word was "others" or not.
*               index  : the index for the a_word char array.
*
*  GLOBAL VARIABLES:
*               the_word  : used to update the current word
*                       for use by the calling functions.
*
*  CALLED BY:   conv_case_to_if
*
*  CALLS:       none
*
*  AUTHOR:      Capt Dennis A. Rumbley
*               AFIT Masters Student
*               AFIT/ENG, GCS-91D
*
*  DATE:        20 Sep 91
*
*  VERSION:     1
*********************************************************************
*/
void read_cond_value (FILE *in_file, char *others_found)
```

```
{
        /* local variables */
        char a_char = '';
        char a_word[132] = "";
        char other_word[7] = "";
        int index = 0;

        /* process statements */
        a_char = fgetc (in_file);

        /* strip whitespace from front of condition value */
        while (!feof (in_file) && (a_char == ' ' ||
              a_char == '\t' || a_char == '\n'))
                a_char = fgetc (in_file);

        /* get condition value string */
        while (!feof (in_file) && a_char != '=')
                /* the first symbol of "=>" */
        {
                a_word[index++] = a_char;
                a_char = fgetc (in_file);
        }
        if (!feof (in_file)) ungetc (a_char, in_file);
        strcpy (the_word, a_word);

        /* check if other found */
        index = 0;
        while (index <= 5)
        {
                other_word[index] = a_word[index];
                ++index;
        }
        if (!strcmpi (other_word, "others"))
                strcpy (others_found, "true");
        else strcpy (others_found, "false");

}  /* end read_cond_value */



/*****************************************************
*  FUNCTION:  CONV_CASE_TO_IF
*****************************************************
*  PURPOSE:  Converts case statements to if statements.
*       When called, a word "case" has been read by
*       the calling function.  The case argument is read,
*       "is" and "when" are passed thorough, and the first
*       condition value is read.  The if_clause is now
*       generated.  The "=>" symbol is read but nothing output.
*       The "when_clause" is now processed.  All the words of
*       the when clause are passed to the output file with the
*       exception of the word "case" which if not preceded by
```

```
*       "end" causes a recursive call to conv_case_to_if.
*       When the next "when" is seen, the condition value is
*       read, and if not equal to "others" an elsif_clause is
*       generated and processed similarly to the if_clause.  If
*       the condition value is "others" an else_clause is
*       generated.
*
*  PARAMETERS:
*       in_file : a pointer to the input file.
*       out_file : a pointer to the output file)
*
*  LOCAL VARIABLES:
*       cond_arg[132] : a string to hold the case argument
*       cond_value[132] :  a string to hold the representation
*               of the current value of the case argument during
*               the conversion process.
*       *others_found : a string pointer used to check if the
*               word "others" has been found.
*       prev_word[132] :  a string ptr used to track the word
*               processed just before the current word during
*               the converion process.  Critical to recognize
*               the "end case" phrase signaling the conversion
*               is complete and only "if" needs to be written.
*
*  GLOBAL VARIABLES:
*       the_word : a string pointer used to contain the current
*               word being processed during the convert process.
*
*  FUNCTIONS CALLED:
*               fatal_case_word_error (char *);
*               read_word_or_pass (FILE *, FILE *);
*               read_cond_value (FILE *, char *);
*
*  CALLED BY: case_to_if_conv
*             conv_case_to_if
*
*  USE OF GOTO:  The goto statement is used only to emulate the
*     Ada-type EXIT statement.  The C break statement only exits
*     a loop, not necessarily a function.  The use of goto is
*     limited to goto EXIT_CONV_CASE_TO_IF which is located at
*     the end of the function.
*
*  AUTHOR:       Capt Dennis A. Rumbley
*                AFIT Masters Student
*                AFIT/ENG, GCS-91D
*
*  DATE:         20 Sep 91
*
*  VERSION:      1
*************************************************************
*/
```

