The Automatic Generation of Bus-Interface Models*

Ajay J. Daga, William P. Birmingham
EECS Department,
The University of Michigan,
Ann Arbor, MI, 48105
(313)-936-1590

Abstract

This paper describes HIDE, a system that automatically generates VHDL and Verilog bus-interface models from a high-level specification of interface behavior. HIDE users need not be familiar with VHDL or Verilog, and instead specify interface behavior using familiar hardware constructs, such as timing diagrams, state diagrams and truth tables. We also introduce a hierarchical model of interface behavior that supports the model-generation task.

1. Introduction

Simulation of digital systems is common design practice. For simulation to be effective, accurate models of components must be available. Components can be described by *behavioral* and *bus-interface models* (BIMs).

Behavioral models capture the full functionality of a component, *i.e.*, both the computation and interface functions are described. For components such as microprocessors, these models realize the complete instruction set, allowing small programs to be executed. Behavioral models are difficult to construct, since all details of the component's behavior must be known.

BIMs, also known as hardware-verification models [1], bus-functional models [2], and chip-level models [3] describe only the interface behavior of a component. Essentially, these models capture the communication activity of a component. BIMs treat internal circuitry as a black box, while modeling how a component communicates with its environment.

Since BIMs implement a subset of a component's behavior, they are concise, and can be executed quickly on general-purpose hardware. As such, BIMs fit very well with the typical computer-aided engineering (CAE) environment, where a designer captures a design as a schematic, and performs logic simulation of that design. BIMs allow a user to ensure that a component communicates correctly with the components it is connected to for a set of test vectors supplied by the user.

Though less complex than behavioral models, BIMs can be difficult to build because of the complexity of the interface behavior of VLSI components. Development of simulation models for highly sophisticated VLSI components may require several man-months of effort. This effort is spent in three places: understanding how the component operates from its documentation, understanding the syntactic details of a hardware-description language (HDL), and writing the model. Often, models are produced by third parties who are not developers of the component being modeled. For these groups, understanding the operation of a component is a major undertaking; consider that complete descriptions of the operation of complex microprocessors fill several hundred pages. The modeling process is further hampered by HDLs. While HDLs are intended to facilitate model creation, they are sometimes cumbersome and complex, and require considerable programming skill to be effectively used. Thus, a model writer must be an expert in both hardware and an HDL.

*This research was supported by Digital Equipment Corporation and the National Science Foundation grant MIPS-905781. All views expressed here are those of the authors, and not necessarily those of the funding agencies.

To alleviate the problems associated with building BIMs, it is useful to automate their generation from high-level specifications. Some of the desirable features of automated-model generation are the following:

- A *succinct* methodology for the specification of interface behavior, from *constructs familiar to a hardware engineer* [4, 5].
- A *substantial reduction* in the time taken to generate a model.
- A *method for generating an executable specification of component behavior* (prior to the realization of this behavior in hardware), based on desired signal activity at the interface of a component. This specification may be easily modified, if required, during the design process.

In this paper, we present a tool, HIDE (HDL Interface Models Designer) [6], that automatically generates BIMs. Interface behavior is specified graphically using a tool, SpecIT (Specification of Interface Transactions), composed of a timing-diagram editor (Xwave), a state-diagram editor (Xstate), and a truth-table editor (Xtable). Both HIDE and SpecIT use a hierarchical model of interface behavior. HIDE is a tool that interprets an interface specification, identifies syntactic and semantic errors, and generates HDL code that encapsulates behavior contained in the specification.

This paper is organized as follows. Section 2 defines our model of interface behavior, and Section 3 outlines how this behavior is captured using SpecIT. Section 4 discusses the model-generation process. Section 5 contrasts HIDE with related work. Section 6 presents experimental results of the application of SpecIT and HIDE to model generation tasks, and Section 7 summarizes the paper. We illustrate the model-generation process for the Intel 8086 microprocessor [9].

2. Model of Interface Behavior

A component may be viewed as consisting of an *interface* and *internal circuitry*, as shown in Figure 2.1.

