UC Irvine
ICS Technical Reports

Title
VSS : a VHDL synthesis system

Permalink
https://escholarship.org/uc/item/3r87f9c5

Authors
Lis, Joseph S.
Gajski, Daniel D.

Publication Date
1988-05-06

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Notice: This Material
may be protected
by Copyright Law
(Title 17 U.S.C.)

Archives
Z
699
C3
no. 88-13
c.2

VSS: A VHDL

Synthesis System

by

Joseph S. Lis
Daniel D. Gajski

Technical Report 88-13

Information and Computer Science
University of California at Irvine
Irvine, CA 92717
(714) 856 7063

Abstract: This report describes a register transfer synthesis system that allows a designer to interact with the design process. The designer can modify the compiled design by changing the input description, selecting optimization and mapping strategies, or graphically changing the generated design schematic. The VHDL language is used for input and output descriptions. An intermediate representation which incorporates signal typing and component attributes simplifies compilation and facilitates design optimization. The compilation process consists of two phases. First, a design composed of generic components is synthesized from the input description. Second, this design is translated into components from a particular library by a mapper and optimized by a logic optimizer. Redesign to new technologies can be accomplished by changing only the component library.

TABLE OF CONTENTS

1. Introduction 1
1.1. Motivation 1
2. Previous Work 6
3. System Description 19
4. VHDL Input Description 24
4.1. Signal Declarations and Types 24
4.2. Data Flow Description Style 26
4.2.1. VHDL Concurrent Statements 26
4.2.1.1. Conditional Signal Assignment 26
4.2.1.2. Selected Signal Assignment 28
5. Graph Representation 31
5.1. Node and Net Data Structure 33
5.2. Node Types 34
6. Compilation Algorithm 40
7. Graph Critic 42
8. Design Generation 46
9. Reverse Compiler 50
9.1. Generic Component Netlist 54
9.2. Interface to Other Synthesis Tools 59
10. An Example 60
11. Conclusions 69
12. References 71

TABLE OF FIGURES

Figure 1: VHDL Controlled Counter Chip Model 10
Figure 2: VHDL Synthesis System 20
Figure 3: Conditional Signal Assignment 30
Figure 4: Flowgraph for Selected Signal Assignment 30
Figure 5: Graph Representation: Input Description and Symbol Table 32
Figure 6: Graph Representation: Flow Graph 33
Figure 7: Graph Critic Cleanup Rules 44
Figure 8: Graph Critic μ Architecture Substitution Rules 45
Figure 9: VHDL Generic Component Netlist Format 51
Figure 10: Schematic for Example Design 51
Figure 11: VHDL Structural Description of an Example Circuit 54
Figure 12: VHDL Description of the Bus Interface Circuit 63
Figure 13: Interconnected Flowgraph for Bus Interface Example 64
Figure 14: Graph Critic Rule Applications 65
Figure 15: Graph Critic Rules 65
Figure 16: Flowgraph after Optimization 67
Figure 17: Generic Component Schematic for Bus Interface Example 67
Figure 18: VHDL Generic Component Netlist 69

1. Introduction

In order to successfully exploit new technologies, the problem of rapid prototyping of new systems and redesigning old parts must be solved. To solve these problems, a new generation of design tools that capture human design knowledge must be developed. However, this knowledge about translating functional specifications to structural representations and structural representations into physical design is not sufficiently well understood to allow the development of CAD tools based on simple algorithms. To make the problem even more complicated, the functional specifications are often incomplete and given with conflicting design goals.

1.1. Motivation

Design Synthesis on the register transfer level covers several areas: hardware description languages, design representation (data base), logic synthesis, and behavioral synthesis. The *hardware description language* (*HDL*) provides a method of specification for the designer so that the design automation system can be supplied with sufficient information to synthesize the desired circuit. A *design representation* or *data base* is the internal description used by the design automation system which organizes information extracted from the input

specification necessary for synthesis. This representation is created and optimized by the system so that a netlist or other output specification can be produced.

The purpose of a true hardware description language is to specify the structure of a design in terms of interconnected components. This representation contains sufficient information for tasks such as technology mapping (the process of transforming a technology independent design into a technology specific one) which operate on an existing design. However, this description is at too low a level to support synthesis since design decisions have already been made in selecting and interconnecting components. Simulation languages classified as HDLs are used to model hardware. These languages are tailored to optimize the performance and correctness of the simulation through the use of constructs which have no direct correlation to hardware; consequently, synthesis from these descriptions is difficult. The ideal HDL is a language which captures the intended functionality of a design in terms of generic components (a universal set of technology independent hardware primitives) and provides sufficient information to guide the synthesis process.

The design process often proceeds through several stages of the abstraction hierarchy (processor, register transfer (RTL), gate, circuit, layout). In

particular, two stages of the digital design process will be addressed in the context of this work: logic synthesis and behavioral synthesis. *Logic sythesis* operates at the gate level which uses simple logic gates (AND, OR, INVERTER, etc.), flipflops and selected MSI components as design primitives. Systems which attempt to automate this process accept a textual (netlist, HDL, boolean equations, tables) or graphical (schematic, menu interaction) input and perform logic level optimizations to meet area, timing or other design constraints. *Behavioral synthesis* operates at the higher levels. It involves the description of the functionality and input/output interfaces of the hardware to be designed in algorithmic form (programming language, HDL). A system which addresses this level of synthesis first translates this specification into a design representation which can be operated on by design automation tools. This design data base is then used to create a more detailed description of the design at lower levels of abstraction.

The types of designs produced by behavioral synthesis can be classified in several different ways. The designs may consist of purely combinational elements or may contain registers or other storage elements, thereby making them sequential. Since sequential designs have more than one state, a further classification according to the modeling level is introduced. A one state per process model may be assumed where each process or block describes one state

of the design. Alternatively, each process may model many states. In this case, a design usually consists of a data path and control unit. The data path can perform different operations on different clocks as determined by control signals supplied by the control unit. The behavioral synthesis process allocates data path units, schedules operations which can be performed on available units in the given state, and determines the control which must be supplied to the data path for each state.

The objective of this work is to develop a system which performs behavioral synthesis; specifically, the translation of a VHDL description to a netlist of generic components. The VHDL description may be written in a data flow style where concurrent statements describe the flow of information between memory and gating elements or in a behavioral style which uses sequential statements to abstractly describe the function of the hardware. There are two motivations for this work. The first of these is to provide a methodology for high level design. Because the designer is working at a higher level of abstraction (at the processor or RT levels instead of gate, circuit or layout levels), requirements for design expertise at the lower levels of design are removed, thereby increasing productivity. A second motivation is to allow for redesign of existing designs using new technologies. The use of a generic component netlist makes the design independent of implementation style (gate arrays, standard cells, custom). By

mapping generic components to a particular gate array, standard cell, or custom component library, the designer may consider alternative styles and fabrication processes.

2. Previous Work

There are two main applications of hardware description languages: to document a design and to model a design. VHDL has been used primarily as a modeling and simulation language in hardware design efforts up to this point. Recent efforts in design automation, including this work, seek to work with a behavioral description that captures both the information necessary to simulate the function of a design as well as to include attributes necessary to synthesize the structure of the design.

Armstrong [Arms87][Arms88] illustrates how VHDL can be used to model hardware at the various levels of abstraction. His work focuses on methods for representing various behavioral aspects of chip level modeling. At this level, a component is a complete VLSI chip such as a microprocessor, memory chip, or UART. The chip is modeled as a single entity (not constructed hierarchically from more basic primitives) which performs a sequence of micro-operations coded in an HDL. The model defines the input/output response of the device by specifying the algorithm the chip is to implement.

Because logic signals flow in parallel, any hardware model must include a provision for concurrency of execution. The VHDL language handles this notion of simultaneity with the use of the process statement. Each process represents a

block of logic, with all processes executing in parallel. Armstrong defines a graph representation termed the *process model graph* which uses nodes to represent a partitioning of the function of the model into subfunctions. Intercommunication between process nodes is denoted by arcs. VHDL process statements implement each node subfunction, while signals appearing in the process signal list model timing delay and input/output relationships between subfunctions. A node may be functionally decomposed into more than one VHDL process. For example, a node representing a register with synchronous load and asynchronous clear attributes can be modeled by two processes, one representing the effects of the load operation, the other reflecting the effects of the clear operation.

While this style of VHDL description may correctly simulate the behavior of the hardware, it presents several problems when viewed from the synthesis perspective. First, the separation of the description of a single component into several process statements complicates the task of collecting and identifying attributes to be associated with that component. Furthermore, this description style relies on the VHDL simulator's notion of a container to assign the correct value to a signal at any given time based on one or more drivers. A container represents signal nets as well as registers, making the task of identifying these entities difficult for the compiler. Often, complicated language constructs are

used to combine these drivers which result in a suboptimal design when mapped to logic components.

For example, Figure 1 shows a VHDL description which uses separate statements to model the asynchronous clear and synchronous up/down count of a controlled counter [Arms87]. The drivers (OUT1, OUT2) generated to represent the effects of each event on the register's output value are combined using a conditional signal assignment statement MUX1. Note that MUX1 is a virtual component which should have no hardware realization. The sole purpose of the statement is to collect the multiple drivers for simulation.

Two approaches may be taken to translate this behavioral description into hardware: direct mapping of VHDL constructs to appropriate microarchitecture components, or recognition of certain VHDL construct patterns as a representation of a particular hardware concept. If a straightforward mapping of VHDL constructs is performed, inefficient hardware will often result. In the above example, an unnecessary multiplexor will be introduced when mapping the MUX1 statement to hardware, with each driver as a data input and complicated selection logic. A sophisticated logic critic would then be needed to transform this design into an optimal one (i.e., a register with up/down count and clear control inputs). The latter method of translation requires

```
Architecture PROCESS_IMPL of CONTROLLED_CTR is

  signal CLK,EN: BIT;
  signal CONSIG: BIT_VECTOR(0 to 3);
  signal OUT_TMP,OUT1,OUT2: BIT_VECTOR(0 to 3);

CLEAR_CTR: block (CONSIG(0) = '1' and not CONSIG(0)'stable)
  begin
    OUT1 <= guarded "0000" after CLRDEL;
  end block CLEAR_CTR;

CNT_UP_OR_DOWN: process (CLK,EN)

  variable CNT: BIT_VECTOR(0 to 3);
  variable CLKE: BOOLEAN;

  begin
    if EN'stable then
      if EN = '0' then
          CLKE := TRUE;
      else
          CLKE := FALSE;
      end if;
    end if;
    if (CLK = '1' and not CLK'stable and CLKE) then
      if (CONSIG(2) = '1') then
          CNT := INC(CNT);
      else if (CONSIG(3) = '1') then
          CNT := DEC(CNT);
        end if;
    end if;
    OUT2 <= CNT after CNTDEL;
  end process CNT_UP_OR_DOWN;

MUX1: OUT_TMP <= OUT1 when not OUT1'quiet else
                OUT2;

end block PROCESS_IMPL;
```

Figure 1: VHDL Controlled Counter Chip Model

identification of the type of signals used to select the input driver. Since VHDL allows the designer to express the same functionality in many different ways, the task of developing a rule set which recognizes all valid VHDL representations of a desired set of hardware concepts would be extremely difficult, if not impossible. The compilation process becomes simplified if the descriptions are not allowed to contain virtual components.

In order to reduce the complexity of the synthesizer, several systems have been designed which use a subset of language constructs that can be mapped to an appropriate set of hardware components. Systems such as SOCRATES [GrBa86], LSS [JoTr86] and Logic Consultant [Kim87] are targeted to synthesis at the gate or logic level. Each accepts input in the form of boolean equations, PLA format, structural (netlist) or algorithmic behavioral description. The IBM VHDL environment [Saun87] combines a subset of VHDL language constructs with language extensions to synthesize an RTL level design using components from a target technology library.