```
void conv_case_to_if (FILE *in_file, FILE *out_file)
{
        /* called functions
        void fatal_case_word_error (char *);
        void read_word_or_pass (FILE *, FILE *);
        void read_cond_value (FILE *, char *);
        */

        /* variable declaration section */

        char cond_arg[132] = "";
        char cond_value[132] = "";
        char *others_found = "false";
        char prev_word[132] = "";

        /* function statements section */

        /* begin building if clause */
        read_word_or_pass (in_file, out_file);
        strcpy (cond_arg, the_word);

        /* read IS but pass nothing for it */
        read_word_or_pass (in_file, out_file);
        if (strcmpi (the_word, "is"))  /* the_word != "is" */
        {
                fatal_case_word_error (" IS ");
                goto EXIT_CONV_CASE_TO_IF;
        }

        /* read WHEN but pass nothing for it */
        read_word_or_pass (in_file, out_file);
        if (strcmpi (the_word, "when"))
         /* the_word != "when" */
        {
                fatal_case_word_error ("WHEN");
                goto EXIT_CONV_CASE_TO_IF;
        }

        /* read the conditional value for the if_clause */
        read_cond_value (in_file, others_found);
        strcpy (cond_value, the_word);

        /* read becomes symbol and pass THEN for it */
        read_word_or_pass (in_file, out_file);
        if (strcmp (the_word, "=>"))  /* the_word != "=>" */
        {
                fatal_case_word_error ("=>");
                goto EXIT_CONV_CASE_TO_IF;
        }

        /*  write if_clause to output */
        fputs ("if ", out_file);
```

```
fputs (cond_arg, out_file);
fputs (" = ", out_file);
fputs (cond_value, out_file);
fputs (" then ", out_file);

/*  read if_clause (when) body */
strcpy (prev_word, the_word);
read_word_or_pass (in_file, out_file);
while (strcmpi (the_word, "when"))
                                /* the_word != "when" */
{
        if (strcmpi (the_word, "case"))
           /* the_word != "case" */
        {
                fputs (the_word, out_file);
                strcpy (prev_word, the_word);
                read_word_or_pass (in_file, out_file);
        }
        else if (!strcmpi (prev_word, "end"))
                /* prev_word == "end" */
             {
                        fputs ("if", out_file);
                        goto EXIT_CONV_CASE_TO_IF;
             }
             else   /* new case statement encountered */
                    conv_case_to_if (in_file, out_file);
}
/***********************************************
 * second when now seen.  ready to process elsif
 * or else clauses.  can determine which when
 * read_cond_value returns.
 ***********************************************
 */
do
{
read_cond_value (in_file, others_found);
if (strcmpi (others_found, "true"))
        /* others_found != "true" */
{
        strcpy (cond_value, the_word);

   /* read becomes symbol and pass THEN in the clause */
        read_word_or_pass (in_file, out_file);
        if (strcmp (the_word, "=>"))
                                    /* the_word != "=>" */
        {
                fatal_case_word_error ("=>");
                goto EXIT_CONV_CASE_TO_IF;
        }

        /*  write elsif_clause to output */
        fputs ("elsif ", out_file);
```

```
            fputs (cond_arg, out_file);
            fputs (" = ", out_file);
            fputs (cond_value, out_file);
            fputs (" then ", out_file);

            /*  read elsif_clause (when) body */
            strcpy (prev_word, the_word);
            read_word_or_pass (in_file, out_file);
            while (strcmpi (the_word, "when"))
                                    /* the_word != "when" */
            {
                    if (strcmpi (the_word, "case"))
                    /* the_word != "case" */
                    {
                      fputs (the_word, out_file);
                      strcpy (prev_word, the_word);
                      read_word_or_pass (in_file, out_file);
                    }
                    else if (!strcmpi (prev_word, "end"))
                            /* prev_word == "end" */
                    {
                            fputs ("if", out_file);
                            strcpy (the_word, "");
                            goto EXIT_CONV_CASE_TO_IF;
                    }
                    else   /* new case stmt encountered */
                        conv_case_to_if (in_file, out_file);
            }
    }
    else  /* "others" was seen -- process else statement */
    {
           /* read becomes symbol and pass nothing for it */
            read_word_or_pass (in_file, out_file);
            if (strcmp (the_word, "=>"))
                                    /* the_word != "=>" */
            {
                    fatal_case_word_error ("=>");
                    goto EXIT_CONV_CASE_TO_IF;
            }

            /*  write else_clause to output */
            fputs ("else ", out_file);

            /*  read else_clause (when) body */
            strcpy (prev_word, the_word);
            read_word_or_pass (in_file, out_file);
            while (strcmpi (the_word, "when"))
                                    /* the_word != "when" */
            {
                    if (strcmpi (the_word, "case"))
                    /* the_word != "case" */
                    {
```

```
                        fputs (the_word, out_file);
                        strcpy (prev_word, the_word);
                      read_word_or_pass (in_file, out_file);
                    }
                    else if (!strcmpi (prev_word, "end"))
                            /* prev_word == "end" */
                    {
                            fputs ("if", out_file);
                            strcpy (the_word, "");
                            goto EXIT_CONV_CASE_TO_IF;
                    }
                    else   /* new case stmnt encountered */
                        conv_case_to_if (in_file, out_file);
                }
        }
        } while (!strcmp (others_found, "false"));
                         /* others_found == false */
        EXIT_CONV_CASE_TO_IF: strcpy (others_found, "false");
                        /* needed due to recursion */
}  /* end conv_case_to_if */


/*************************************************************
*  FUNCTION:  CASE_TO_IF_CONV (input_file, output_file)
*************************************************************
*  PURPOSE:   Reads an input file and passes words read to
*       an output file until a word "case" is read.  At
*       that point, "case" is not passed to the output.
*       Instead, conv_case_to_if is called to convert the
*       case statement to an if statement.  Then control
*       returns to this function.
*
*  PARAMETERS:
*       infile : a string value (the name of the input file)
*       outfile : a string value (the name of the output file)
*
*  LOCAL VARIABLES:
*       in_file : a string pointer used to identify the
*               input file to function case_to_if_conv.
*       out_file : a string pointer used to identify the
*               output file to function case_to_if_conf.
*
*  GLOBAL VARIABLES:
*       pgm_success : an integer which if 0 tells that
*               the program was not successful.
*       the_word : a string pointer used to contain the current
*               word being processed during the convert process.
*
*  FUNCTIONS CALLED:  conv_case_to_if
*                     read_word_or_pass
*
*  CALLED BY: main
```

```
*
*  AUTHOR:       Capt Dennis A. Rumbley
*                AFIT Masters Student
*                AFIT/ENG, GCS-91D
*
*  DATE:         20 Sep 91
*
*  VERSION:      1
************************************************************
*/
void case_to_if_conv (char *infile, char *outfile)
{
        /*********************************/
        /* variable declarations section */

        /* called functions declarations */
/*      void conv_case_to_if (FILE *, FILE *);
        void read_word_or_pass (FILE *, FILE *);
*/
        /* internal file identifiers */
        FILE *in_file;
        FILE *out_file;

        /*********************************/
        /* function statements section */

        /* open files
        */
        in_file = fopen (infile, "rt");
        out_file = fopen (outfile, "wt");

        /* move words and nonwords to output
           until a case statement is encountered,
           the end of the input file is reached,
           or an abnormal condition causes unsuccessful
           program termination.
        */

        read_word_or_pass (in_file, out_file);
        while (!feof (in_file) && pgm_success == 1)
        {
                /* function strcmp required to see if the_word
                   is equal to a given string.  If so, strcmp
                   returns a 0.  Must not (!) strcmp to get a
                   "true" (1) reply when the stings are equal
                */
                if /* the_word == "case" */
                   (!strcmpi (the_word, "case"))
                        conv_case_to_if (in_file, out_file);
                else
                {
                        fputs (the_word, out_file);
```