A common characteristic of the above approaches to synthesis is the direct correspondence of a set of language constructs to a fixed set of hardware components. Technology details are hard coded into the translation process through the use of a fixed component library. Such restrictions limit the

flexibility of these synthesis systems when using different component libraries. The input descriptions, while easier to map to hardware, force the designer to use a representation style which is more structural than algorithmic or behavioral.

When a behavioral description is used, synthesis consists of the following phases: language translation, global optimization, data path allocation, data path module binding, control path allocation and control path module binding [TsWe88]. Language translation involves parsing a high level behavioral description and translating it into an intermediate representation (usually a control/data flow graph). Global optimizations such as standard language compiler data flow analysis or identification of signals and registers are then performed on this intermediate form in order to increase the efficiency and amount of parallelism in a design. Data path allocation involves component selection, state synthesis and connectivity binding. During this phase, the number and types (attributes) of components are selected, graph operation nodes are assigned to machine states, and assignment of operations to hardware components is performed. A symbolic microcode table is usually generated as the allocation is made, indicating the control signals to be supplied to components during each machine state. Once the data path is synthesized, control paths can be generated after selecting a control style (hardwired, PLA, ROM, pipelined).

Facet [TsSi83] is a system for automated synthesis of data paths developed at CMU. It uses an input graph representation known as a Value Trace (VT) [McFa78] that is derived from an ISPS [Barb81] input description. The Value Trace is a collection of VT-bodies, each of which consists of a linear sequence of data flow operations. In order to maintain control information in the same representation, control constructs are translated into their data flow equivalents and included in the VT. A clique-partitioning method is used to minimize the number of storage elements, data operators and interconnection units required to realize the design. Resources are shared in the final design wherever possible unless it is specified in the ISPS description that elements should not be combined (the reserved variable declaration). The Emerald design generator [TsSi86] was developed to perform manipulations (orderings) on the operation sequence before passing the VT-body on to Facet. Emerald also provides the designer with the capability to select design criteria (in the case of multiple alternatives for component allocation) and generates simple design evaluation statistics such as gate count.

Bridge [TsWe88] is a high level synthesis system which is an extension of the Facet work. The system firsts allocates the generic structure of data and control paths and then proceeds with technology specific data/control module binding. The FDL2 behavioral language is used as input and is parsed into a

control/data flow graph. Global optimizations identify intermediate variables as signals or registers in addition to performing scheduling optimizations. Structural synthesis performs allocation and produces a design consisting of RTL components (registers, memories, ALUs, multiplexors) and a symbolic control table. The allocation program partitions the data flow representation into a number of code slices. Methods described for Facet are used to determine the storage, arithmetic and logic, and interconnection resources to be used in the design. If high speed performance is an important constraint on the design, pipelining schemes are considered. Module binding is then performed on the structural description to map generic components of the data path to a standard cell implementation using the FDS system for MOS chip design. Once the data path is physically generated, the control path is synthesized.

The BECOME [Wei88] system assumes a finite state machine circuit model with an external view of only primary inputs and outputs. It accepts a C-like behavioral modeling language input. An intermediate form similar to the Value Trace is generated by an input parser with the purpose of resolving semantic differences among modeling languages. A Data Flow Analyzer identifies temporal or *scan variables* which effect the input/output behavior of the model. These variables are allocated to a minimal set of registers or latches, and the representation is then modified to create a combinational model (one in which

no variables are assigned more than once). If a variable is assigned in more than one branch of a complex statement, *name splitting* is performed to rename each multiply assigned variable. These multiple assignments are recombined in the bus allocation phase through the insertion of multiplexors. A *structure description* is generated from a one-to-one mapping of operators in the flattened model to functional elements. Global optimizations are performed by a *module logic generator*. The output of this stage is either a generic logic description or a standard cell implementation produced by technology binding. The generic logic is represented in BLIF format which can be fed to logic minimization systems such as ESPRESSO. The optimized logic is then mapped into the chosen implementation technology.

The V-SYNTH system [Bhas86][KBhN] provides a VHDL input interface to the existing MIMOLA Synthesis System (MSS) [Zimm85]. It seeks to improve upon the drawbacks of the MIMOLA system: to remove the burden of decomposing the behavioral description into operations to be performed in individual control states, and to perform global data flow analysis which minimizes the required number of operators and storage elements. A *Process Graph Analyzer* accepts a VHDL behavioral input and generates a process graph by decomposing each statement and expression into a simple form (one operator and at most two operands). Compiler-like techniques (constant folding, local

code optimization, code motion, common subexpression elimination) are used to optimize this description. The *Control State Generator* partitions the process graph into control states, introducing parallelism where possible. A reverse transformation from the process graph to legal VHDL syntax is performed by a translator. This description consists of a set of process statements, with each process describing operations to occur in a single control state.

The MIMOLA design system is intended to be an interactive design aid. To that end, the Design Representation is a database where the output of the Process Graph Analyzer is stored for designer interaction. The designer is allowed to modify the Design Representation by adding hardware bindings or constraints before presenting the description to MSS for synthesis. When MSS is invoked, an implementation is generated by binding hardware components to operators and variables in the representation. A statistical analyzer provides information such as component utilization to aid the designer in determining constraints to meet design goals.

The above mentioned work in behavioral synthesis has several limitations. First, the input languages used have restricted the designer to specify the design at the level (instruction set or algorithm, register transfer, logic) to which the system had been targeted. Simple design models are used (e.g., microprocessor

or DSP) which only apply to a limited domain of design problems. These systems fail to address important aspects of real world circuit design such as signal timing attributes or asynchronous behavior. Examples which demonstrate the operation of these systems do not represent typical design problems.

In evaluating the strengths and weaknesses of the approaches discussed above, a set of criteria for a synthesis system can be derived:

(1) A common, well defined design representation which will uniquely capture the functionality and intention of several equivalent behavioral descriptions and will lead to an unique, optimal hardware design.

(2) The ability to specify and synthesize attributes such as timing protocols, asynchronous control, pipelining, multiclocking, multifunction units, and multiple processes with different clocking strategies within the design representation.

(3) Separation of procedural tasks (such as translation, allocation, binding, global optimization) which can be accomplished using known algorithms from heuristic tasks (local optimizations) which are best accomplished using a specialized set of rules. This implementation strategy takes advantage of the ability to perform optimization on the design at various levels where the

information is most accessible. For example, optimizations for circuit area are accomplished more effectively at the local level where tradeoffs can be made for individual components based on available primitives in the library, while critical path timing requires global analysis and optimization.

(4) The system should remain technology independent for as long as possible. This suggests that the synthesized design should be composed of *generic components* which can be considered primitive building blocks of all target technologies. The system can then be adapted to any implementation technology by defining the mapping of these generic components to technology specific components and specifying local optimizations to tune the design to the selected technology specific components.

(5) It would be desirable to generate a design representation which can be used as input to simulation or other synthesis tools. Using the same language for the input and output descriptions would be particularly beneficial (e.g. VHDL behavioral input, VHDL structural output). In this manner, the design can be first synthesized at a higher level of abstraction using a behavioral input description, then passed to tools which perform logic level refinement and design which operate on a more structural description of the design.

The remainder of this report presents a system which addresses these issues. Advantages recognized in the previous work cited above are incorporated and expanded into this work. The VHDL language has been selected to specify the input behavioral description and the output netlist format because of its flexibility for description of hardware at various levels of abstraction and its ability to associate attributes with signals or other design entities.

3. System Description

This section outlines a specification for a complete system for designing from a VHDL specification. Such a system translates VHDL descriptions into manufacturable chips. The input is a VHDL description at the system, register-transfer, and logic levels with some or all components described behaviorally. The output is a description ready to be sent for fabrication. The system can be targeted to gate arrays, standard-cells or custom designs by writing a technology translator.

The system consists of several major components as illustrated in Figure 2.

(1) The **Data Base** stores the design, its versions and alternatives, and manages hierarchy. The relations between original and compiled designs as well as reasons for particular design styles and modifications are stored in the database. This information can be used by an Explanation Facility to explain design changes to the designer.

The Data Base stores four types of information. For each hardware block or component it stores its VHDL behavioral description, the **Control/Data Flow Graph** obtained from the behavioral description, a structural description consisting of generic components, and the technology specific

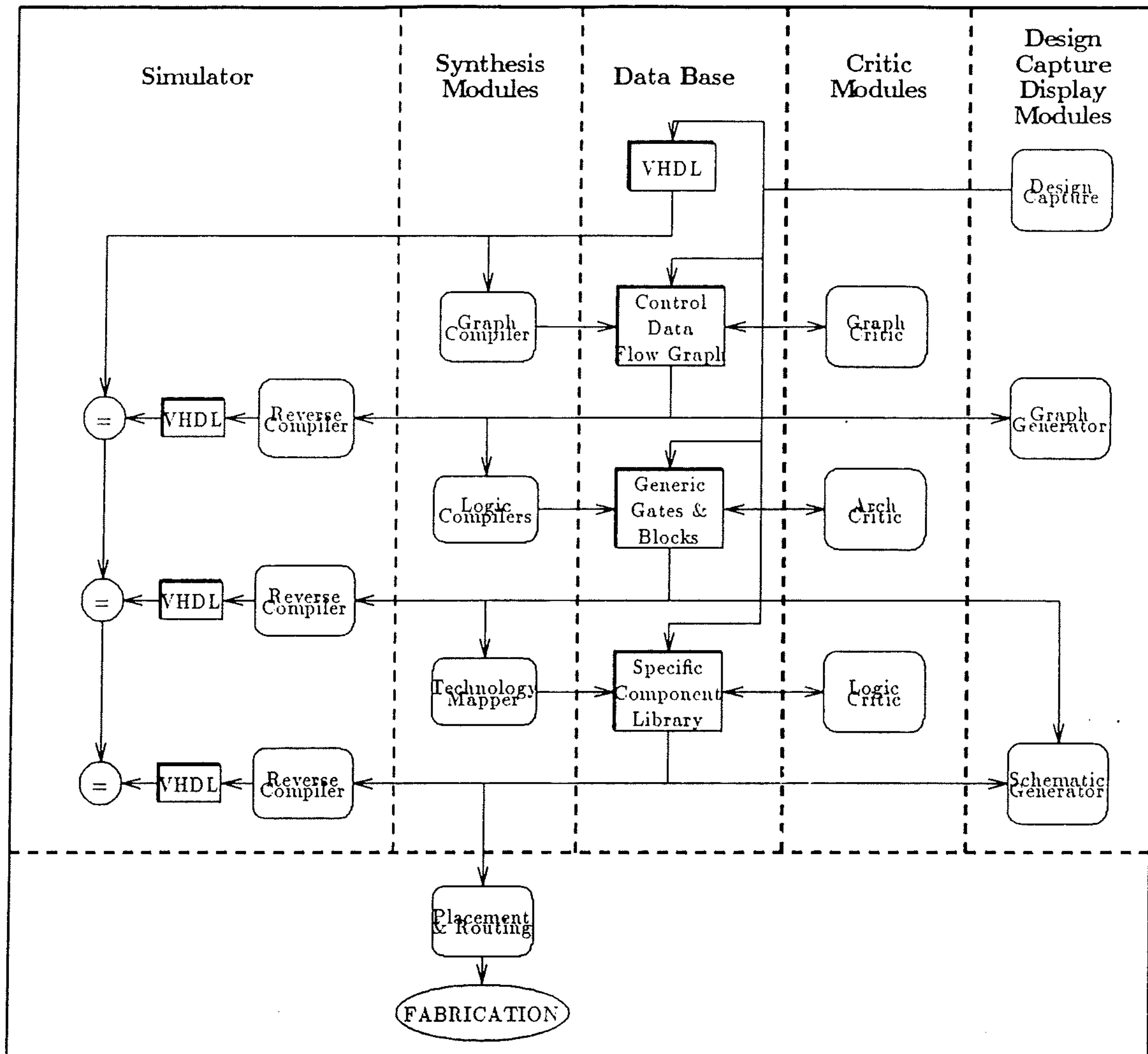

Figure 2: VHDL Synthesis System

structure description in which generic components are replaced with components from prespecified libraries and then optimized. The data base can be considered to be the blackboard of a blackboard expert system

architecture in which the knowledge sources are the graph, architecture or logic critics.