```
                        read_word_or_pass (in_file, out_file);
                }
        }  /* while !eof(in_file_handle) or pgm_success = 1 */

        /*  close files after processing
        */
        fclose (in_file);
        fclose (out_file);

}  /* end CASE_TO_IF_CONV */


/************************************************************
*  FUNCTION:  MAIN
************************************************************
*  PURPOSE:   Driver routine for program to convert case
*             statements in an input file to if statements
*             in an output file.
*
*  PARAMETERS:
*       argc : an integer that tells how many arguments
*              are on the command line (incl pgm name)
*       argv[] : a string array of arguments on the cmd
*              line.  The number of these arguments is
*              argc - 1.
*
*  LOCAL VARIABLES:
*       in_file : a string pointer used to identify the
*              input file to function case_to_if_conv.
*       out_file : a string pointer used to identify the
*              output file to function case_to_if_conf.
*
*  GLOBAL VARIABLES:
*       pgm_success : an integer which if 0 tells that
*              the program was not successful.
*
*  FUNCTIONS CALLED:  case_to_if_conv
*
*  CALLED BY: none
*
*  AUTHOR:       Capt Dennis A. Rumbley
*                AFIT Masters Student
*                AFIT/ENG, GCS-91D
*
*  DATE:         20 Sep 91
*
*  VERSION:      1
************************************************************
*/
int main (int argc, char *argv[])
{
```

```
                /* variable declarations section */

                char *in_file = "";
                char *out_file = "";
/*              void case_to_if_conv (char *, char *);
*/
                /* program statements */
                if (argc != 3)
                {
                        puts ("****** ERROR:  Wrong Number of Files on
                                                Command Line ******\n");
                        puts (" *** Please enter the name of your input
                                                    file and a name\n");
                        puts (" *** for the output file.  EX.  casetoif
                                                  in_fil out_fil\n\n\n");
                        pgm_success = 0;
                }
                if (pgm_success == 1)
                {
                in_file = argv[1];
                out_file = argv[2];
                puts ("Now converting CASE statements in file ");
                puts (in_file);
                puts ("\nto IF statements in a file called ");
                puts (out_file);
                puts ("\n\n");
                case_to_if_conv (in_file, out_file);
                puts ("*** Thank you for using CASETOIF.EXE ***\n\n\n");
                }
                if (pgm_success == 1)
                        return 0;
                else
                {
                        puts ("*** ABNORMAL PROGRAM RUN ***");
                        return 1;
                }
        }   /* end main */
```

SECTION 3: TEST FILES AND RESULTS

TEST 1

```
--************************************************************
--            This is a file of comments and
--            should be passed straight through
--************************************************************
--  case BOX of
--       when crackers => eatem;
--       when bolts => useem;
--  end case;
------------------------------------------------------------
```

TEST 1 RESULTS

```
--************************************************************
--            This is a file of comments and
--            should be passed straight through
--************************************************************
--  case BOX of
--       when crackers => eatem;
--       when bolts => useem;
--  end case;
------------------------------------------------------------?
```

TEST 2

```
and
the
-- 7789
december
case
edit
tom
the folks at home
party
end case
bye
```

TEST 2 RESULTS

```
and
the
-- 7789
```

december

TEST 3

```
-- This file is to test if CASETOIF.EXE can process string
-- literals
-- that contain the keyword CASE without trying to process
-- a case statement
signal_1 <= "0010" after 10 ns;
write ("In case you didn't know, signal_1 is now '0010'.");
signal_2 <= signal_1 after 50 ns;
case test is
        when 1 => load_memory;
        when 2 => disable_chip;
        when 3 => reset_pgm_counter;
        when others => null;
end case;
write ("That is the end of the CASE file.");
```

TEST 3 RESULTS

```
-- This file is to test if CASETOIF.EXE can process string
-- literals
-- that contain the keyword CASE without trying to process
-- a case statement
signal_1 <= "0010" after 10 ns;
write ("In case you didn't know, signal_1 is now '0010'.");
signal_2 <= signal_1 after 50 ns;

        if test = 1  then  load_memory;
        elsif test = 2  then  disable_chip;
        elsif test = 3  then  reset_pgm_counter;
        else  null;
end if;
write ("That is the end of the CASE file.");
```

TEST 4