(2) A set of compilers is needed to make the design description more explicit or add new detail that did not exist before. The **Graph Compiler** translates textual VHDL description into a graph which is used to optimize the design description and remove redundancy associated with textual description. **Logic Compilers** then translate each graph node into a set of generic components (gates, registers, counters, ALUs, etc.) while the **Technology Mapper** converts generic components to components in a chosen technology (gate macros for gate arrays, standard cells or custom blocks generated by silicon compilers).

(3) A set of critics or optimizers is required to simplify or optimize the design. The **Graph Critic** removes redundancies from the language and improves the efficiency of the design using rearrangement and merging. For example, it converts a nested IF statement into a CASE statement. The **Architecture Critic** optimizes the design on the functional or register-transfer level. Merging an incrementer and a register into a counter is an example of this type of optimization. A **Logic Critic** optimizes the design on the logic level by simplifying components with constant inputs and performing logic

minimization. For example, this critic will replace an EXOR gate with one input connected to logic "1" with an inverter. The Logic Critic will also speed up critical paths (paths that do not satisfy timing constraints) by using faster units, by design rearrangement, or by introducing more parallelism into the design.

(4) The **Design Capture and Display** component is a man machine interface. The designer specifies the design as a set of interconnected modules. Each module is described by VHDL block or process statements. After compilation or any design modification, the compiled design is displayed back to the designer with a possible explanation of why and how a design was changed from the previous version. In addition to providing a monitoring function, the **Graph Generator** and **Schematic Generator** are necessary tool development aids which are used during code development and debugging.

(5) A **VHDL Simulator** is used to check the intent of the initial descriptions. It is also used during compilation to check the consistency of the compiled design by comparing output vectors. The description of the compiled or manually modified design is obtained by a set of **Reverse Compilers** that link together models of the compiled structural description or graph

representation of the design. Reverse compilers are needed to fit this system into the VHDL environment since the proposed synthesis system takes a behavioral VHDL description and compiles it into a structural VHDL description.

4. VHDL Input Description

The VHDL behavioral description itself consists of a *design entity* composed of two major sections: the *entity block* and the *architecture body*. The entity block contains the specification of external input/output port connections to the hardware to be designed. The architecture body consists of a description of the hardware to be designed in one of three styles: **structural**, which describes a hierarchy of interconnected components; **data flow**, which uses concurrent statements to describe the flow of information between memory and gating elements; and **behavioral**, which uses sequential statements to abstractly describe the function rather than structure of the hardware.

4.1. Signal Declarations and Types

VHDL supports the following standard data types:

signal
BIT
BIT_VECTOR
BOOLEAN
INTEGER

For synthesis purposes, the following special types are defined:

subtype CLOCK is BIT
subtype SET is BIT
subtype RESET is BIT
subtype REGISTER is BIT_VECTOR
subtype BUS is BIT_VECTOR
subtype WIRED is BIT_VECTOR

VHDL signal declarations can occur in two sections of the behavioral description: within the entity block, where external port connections are declared, and within block or process statements of the architectural body, where internal connections and storage elements are declared. These declarations are of the form:

{signal} <signal-name> : <mode> <type>

The <mode> attribute identifies the direction in which data flows at a port (IN, OUT, INOUT). We will define a signal to be of mode **internal** if it is not declared as a port in the entity portion of the VHDL description but is declared as a local signal within an architectural body. The <type> is one of the data types defined above. As these declarations are processed by the Graph Compiler, entries are made into the symbol table to record the signal attributes.

4.2. Data Flow Description Style

The input description was initially constrained to consist of **concurrent assignment statements** which appear within the VHDL **block** construct. The following VHDL concurrent statements are supported:

1) conditional signal assignment
2) selected signal assignment
3) guarded signal assignment

In addition, the following synthesis issues will be addressed:

1) bus and register signal qualifiers
2) timing specifications
3) use of wired-or and bus nodes
4) assumed predefined signal types
5) recognition of edge transitions on clock signals

4.2.1. VHDL Concurrent Statements

4.2.1.1. Conditional Signal Assignment

The *conditional signal assignment* will occur in one of the following forms:

a) signal <= <waveform> ;

This is the simplest form of assignment statement where

<waveform> ::= <expression> { **after** <delay> }

The VHDL simulator interprets this statement as a directive to compute the value of <expression> and schedule the activation of this driver for the

signal value at time <current-simulation-time> + <delay> (if no delay is specified, the driver is activated immediately).

From the CDFG perspective, a data flow graph is constructed for the RHS expression, and the result is input to a WRITE node for the signal. Associated with each graph arc (connection) is a **signal type** (bus, register, port, wire), **mode** (in/out/inout (for ports only), internal), **number of bits**, and **representation**. The optional delay specification indicates the time which elapses between the READ of all signals/variables which appear on the RHS of the assignment statement and the appearance (WRITE) of the updated expression value at the register/port/wire represented by the signal.

b) signal <= **guarded** <waveform> ;

The *guarded assignment* involves the conditional assignment of the evaluated <waveform> to the signal based on the value of the **guard expression** which appears at the beginning of the enclosing VHDL block statement. When the guard expression evaluates to true, the VHDL simulator activates the signal driver and places its value on the simulator event queue so that the signal is updated at the specified simulation time.

For the purposes of CDFG generation and synthesis, a guarded signal assignment is used for signals declared with the **bus** or **register** qualifier. A data flow graph is generated for the RHS expression and is connected to the true input of a CHOOSE-VALUE node. The CHOOSE-VALUE has a guard input which is a data flow graph representing the block guard expression. The output of the CHOOSE-VALUE node is used as the input to a WRITE node for the signal.

If the signal is declared as a bus, the CHOOSE-VALUE represents a tri-state driver for the bus signal. If the signal is a register, the CHOOSE-VALUE represents a clock or control signal input to the register. The type of the guard input net will indicate the function of the signal.

c) signal <= { **guarded** }
waveform1 **when** condition1 **else**
waveform2 **when** condition2 **else**
.
.
.

```
waveformN when conditionN else
waveformN;
```

This statement corresponds to a nested if arrangement of assignments to the same signal based on different boolean conditions. The VHDL simulator will evaluate waveform/condition pairs in the order in which they appear and will schedule the assignment of the first waveform value to the signal when its associated condition evaluates to true.

This statement can be useful in representing an assignment to a signal based on prioritized conditions. For example, the statement in Figure 3 might be used to represent a register for which the CLEAR is of highest priority, followed by PRESET and CLOCKed assignment. Figure 3 shows the flowgraph generated for the statement.

A chain of CHOOSE-VALUES is constructed to form the data flow graph for the nested if construct. The bottom most CHOOSE-VALUE is guarded by the first condition encountered, the CHOOSE-VALUE above the bottom one is guarded by the next condition, etc. The output of the bottom most CHOOSE-VALUE is connected to the WRITE node input.

4.2.1.2. Selected Signal Assignment

The format of the *selected signal assignment* is as follows:

```
with <expression> select
  signal <= { guarded }
                waveform1 when choice1 ,
                waveform2 when choice2 ,
                              .
                              .
                              .
                waveformN when choiceN ;
```

This is equivalent to the case statement available as a sequential statement within the process construct. The choices are exclusive conditions (either integer or boolean values) such that only the waveform matching the value of the <expression> is evaluated and scheduled for assignment by the VHDL simulator. Figure 4 shows the flowgraph generated for the general form of this

```
reg_A <=
        '0' after 20 ns when CLEAR = '0' else
        '1' after 20 ns when PRESET = '1' else
        DATA after 35 ns;
```

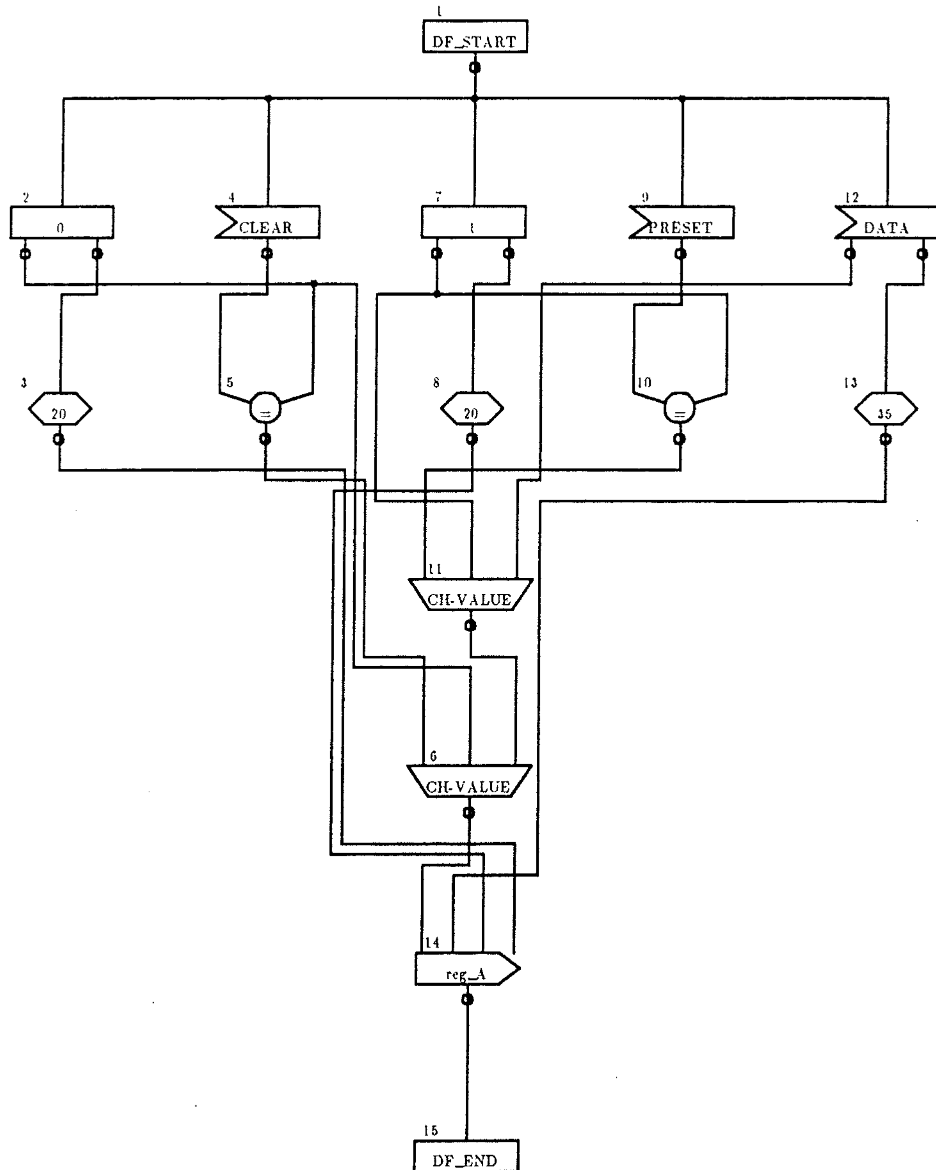

fig2.dgm   Sun Feb 14 18:59:39 1988

Figure 3: Conditional Signal Assignment

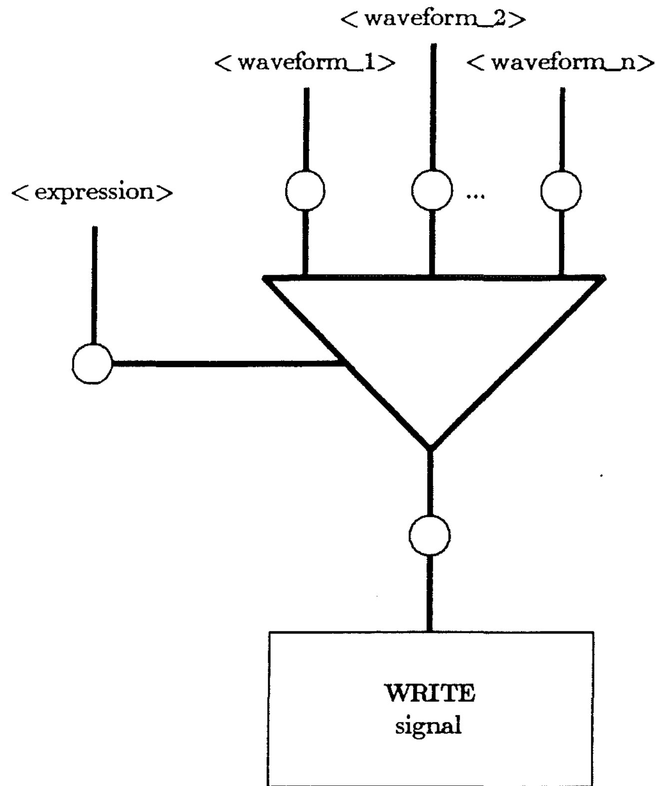

Figure 4: Flowgraph for Selected Signal Assignment

statement.

The data flow graph construct associated with this statement is the multiple input CHOOSE-VALUE guarded by the <expression>. Each waveform will have a corresponding data flow graph generated for its expression value, and the guard test for each input will be stored in the input net.

5. Graph Representation

A control/data flow graph representation [OrGa86] is used and extended for the VHDL language. As the VHDL input description is parsed, a symbol table entry is created for each signal containing the following information: **mode** (in/out/inout (for ports only), internal), VHDL **declared type** (BIT, BIT_VECTOR, CLOCK, etc.), **dimensions** (for vectors and arrays), **wiring** (bus, register, port, wire), and **representation** (magnitude, sign/magnitude, 1's complement, 2's complement). READ nodes are used to access a signal value appearing on the RHS of an assignment statement. Operator nodes represent arithmetic, logic or signal selection operations. The net representing the output of the RHS expression is connected to the input of a WRITE node for the LHS signal. Associated with each graph arc (connection) is a **signal type** (DATA, CLOCK, SET, RESET, TIMING), **number of bits, active edge** (positive, negative) and **sensitivity** (edge, level). Optional delay specifications are represented in the flow graph by DELAY nodes that indicate the time which elapses between the READ of all signals/variables which appear on the RHS of the assignment statement and the appearance (WRITE) of the updated expression value at the register/port/wire represented by the signal.

Figure 5 and Figure 6 illustrate a typical flowgraph representation. Signal attributes are extracted from port and variable declarations. Attributes for the CLK net are collected from the guard condition for the block statement in the VHDL description. Timing specifications are derived from the **after** clause of signal assignment statements.

```
entity REGISTER is
  port
   (DATA_in: in BIT_VECTOR(0 to 3);
    CLK: in CLOCK;
    OUTPUT: out BIT_VECTOR(0 to 3))
end REGISTER;

architecture EXAMPLE of REGISTER is
  signal A: BIT_VECTOR(0 to 3) register;
begin (CLK = '1' and not CLK'STABLE)
  A <= guarded DATA_in after 10 ns;
end EXAMPLE;
```

SYMBOL TABLE					
name	mode	declared type	dimensions min	dimensions max	wiring
CLK	in	CLOCK	-	-	PORT
DATA_in	in	BIT_VECTOR	0	3	PORT
OUTPUT	out	BIT_VECTOR	0	3	PORT
A	internal	BIT_VECTOR	0	3	REGISTER

Figure 5: Graph Representation: Input Description and Symbol Table

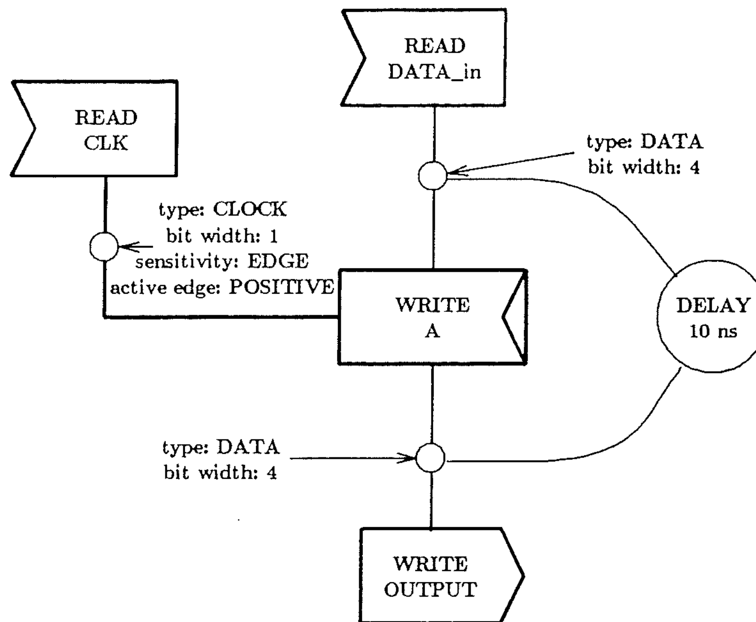

Figure 6: Graph Representation: Flow Graph

5.1. Node and Net Data Structure

As each statement in the VHDL input description is parsed, flowgraph node and net structures are created and interconnected. A node data structure is created for each signal access (READ/WRITE), data operation, or control operation. Information maintained in the node structure includes:

(a) a unique node number
(b) the node type (dataflow, control, demarcation, μarch)
(c) a list of input net connections
(d) output net connection
(e) the operator type
(f) data dependency information (for the behavioral description style)

As the Graph Critic operates on the flowgraph structure, nodes in the graph will be converted to microarchitectural components.

The net data structure maintains node connectivity information as well as signal attributes. It contains the following information:

(a) a unique net number
(b) the source node for the net
(c) a list of destination nodes
(d) a list of delays terminating at the net
(e) a list of delays originating at the net
(f) bit width
(g) signal type (DATA, CLOCK, RESET, SET, TIMING)
(h) signal edge (positive, negative)
(i) signal sensitivity (level, edge)

5.2. Node Types

This section describes all nodes which may appear in the flowgraph. For each node, the attributes and range of values for these attributes is given. Table 1 classifies each of the nodes into groups based on the type of node. Following this classification table, the individual node details are given.

Node Types

Operation Nodes

- Arithmetic
  Adding Operators: ADD, SUB, CONCAT
  Unary Sign: +, -
  Multiplying Operators: MULT, DIV, MOD, REM

- Logical Operators
  n to 1 bit Reductions: OR, AND, EXOR, NOR, NAND
  n to n bit Logic Units: OR, AND, EXOR, NOR, NAND

- Negation: NOT

- Relational Operators: EQ, NE, LT, LE, GT, GE

Storage/Value Reference Nodes

- Signal Reference Nodes: READ, WRITE
- Port Reference Nodes: READ, WRITE
- Register Reference Nodes: READ, WRITE
- Vector/Array Reference: READ_ARRAY, WRITE_ARRAY, SUBSCRIPT
- Constant Read Nodes

Selection Nodes

- Choose-Value Nodes

Delay Nodes

Demarcation Nodes

- START
- END

Table 1: Node Group Classifications

Operation Nodes

Operation nodes represent the execution of the specified operator using one or more supplied inputs. The result of this operation is represented by the output net of the node. The class of operation nodes is further subdivided into Arithmetic, Logical, Negation and Relational subclasses.

The Arithmetic Adding operator nodes ADD and SUB have two input connections and a single output connection which has a bit width that is the maximum of the input bit widths. The CONCAT operator requires an n-bit input and an m-bit input. It concatenates these vectors without modification to form an (n + m)-bit output. The Unary Sign + operator performs no modification. From the synthesis viewpoint, it is a VHDL language redundancy. The - operator inverts the magnitude of the input signal. Multiplying operators represent the operations associated with the designated keyword as described in the VHDL Language Manual [VHDL87].

Logic operators perform the specified logic function on the inputs in one of two fashions: n to 1 bit reductions and n bit logic units. The reduction operator treats each bit of the input as a unique input and performs the logic function on all inputs. The logic units treat each of two n bit inputs as a single input, producing an n bit output. Negation produces an output which inverts each bit

of the input.

Relational operators represent a comparison of two n bit inputs. A single bit output is produced indicating the result of the specified comparison as being TRUE (logic '1') or FALSE (logic '0').

Storage/Value Reference Nodes

This class of nodes represent a retrieval of current signal, port or constant values (READs) or an update of a signal or port value (WRITEs). Signal reference nodes are used as intermediate markers during the synthesis process as flowgraphs for individual statements are created and interconnected. They represent internal signal references. Since no hardware is allocated for these nodes, they are removed as the final flowgraph is created. Port reference nodes denote references to external ports.

Register reference nodes represent assignment and retrieval of values which are to be stored in register components. These nodes have a CLOCK input which is used to load (latch) the data input. Other asynchronous control lines (SET, RESET, ENABLE) may be present. Attachment of these control lines to register reference nodes is determined during the synthesis process by taking into consideration signal type declarations and the scope of guard conditions

surrounding VHDL assignment statements.

Vector/Array reference nodes define the selection of one element (bit or word) or a subrange of elements from a multiple element bus or memory component.

Selection Nodes

The Choose-Value node is used to represent a conditional signal assignment. A SELECT input chooses one of the data inputs which will be passed to the output. Associated with each data input is a constant condition guard against which the SELECT input is tested. If condition guards, are consecutive, the node models a multiplexor component; otherwise, the node models a decoder/multiplexor component.

Delay Nodes

Delay nodes incorporate global timing parameters into the design representation. This information can be used in global timing analysis or be assigned as performance attributes (input/output response times, propagation delays) for a component.

Demarcation Nodes

Demarcation nodes indicate the beginning and end of a data flow graph block. The start of the block is represented by the START node; it is connected to all READ PORT and READ CONSTANT nodes accessed in the flowgraph. Similarly, the END node marks the end of the data flow graph block, and all WRITE PORT nodes are connected to it.

6. Compilation Algorithm

The basic algorithm used by the Graph Compiler to generate a control/data flow graph from the VHDL behavioral description consists of parsing each statement and interconnecting graph sections. For each assignment statement, a data flow graph is constructed for the RHS expression using standard compiler techniques. Based on the description style, statement flow graphs are interconnected and merged, generating a single graph.

Each concurrent assignment statement which appears within a VHDL block construct will have a separate CDFG generated for it. The order of occurrence of concurrent assignment statements is unimportant. This differs from the compilation process for sequential statements which imposes an ordering of execution of these statements and introduces READ/WRITE dependencies based on that ordering.

If a data flow description is being processed, flow graphs are generated for each statement and then interconnected once all statements have been processed. This corresponds to the concurrent data flow style where all operations are assumed to be executed in parallel. The sections of CDFG representing each signal assignment will be appropriately interconnected based on the signal type. It is the signal type that will define whether a VHDL signal

(container) represents a memory element, port or wire. The signal type will also determine the interconnect protocol (wired-or, bus) which results when multiple sources for the same VHDL signal are encountered.