```
-- This file contains nested case statements to enable
-- testing
-- of the CONVERT CASE TO IF program (casetoif.exe) for
-- handling
-- nested cases.

case country is

     when USA => say "Yea!";
     when JAPAN => say "Aso!";
     when USSR => case republic is
          when RUSSIA => say "go Boris!";
          when ESTONIA | LATVIA => say "Free at Last!";
          when GEORGIA => say "Civil War!";
          when otHeRs => say "go Capitalists!";
          end case;
          say "Communism never had a chance!!!";
     when OtherS => say "Who Cares!!!";

end case;

say "Finished processing nested cases."

file end.
```

TEST 4 RESULTS

```
-- This file contains nested case statements to enable
-- testing of the CONVERT CASE TO IF program (casetoif.exe)
-- for handling nested cases.

if country = USA  then  say "Yea!";
elsif country = JAPAN  then  say "Aso!";
elsif country = USSR  then
        if republic = RUSSIA  then  say "go Boris!";
        elsif republic = ESTONIA | LATVIA  then  say "Free
               at Last!";
        elsif republic = GEORGIA  then  say "Civil War!";
        else  say "go Capitalists!";
     end if;
     say "Communism never had a chance!!!";
else  say "Who Cares!!!";

end if;

say "Finished processing nested cases."