Statements are processed and interconnected as they are encountered within descriptions of the behavioral style (identified by the use of the VHDL *process* construct). This method corresponds to sequential execution of operations where data dependencies are important.

The single interconnected flowgraph begins with a START demarcation node which is connected to all external input port references which occur within the body of the behavioral description. Dataflow nodes created and interconnected during the parsing of individual statements comprise the body of the flowgraph. An END demarcation node is connected to all external output port references occurring in the behavioral description.

7. Graph Critic

Once the initial flow graph has been generated and entered into the Design Data Base, a rule-based Graph Critic performs optimizations on the flow graph structure. Because VHDL allows the designer to express the same functionality in many different ways, a Graph Critic module is needed to transform these various representations into an unique construct which represents the hardware concept being described.

The Graph Critic applies two types of rules when optimizing the flowgraph. The first rule set consists of *cleanup* rules. These rules eliminate redundant constructs in the flowgraph. Figure 7 gives several examples of cleanup rules. For example, a WRITE node followed by a READ node for a variable of type signal can be replaced by a simple wire connection since no storage element is required.

The second rule set contains *microarchitectural substitution* rules. These rules implement heuristics which make local substitutions for patterns of interconnected graph nodes based on signal types or other attributes associated with those patterns. In this manner, flowgraph constructs representing the behavior of the hardware are systematically replaced by nodes which more closely represent microarchitectural components with those attributes. Figure 8

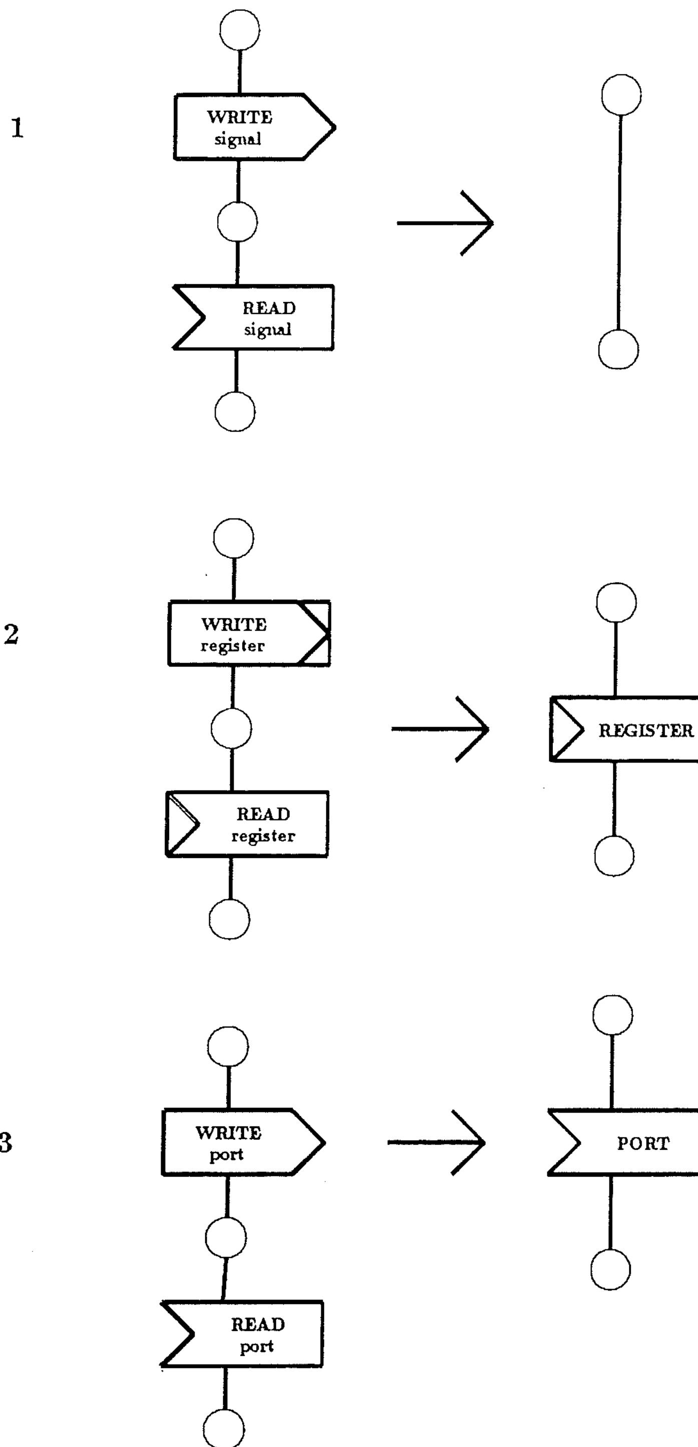

Figure 7: Graph Critic Cleanup Rules

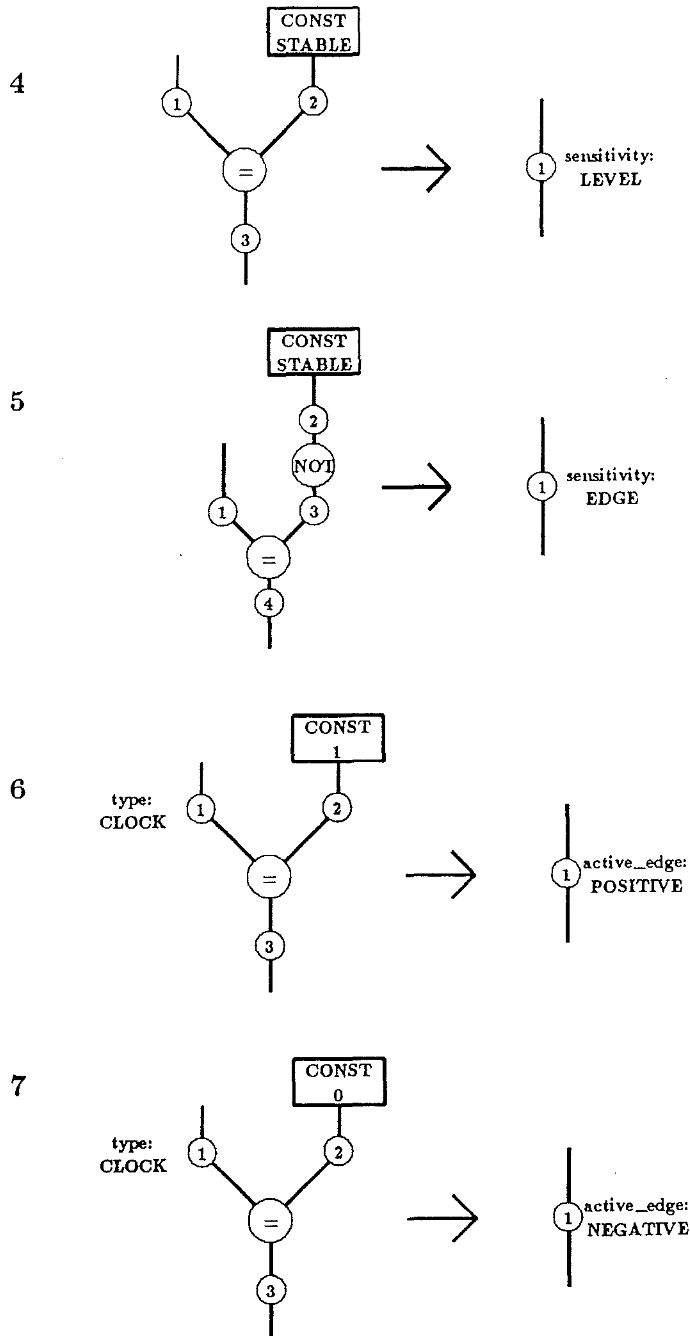

Figure 8: Graph Critic μ Architecture Substitution Rules

shows several examples of rules which remove flowgraph sections generated by the translation of VHDL constructs and captures the necessary design information in the form of net attributes.

These optimizations simplify the task of assigning generic logic components to corresponding operation nodes in the flow graph representation. This approach is unique to our methodology in that the optimizations are performed incrementally at every stage of the design process where the functional information is recognizable, rather than postponing these mapping decisions by passing them on to future (logic) optimization processes where this data is not easily retrieved.

8. Design Generation

Design Generation is the process of mapping each node in the flowgraph to a single or a combination of available microarchitectural component(s) in the generic component library. A component library specification is supplied to the synthesis system in tabular form (reference generic component table specification). Parameters and attributes are extracted from specifications in the flow graph structure so that constraints are met. Certain flowgraph nodes such as the CHOOSE-VALUE node are mapped into several components. After each flowgraph node is replaced by the appropriate library element(s), the partial design will consist of a netlist of generic logic components.

The component table describes the port connections and functionality of a set of generic components. Table 2 illustrates a representative component set used in the MILO system [VZGa88].

MICROARCHITECTURAL COMPONENTS

(every component has loading and delay attributes)

```
GATES
      (function (= AND, OR, INV, NAND, NOR, EXOR, EQ),
      # inputs.
      )

LOGIC UNIT
      (# bits
       function (= 2-variable Boolean functions)
      )

INTERFACE (type (= tristate, buffer, clock),
      level (= TTL, ECL),
      function (= inverting, non-inverting),
      # inputs.
      )

SELECTOR
      (# bits,
       type (= binary),
       # inputs.
      )

DECODER
      (# bits,
       type (= binary),
       control (= enable).
      )
```

Table 2: Module Generators and Parameters

```
COMPARATOR
      (# bits,
       function (= >, <, =, ...).
      )

ALU
      (# bits,
       function (= +, -, INC, DEC, logic functions).
      )

REGISTER
      (# bits,
       type (= latch, D-FF),
       function (= load, shift),
       control (= set, reset, enable),
       i/o (= serial, parallel).
      )

COUNTER
      (# bits,
       function (= load, up, down),
       control (= set, reset, enable),
       mode (= ripple, carry-look-ahead),
       clock (= single),
       type (= custom, binary).
      )

RAM/ROM
      (# bits,
       size
       control (= read, write, request, ready),
       select (= # bits, polarity).
      )
```

Table 2: Module Generators and Parameters (continued)

BARREL SHIFTER
(# bits,
function (= left shift, left rotate, right shift, right rotate),
fill in (= left, right, 0, 1).
)

MULTIPLIER
(# bits,
representation (= magnitude, 1's complement, 2's complement).
)

REGISTER FILES
(# bits,
ports,
type (= FIFO, stack, register),
port type (= in, out, in/out),
port control (= enable, load).
)

Table 2: Module Generators and Parameters (continued)

9. Reverse Compiler

The types of information contained in the design representation after the Graph Compilation, Graph Critic, and Design Generation phases have completed can be classified as follows: **component instances, connectivity, instance parameters,** and **timing**. A *Reverse Compiler* is needed to present the design representation in a textual or graphic form which will allow the designer to examine the results of synthesis and verify design correctness. The *generic component netlist* representation which is the textual output of the synthesis system must express this design information. A standard format for this netlist is desirable so that interfacing to other design tools can be more easily accomplished.

The VHDL structural style of description seems suited to this purpose. The general form of this netlist is shown in Figure 9. Figure 10 shows a schematic of an example circuit whose netlist is given in Figure 11. The following sections describe how the various types of design information are represented in this netlist format.

```
-- interface portion

entity <entity-name> is
  <port-declarations>
  <external-timing-assertions>
end <entity-name>;

-- architectural body (structural description style)

architecture Structure_View of <entity-name> is
  <component-declarations>
  <component-attributes>
  <internal-signal-declarations>
  <internal-timing-assertions>
begin
  <component-instantion-statements>
end Structure_View;
```

Figure 9: VHDL Generic Component Netlist Format

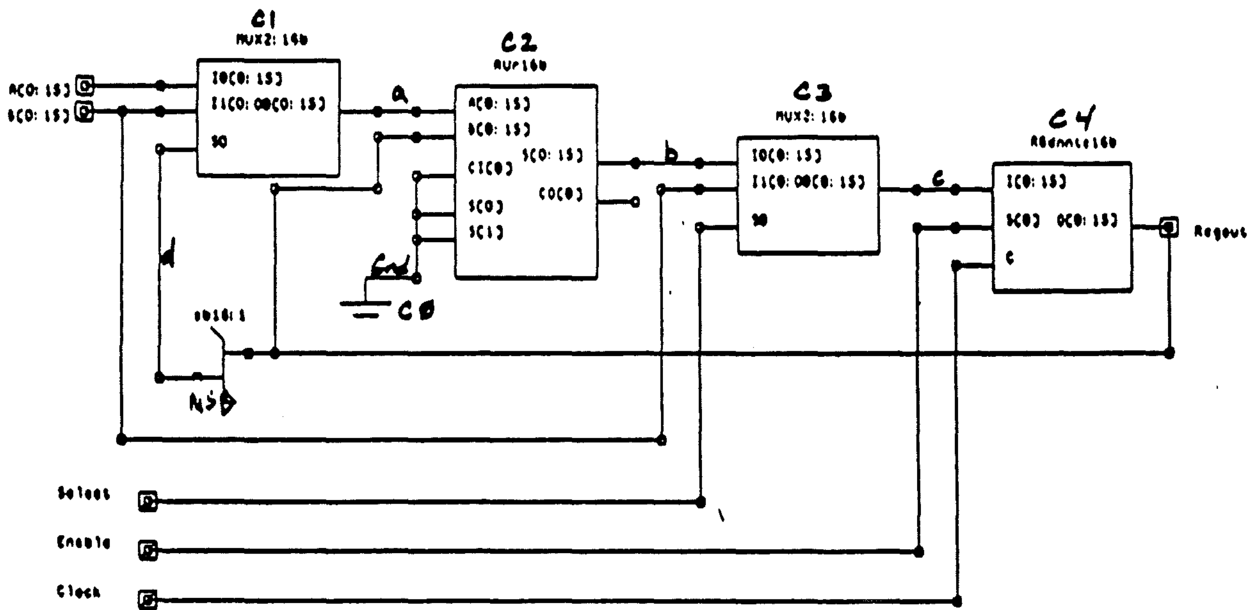

Figure 10: Schematic for Example Design

```
-- interface portion

entity Example1 is
  port (A,B: in: BIT_VECTOR(0 to 15);
        Select,Enable: in BIT;
      Clock: in CLOCK;
      Regout: out BIT_VECTOR(0 to 15));

-- external timing assertions

--T ↑↓ A to ↑↓ Regout: 20 ns average
--T ↑ Clock to ↑↓ Regout: 5,10 ns

end Example1;

-- architectural body (structural description style)

architecture Structure_View of Example1 is

-- component declarations

  Component MUX2_16b
    port (I0,I1: in: BIT_VECTOR(0 to 15);
          S0: in BIT;
        O0: out BIT_VECTOR(0 to 15));

  Component AUr16b
    port (A,B: in: BIT_VECTOR(0 to 15);
        CI: in BIT;
        S: in BIT_VECTOR(0 to 1);
        SUM: out BIT_VECTOR(0 to 15);
        CO: out BIT);

  Component RGdnnte16b
    port (I: in: BIT_VECTOR(0 to 15);
        S: in BIT;
        C: in CLOCK;
        O: out BIT_VECTOR(0 to 15));

  Component sb16_1
    port (I: in: BIT_VECTOR(0 to 15);
```

```
        O: out BIT);

    Component GND
      port ( O: out BIT);

 -- component attributes

    type FUNC_TYPE is (ADD,SUB,INC,DEC);
    type CARRY is (RIPPLE,LOOKAHEAD);
    attribute FUNCTION: FUNC_TYPE;
    attribute ADDER_TYPE: CARRY;
    attribute ENBL: BOOLEAN;

    attribute FUNCTION of AUr16b: component is ADD;
    attribute ADDER_TYPE of AUr16b: component is RIPPLE;
    attribute ENBL of C4: label is TRUE;

 -- internal signal declarations

    signal a,b,c: BIT_VECTOR(0 to 15);
    signal d,Gnd: BIT;

 -- internal timing assertions

 --T ⇅ a to ⇅ b: 20,25,35 ns
 --T ↓ A to ⇅ b: 40 ns max

 -- component instantiations

 begin
   C0: GND port map (Gnd);
   C1: MUX2_16b port map (A,B,d,a);
   C2: AUr16b port map (a => A,Regout => B,Gnd => CI,Gnd => S(0),
          Gnd => S(1),b => SUM);
   C3: MUX2_16b port map (b,B,Select,c);
   C4: RGdnnte16b port map (c,Enable,Clock,Regout);
   C5: sb16_1 port map (Regout,d);
     d <= Regout(15);
 end Structure_View;
```

Figure 11: VHDL Structural Description of an Example Circuit

9.1. Generic Component Netlist

Entity Declaration

The entity declaration portion of the VHDL structural description specifies the design name and defines the design's interface to the outside world. Port declarations are used to define input and output connections. VHDL **assertion** statements are used to specify timing constraints from input to output ports of the design. The section on timing assertions below will describe the format of these statements.

Component Declarations

For each unique component in the netlist, a component declaration must exist. This declaration defines a template containing input and output pin specifications via port declarations. The type and bit width of the signals (nets) to be attached to the component ports are specified in these declaration statements.

In order to generate a generic netlist using a set of generic components, a table of available components and their component declarations must exist. This table should identify the function of each input and output pin and the pin naming conventions for each component. It should also specify the operand port

mappings for multiple operation units. If this component declaration table is available to interface programs which accept the netlist as input, it would not be necessary to include component declarations in the netlist.

Component Attributes

In order to specify parameters particular to a component such as ALU functions, control input codes, etc., the VHDL *attribute* declaration and specification features can be used. Enumeration types can be used to specify the allowable values of an attribute. Attributes may be associated with the template component declaration, or with specific labeled instances of a component. For example, the statements

```
type FUNC_TYPE is (ADD,SUB,INC,DEC);
attribute FUNCTION: FUNC_TYPE;
attribute FUNCTION of AUr16b: component is ADD;
```

will associate the FUNCTION attribute ADD with every instance of an AUr16b component, while the attribute specification

```
attribute ENBL of C4: label is TRUE;
```

will associate the ENBL attribute with RGdnnte16b instance C4 only.

Internal Signal Declarations

Internal connection of components is accomplished by defining each internal net of the generic component netlist using signal declaration statements. These signal (net) names are used in the port map specification of component instantiations described below in order to identify uniquely the net connections between component ports.

Timing Assertions

It is often necessary and useful when specifying timing constraints of a circuit to have the capability of specifying relationships between signals. For example, a common requirement is that the data input to a clocked register be stable a duration of time prior to the clock transition that strobes the data into the register (sometimes known as set up time) [Arms87]. The following signal transitions should be representable:

1. ↑S transition from 0 to 1 of signal S (rising)
2. ↓S transition from 1 to 0 of signal S (falling)
3. ↑↓S any transition of signal S (change)

The timing relationship is expressed as follows:

<transition1> to <transition2> : <duration>

where <transition1> and <transition2> are of the form specified above. The

<duration> specification is used to specify the minimum, maximum and/or average time interval(s) between two events. A single time period specification must be followed by a qualifier (**max**, **min**, or **average**). For example:

--T ↓ A to ↑↓ b: 40 ns max

A list of two time intervals specifies a minimum/maximum timing specification, such as:

--T ↑ Clock to ↑↓ Regout: 5,10 ns

A triplet of time intervals denotes minimum, average, and maximum, as in:

--T ↑↓ a to ↑↓ b: 20,25,35 ns

One method of expressing timing information which conforms to the VHDL language definition would be to use comments. For example, the statements

--T ↑↓ A to ↑↓ Regout: 20 ns average
--T ↑ Clock to ↑↓ Regout: 5,10 ns

would be parsed as comments by the VHDL Analyzer, but the netlist parser could recognize the *--T* timing assertion delimiter and record the specified timing information.

Component Instantiations

A component is instantiated through the use of a VHDL *component instantiation statement* within the block of the architectural body. This statement has the form:

<label>: *<component-name>* *<port-map>*;

The <label> is a unique id for the component. A component declaration statement for <component-name> must exist, defining the ports (mode, bit width) to be found in the <port-map> list. The <port-map> is a list of previously defined port or internal signal names which defines the interconnection of components. This list may be of *named* or *positional* format. Named format is an unordered list of association of signals to ports. For example, if net N1 is attached to port P1 (as defined in the port list of the component declaration), N1 => P1 would appear in the <port-map> list. Positional format assigns elements of the <port-map> to ports with the corresponding position in the port list of the component declaration.

Concurrent assignment statements may be used to specify necessary behavior characteristics of a component. Examples of this type of specification include the concatenation of input signals to form output signals for the

switchbox component of Figure 10, or the specification of the functionality of a random logic component using boolean equations.

9.2. Interface to Other Synthesis Tools

Since the generic component netlist output by the VHDL synthesis system described in this report is of valid VHDL syntax, the opportunity exists to verify its correctness. If behavioral VHDL models are developed for each generic component, the netlist can be used as input along with these models to the VHDL simulator. A comparison to simulation results generated using the initial behavioral description can verify the proper operation of the synthesized netlist.

10. An Example

An example of a VHDL description of a Bus Interface circuit from the VHDL Tutorial [VHDL87] will be used to illustrate the Data Flow compilation process. Figure 12 shows the VHDL input description.

The Graph Compiler parses each individual statement as it is encountered, generating a corresponding flowgraph for that statement. After all statements have been processed, the flowgraph sections are interconnected to produce the flowgraph shown in Figure 13.

The Graph Critic is then invoked to optimize the flowgraph representation so that translation to generic microarchitecture components is straightforward. Figure 14 identifies sections of the initial flowgraph which are optimized through the application of Graph Critic rules. The actual rules that the Graph Critic will apply for this example are shown in Figure 15. The flowgraph that results after optimization appears in Figure 16.