file end.
```

(Note: May need to expand symbol "|" to "or cond_var = ". Will know more when run through IDAS. None of the FPASP CASE statements contained this operator, so VHDL analysis did not test for this case.)

TEST 5 was a run of the CASE_to_IF program against files of the FPASP design. The results of this test are included in Appendix C.

Bibliography

Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Reading MA: Addison-Wesley Publishing Company, 1977.

Albrecht, Paul F. and others. "Source-to-Source Translation: Ada to Pascal and Pascal to Ada," Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language, in SIGPLAN Notices, 15: 183-193 (November 1980).

Allman, E. "YACC Attack," UNIX Review, 6: 76-81 (September 1988).

Bothe, K. and others. "A Portable High-Speed PASCAL to C Translator," SIGPLAN Notices, 24: 60-65 (September 1989).

Boyle, James M. "Lisp to Fortran -- Program Transformation Applied," Proceedings of the NATO Advanced Research Workshop on Program Transformation and Programming Environments, in Program Transformation and Programming Environments. edited by Peter Pepper. Berlin: Springer-Verlag, 1984.

Calingaert, Peter. Program Translation Fundamentals: Methods and Issues. Rockville MD: Computer Science Press, Incorporated, 1988.

Cohen, Daniel I. A. Introduction to Computer Theory. New York: John Wiley & Sons, Incorporated, 1986.

Davis, Alan M. Software Requirements: Analysis & Specification. Englewood Cliffs NJ: Prentice-Hall, Incorporated, 1990.

Fisher, Charles N. and Richard J. LeBlanc, Jr. Crafting a Compiler. Menlo Park CA: The Benjamin/Cummings Publishing Company Incorporated, 1988.

Fox, David L. "Learning About LEX," Computer Language, 51-56 (February 1987).

Ganapathi, M. "Semantic Predicates in Parser Generators," Computer Languages, 14: 25-33 (June 1989).

Gane, Chris and Trish Sarson. Structured Systems Analysis: Tools and Techniques. Englewood Cliffs NJ: Prentice-Hall Incorporated, 1979.

Hayes, John P. Computer Architecture and Organization (Second Edition). New York: McGraw-Hill, Incorporated, 1988.

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

Integrated Design Automation System IDAS Product Description Summary. JRS Research Laboratories Incorporated, Orange CA, June 1988.

Krieg-Brückner, Bernd. "Language Comparison and Source-to-Source Translation," Proceedings of the NATO Advanced Research Workshop on Program Transformation and Programming Environments, in Program Transformation and Programming Environments. edited by Peter Pepper. Berlin: Springer-Verlag, 1984.

Lesk, M.E. Lex -- A Lexical Analyzer Generator, Computer Science Technical Report No. 39. Murray Hill NJ: Bell Laboratories, October 1975.

Lipsett, Roger and others. VHDL: Hardware Description and Design. Kluwer Academic Publishers, 1989.

Miller, Capt Richard L. Specification and Equivalence Verification of Sequential Circuits via VHDL. MS thesis, AFIT/GE/ENG/90D-41. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, November 1990.

Owen, G. Scott. "Computer Assisted Pascal to Ada Program Translation," in Empirical Foundations of Information and Software Science IV: Empirical Methods of Evaluation of Man-Machine Interfaces. edited by Pranas Zunde and Jagdish C. Agrawal. New York: Plenum Press, 1987.

Pintelas, P.E. and others. "A Translator from Small Euclid to Pascal," SIGPLAN Notices, 24: 93-101 (May 1989).

Sideri, M. and others. "Semantically Driven Parsing of Context-free Languages." The Computer Journal, 32: 91-93 (February 1989).

Software User's Manual for the Enhanced Automated VHDL/Microcode Compiler Synthesis and Design System (AMSDS). Document Number SUM.0074-05. JRS Research Laboratories Incorporated, Orange CA, 21 March 1989.

VHDL Style Guide for Ada To Microcode Compiler Retargeting and VHDL Simulation. Document Number SUM.0086-01. Contract N00039-87-C-0256. JRS Research Laboratories Incorporated, Orange CA, 27 February 1989.

Waters, Richard C. "Program Translation Via Abstraction and Reimplementation," AI Memo No. 949, December 1986. Contracts DARPA N00014-85-K-0124 and NSF MCS-7912179. Boston: Massachusetts Institute of Technology, December 1986 (AD-A185 845).

Webster's II New Riverside University Dictionary. Boston: Houghton Mifflin Company, 1984.

Yourdon, Edward. Structured Design: Fundamentals of a Disciplined of Computer Program and Systems Design. Englewood Cliffs NJ: Prentice-Hall Incorporated, 1979.

-----. Electronic Data Processing -- Structured Techniques for System Analysis. Englewood Cliffs NJ: Prentice-Hall Incorporated, 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1991	Master's Thesis

4. TITLE AND SUBTITLE
Design of Style-V – A Translator to Convert Standard VHDL into a Stylized Form for Automated Microcode Generation

5. FUNDING NUMBERS

6. AUTHOR(S)
Dennis A. Rumbley, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/91D-19

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

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

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

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Ths thesis provides an analysis and preliminary design of Style-V, a source-to-source computer language translator. Style-V converts IEEE standard VHDL into a special style of VHDL defined for a commercial tool, the Integrated Design Automation System (IDAS). Thirteen mappings between stand- ard VHDL and the IDAS subset were identified. The mappings were analyzed using Domain Analysis and Modern Structured Analysis techniques. Four processes covering several of the mappings were completely analyzed. One mapping to convert CASE statements to IF statements was implemented. Since the IDAS restricts designs to bit logic, a method for represent- ing multilevel logic with bit logic was devised. Unaccept- able multiple process architectures were converted to multi- ple single process architectures which are acceptable to IDAS. The IDAS microcode generator does not recognize user- defined procedures, but in one case, mapping user-defined procedures to IDAS defined procedures was not possible. In general, this problem amounts to showing two programs are functionally equivalent. Exhaustive testing was ruled out since proving two 32-bit adders are equivalent would take over 11 billion years at 100 procedure runs per second. The program equivalence problem was not solved by this thesis. Useful results were obtained, though IDAS failed to work.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Language Translation, Programming Languages, Machine Translation, High Level Languages, Computer Programs, Source-to-Source Translation	221
	16. PRICE CODE

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