Node compilers are then invoked for each of the remaining flowgraph nodes. The final generic netlist resulting after all nodes have been replaced by corresponding generic components is shown in Figure 17. A VHDL Generic Component Netlist as shown in Figure 18. This netlist describes the connectivity, component attributes and timing assertions of the interconnected microarchitecture components which comprise the design.

```
entity BusInterface
 (ABus: in BIT_VECTOR(0 to 31);
  DBus: out BIT_VECTOR(0 to 7);
  MemReq: in BIT;
  BusReq: out BIT;
  BusAck: in BIT;
  DataRdy: out BIT;
  Addr: out BIT_VECTOR(0 to 15);
  In_Data: in BIT_VECTOR(0 to 7);
  MR: out BIT)
is end BusInterface;

architecture DataFlow of BusInterface is

block

   signal Done,Enable: BIT;
   signal MRint: CLOCK;
   signal Board_id: integer;

begin

   DataRdy <= not Enable ;
   MR <= not MRint ;
   Enable <= Done and not BusAck ;
   BusReq <= not (Done and BusAck) ;
   Done <= MRint and MRint'DELAYED(175 ns);
   MRint <= MemReq and (ABus(16 to 18) = Board_id) ;
   with Enable select
      DBus <= In_Data when '1',
              "ZZZZZZZZ" when '0';

   block (MRint = '0' and not MRint'STABLE)

      signal Addr_reg: BIT_VECTOR(0 to 15) register;

   begin

      Addr_reg <= guarded ABus(0 to 15) ;
      Addr <= Addr_reg;

   end block;

end block;
end DataFlow;
```

Figure 12: VHDL Description of the Bus Interface Circuit

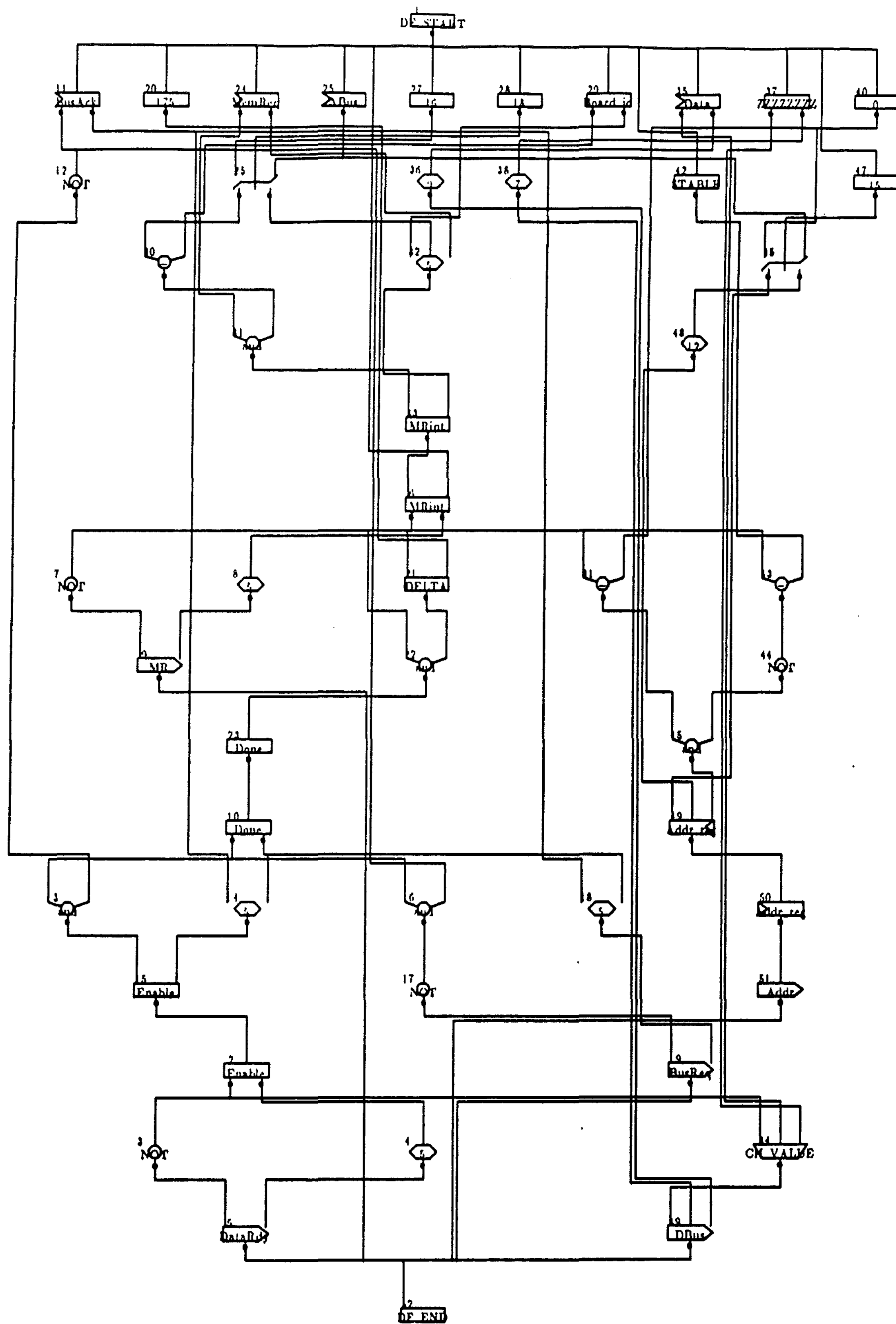

Figure 13: Interconnected Flowgraph for Bus Interface Example

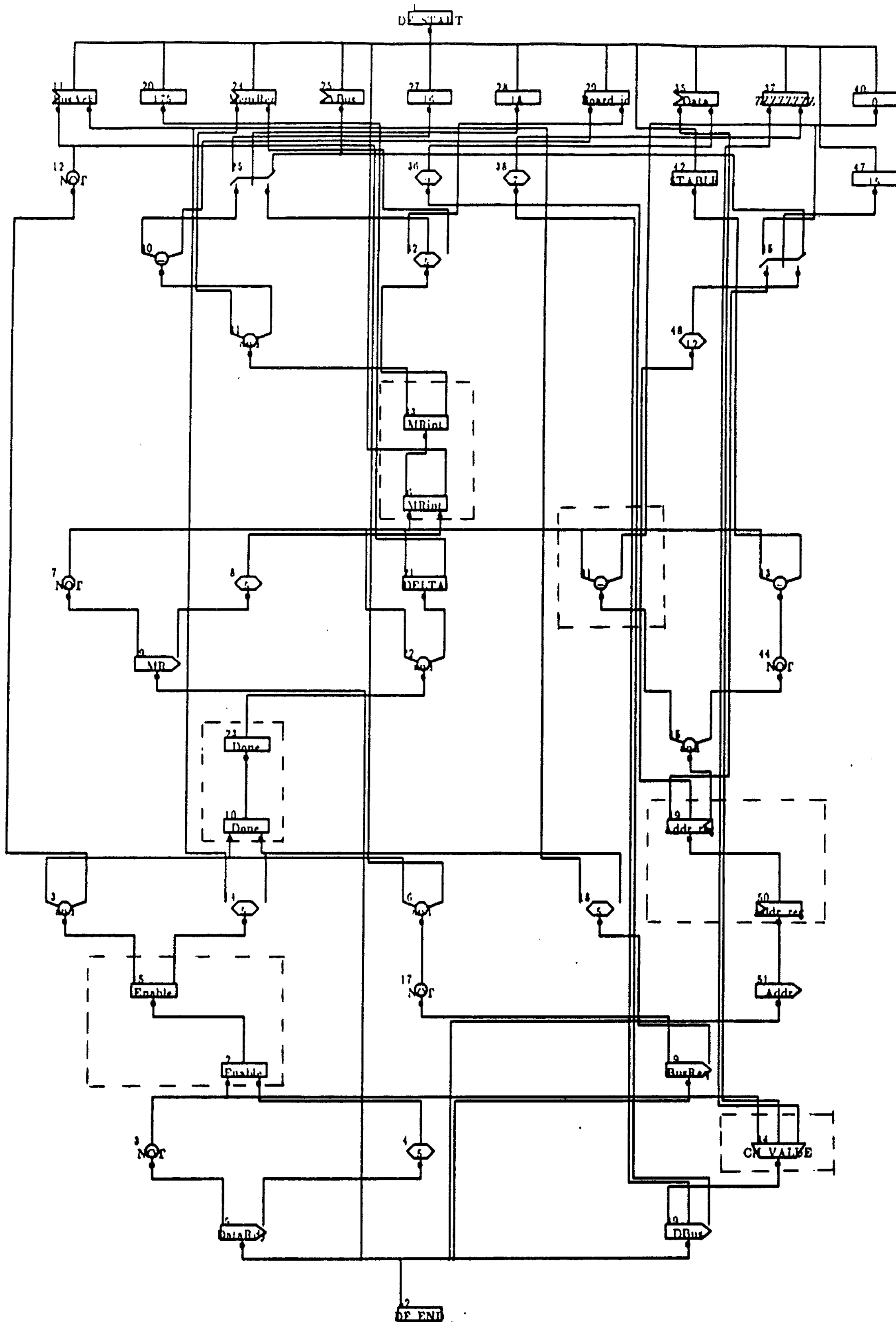

Figure 14: Graph Critic Rule Applications

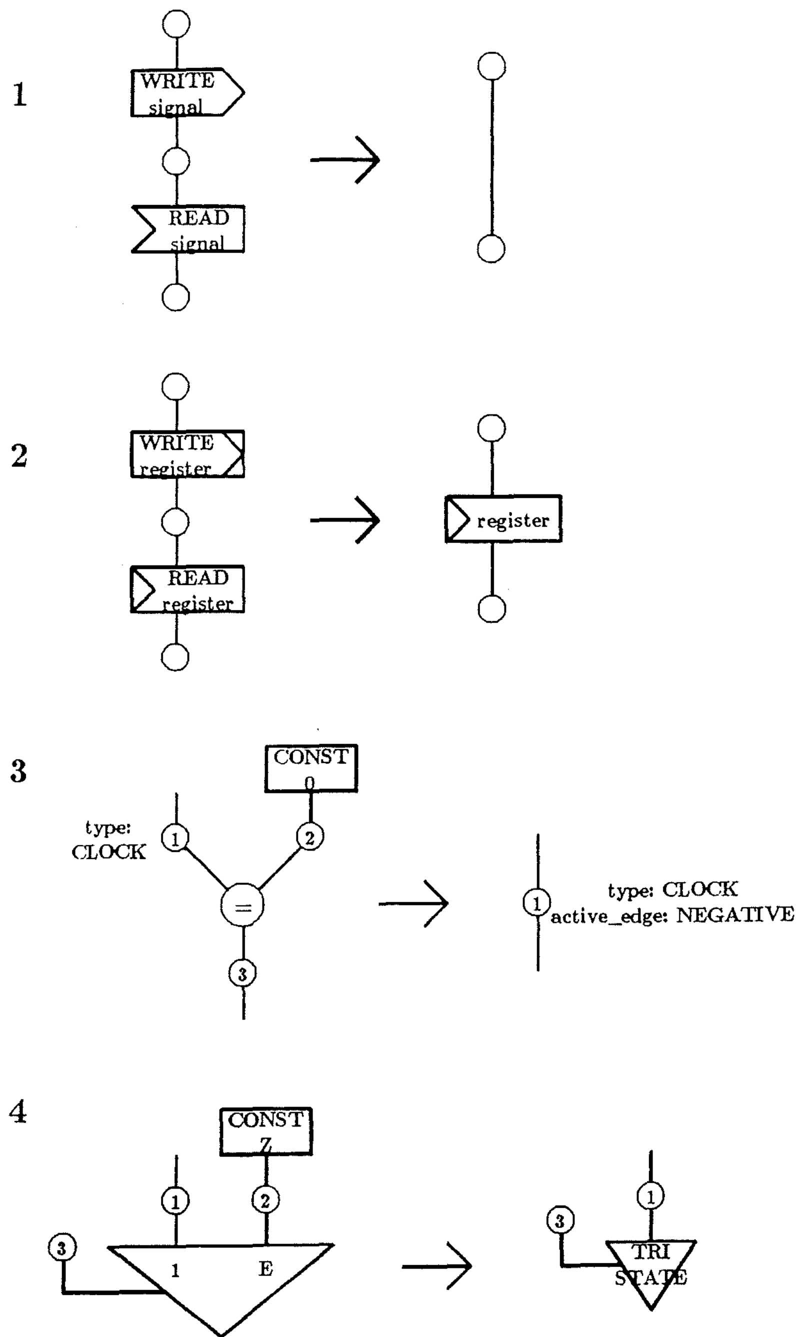

Figure 15: Graph Critic Rules

Figure 16: Flowgraph after Optimization

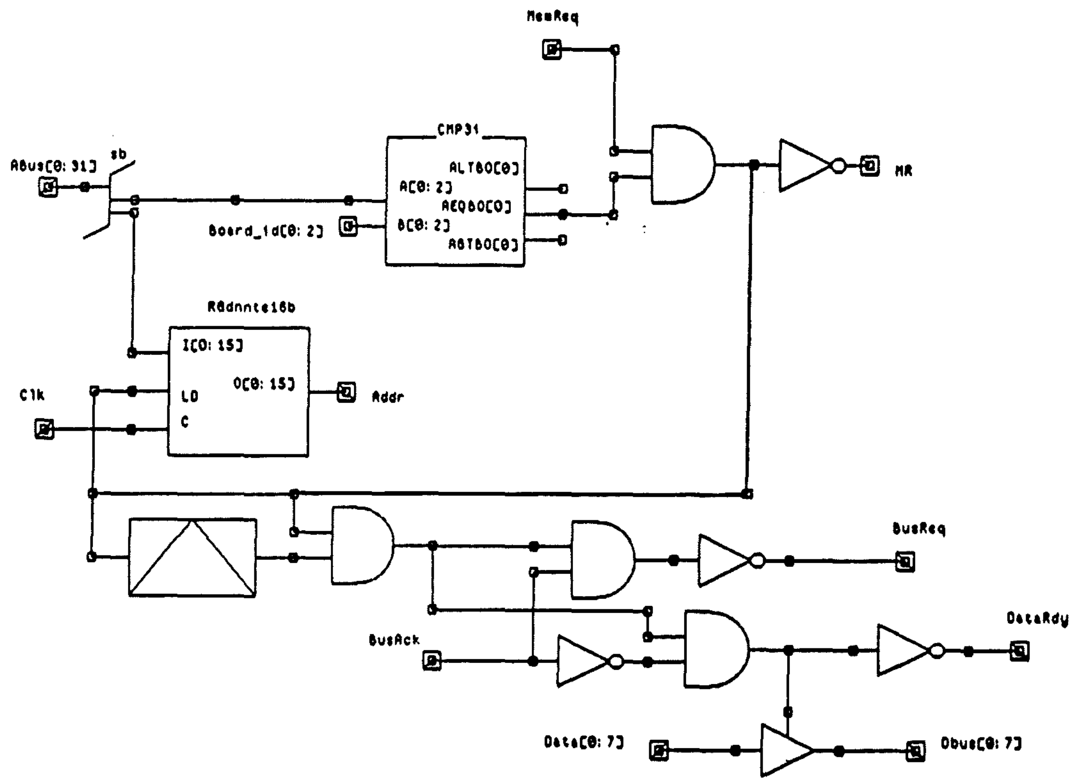

Figure 17: Generic Component Schematic for Bus Interface Example

```
entity BusInterface is
 generic (Board_id: in BIT_VECTOR(0 to 2));
 port
  (ABus: in BIT_VECTOR(0 to 32);
   DBus: out BIT_VECTOR(0 to 7);
   MemReq: in BIT;
   BusReq: out BIT;
   BusAck: in BIT;
   DataRdy: out BIT;
   Addr: out BIT_VECTOR(0 to 15);
   Data: in BIT_VECTOR(0 to 7);
   MR: out BIT)

--T Global timing assertions

end BusInterface;

architecture Structure_View of BusInterface is

   Component AND2
      port (I0,I1: in BIT; O: out BIT);
   Component INV
      port (I0: in BIT; O: out BIT);
   Component REG
      port (I: in BIT_VECTOR(0 to 15);
            CLK: in BIT;
            O: out BIT_VECTOR(0 to 15));
   Component CMP
      port (A,B: in BIT_VECTOR(0 to 2);
            AEQB: out BIT);
   Component BUF
      port (I: in BIT_VECTOR(0 to 7);
            EN: in BIT;
            O: out BIT_VECTOR(0 to 7));
   Component ONE_SHOT
      port (I: in BIT; O: out BIT);

   signal Done,Enable,MRint: BIT;
   signal S1,S2,S3,S4: BIT;

begin

   C1: AND port map (S1, MemReq, MRint);
   C2: AND port map (MRint, S4, Done);
   C3: AND port map (Done, BusAck, S2);
   C4: AND port map (S3, Done, Enable);
   C5: INV port map (MRint, MR);
   C6: INV port map (S2, BusReq);
   C7: INV port map (BusAck, S3);
   C8: INV port map (Enable, DataRdy);
   C9: REG port map (ABus(0 to 15), MRint, Addr);
   C10: CMP port map (ABus(16 to 18), Board_id, S1);
   C11: ONE_SHOT port map (MRint, S4);
   C12: BUF port map (Data, DBus);

end Structure_View;
```

Figure 18: VHDL Generic Component Netlist

11. Conclusions

In this technical report, we have presented a methodology for design synthesis from a VHDL description. The implementation of this methodology resulted in a system that increases designer productivity by removing the requirement of expert knowledge at lower levels of design. Designers can experiment by modifying design descriptions and parameters in order to evaluate alternative styles and target technologies.

The VHDL input description language was restricted and extended to incorporate information necessary for synthesis. A well defined design representation, the control/data flowgraph, incorporates signal typing and other attributes. The flowgraph also provides a canonical form for a many to one mapping of equivalent language constructs to a unique representation. Graph optimizations are performed at various stages of the synthesis process, leading to a near optimal design.

This approach to synthesis decomposes the process into two interacting stages: compilation and optimization. Compilation is procedural in nature and involves the generation of the design representation (flowgraph) given the input behavioral description. Emphasis is placed on creating a design data base which includes sufficient information for synthesis such as signal functionality and

timing relationships. Optimizations are heuristic in nature and are performed most effectively when the necessary information is accessible. Local graph optimizations such as recognition of signal attributes can modify the representation so that optimal generic component mapping will result; global optimizations can be performed more easily if information such as critical path specifications or propagation delays remain associated with generic structural components. This avoids the difficult task of attempting to optimize a structural design which has been synthesized directly from a suboptimal behavioral description and has lost information during the translation process.

The system remains technology independent through the use of a generic component library. Unlike systems which assume a limited design model which is hard coded into the software, the flexibility of this approach allows for the addition or modification of the generic component library by simply changing the input generic component table. The design can then be translated to technology components by a mapper and logic optimizer. Simple redesign to new technologies can be accomplished by changing the technology specific component library.

12. References

[Arms87] J. R. Armstrong, *Chip Level Modeling with VHDL*, draft, June 1987.

[Arms88] J. R. Armstrong, "Modeling with HDLs", IEEE Design & Test, pp. 8-18, February 1988.

[Barb81] M. Barbacci, "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications", IEEE Transactions on Computers, Vol. C-30 no. 1, January 1981.

[Bhas86] J. Bhasker, "Process Graph Analyzer: A Front End Tool for VHDL Behavioral Synthesis", Proceedings of the 10th Annual Honeywell International Computer Sciences Conference, Minneapolis, 1986.

[BrHa84] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. Gangiovanni-Vincentelli, *EXPRESSO-IIC: Logic Minimization Algorithms for VLSI Synthesis*, Kluwer Academic Publishers, 1984.

[BrRu87] R. K. Brayton, R. Rudell, A. Sangiovanni-Vincentelli and A. R. Wang, "MIS: A Multiple-Level Logic Optimization System", IEEE Trans. CAD, Vol. CAD-6, No. 6, Nov. 1987.

[BrGa86] F. D. Brewer and D. D. Gajski, "An Expert-System Paradigm for Design", 23rd DAC, June 1986.

[dGCo85] A. J. deGeus and W. Cohen, "A Rule-Based System for Optimizing Combinational Logic", IEEE Design & Test, pp. 22-32, Aug. 1985.

[FoFr85] J. R. Fox and J. A. Fried, "Telecommunications Circuit Design Using the SILC Silicon Compiler", ICCAD, June 1985.

[Gajs85] D. Gajski, *DESCART System*, prepared for Gould Research Center, August 1985.

[GOKB86] D. Gajski, A. Orailoglu, B. Kuhn, C. Bosco, "An Expert Silicon Compiler", IEEE 1986 Custom Integrated Circuit Conference, pp. 116-119.

[GrBa86] D. Gregory, K. Bartlett, A. deGeus, G. Hatchel, "SOCRATES: A System for Automatically Synthesizing and Optimizing Combinational Logic",

23rd DAC, June 1986.

[JVJC86] A. A. Jerraya, P. Varinot, R. Jamier, and B. Curtois, "Principles of the SYCO Compiler", 23rd DAC, June 1986.

[JoTr86] W. Joyner, Y. Trevillyan, D. Brand, T. Nix, S. Gundersen, "Technology Adaptation in Logic Synthesis", 23rd DAC, June 1986.

[Kim87] J. Kim, "Artificial Intelligence Helps Out ASIC Design Time", Electronic Design, June 1987, pp. 107-110.

[KBhN] S. J. Krolikoski, J. Bhasker, S. Natarajan, "V-SYNTH: A VHDL Behavioral Synthesis System".

[KoTh85] T. J. Kowalski and D. E. Thomas, "The VLSI Design Automation Assistant: What's in a Knowledge Base", 22nd DAC, June 1985.

[LiGa87] S. Lin and D. D. Gajski, "LES: A Layout Expert System", 24th DAC, June 1987.

[McFa78] M. McFarland, "The Value Trace: A Data Base for Automated Digital Design", Master Thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, December 1978.

[OrGa86] A. Orailoglu, D. Gajski, "Flow Graph Representation", DAC, 1986.

[PaGa86] B. M. Pangrle and D. D. Gajski, "State Synthesis and Connectivity Binding for Microarchitecture Compilation", ICCAD, Nov. 1986.

[PaGa87] B. M. Pangrle and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", IEEE Trans. on CAD, vol CAD-6, no. 6, Nov. 1987.

[PTSB79] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L. Hafer, G. Lieve, and J. Kim, "The CMU Design Automation System: an Example of Automated Data Path Design", 16th DAC, June 1979.

[PaKM84] A. Parker, F. Kurdahi, and M. Mlinar, "A General Methodology for Synthesis and Verification of Register-Transfer Designs", 21st DAC, June 1984.

[PaPM86] A. C. Parker, J. Pizarro, M. Mlinar, "MAHA: a Program for Datapath Synthesis", 23rd DAC, June 1986.

[Saun87] L. Saunders, "The IBM VHDL Design System", 24th DAC, June 1987.

[Sout83] J. R. Southard, "MacPitts: an Approach to Silicon Compilation", IEEE Computer, vol. C-16, Dec. 1983.

[THKR83] D. E. Thomas, C. Y. Hitchcock III, T. J. Kowalski, J. V. Rajan, and R. A. Walker, "Automatic Data Path Synthesis", IEEE Computer, Dec. 1983.

[TsSi83] C. J. Tseng and D. P. Siewiorek, "Facet: A Procedure for the Automated Synthesis of Digital Systems", 20th DAC, June 1983.

[TsSi84] C. J. Tseng and D. P. Siewiorek, "Emerald: A Bus Style Designer", 21st DAC, June 1984.

[TsSi86] C. J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Paths in Digital Systems", IEEE Trans. Computer-Aided Design, vol. CAD-5, no. 3, July 1986.

[TsWe88] C. J. Tseng, Ruey-Sing Wei, et. al., "Bridge: A High Level Synthesis System in Industry", 25th DAC, June 1988.

[VHDL87] *VHDL Language Reference Manual, Draft Standard 1076/B*, IEEE, June 1987.

[VZGa88] N. VanderZanden and D. Gajski, "MILO: A Microarchitecture and Logic Optimizer", 25th DAC, Anahiem, CA, June 1988.

[Wei88] R. S. Wei, "BECOME: Behavior Level Circuit Synthesis Based on Structure Mapping", 25th DAC, June 1988.

[Zimm80] G. Zimmerman, et. al., *MDS - The MIMOLA Design Method*, Journal of Digital Systems, Volume IV Issue 3, 1980.