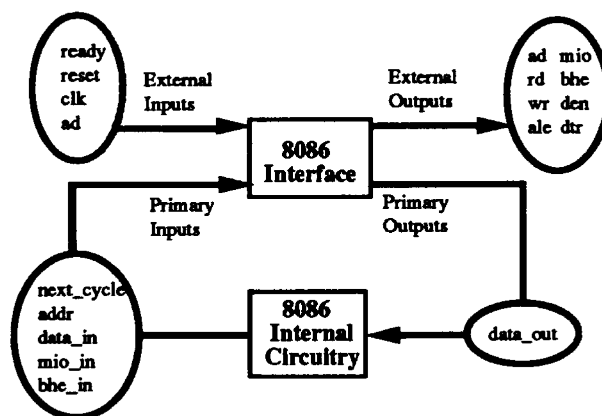

Figure 2.1: Component model

A component interface consists of *external* and *internal* signals that either receive (input signals) or transmit (output signals) information[1]. External signals are used to connect one or more component interfaces with each other. Internal signals are *primary input* (receive external stimulus) or *primary output* (provide external stimulus) signals to a component interface. These signals specify the stimulus that may be applied (received) to (from) a component interface by its internal circuitry. For example, in

[1] A bi-directional signal may be viewed as a pair of input and output signals.

Figure 2.1, *addr*[2] is a primary-input signal to the 8086 interface, and determines the value placed on the time-multiplexed output signal *AD*. External and internal signals may have widths greater than one, as is the case for *addr* and *AD*.

Component interface behavior is defined by *signal activity* on its external signals. Signal activity refers to *digital* signal values, *events* (changes in signal values), and *temporal relations* between events. Digital signal values are 0, 1, Active (any valid signal value), and Z (high impedance). Event *types* are $0 \rightarrow 1$ (rise), $1 \rightarrow 0$ (fall), *etc.* An event on an output (input) signal is called an *output event* (*input event*). Two or more sequences of signal activity are *equivalent* if the same number of events are generated on each signal, and the delay associated with each event is the same; otherwise they are *unique*. Interface behavior is *synchronous*, if events occur relative to the edges of a clock signal, or *asynchronous*, if events are not clocked.

Interface behavior is modeled hierarchically through *cycles, states*, and *signal activity associated with states* as shown in Figure 2.2.

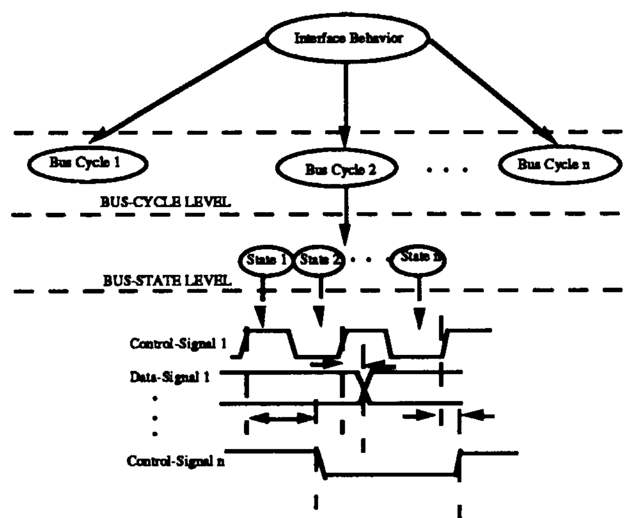

Figure 2.2: Hierarchical model of interface behavior.

At the most abstract level, the *cycle level*, component-interface behavior is partitioned into a collection of *cycles*. Intuitively, a cycle defines a sequence of signal activity that taken together specifies, from initiation to completion, a meaningful interface transaction. The signal activity initiated by each cycle is unique. For example, the interface behavior on the 8086 includes *read, write, idle*, and *reset* cycles.

Each cycle has an *initiation condition* that is a Boolean expression composed of *cycle-control signals*. Cycle-control signals are a subset of the internal and external signals on a component interface. The initiation condition for each cycle is unique. Therefore, for any assignment of cycle-control signal values exactly one cycle is enabled, and the same cycle is always enabled for the same assignment of cycle-control signal values. The unique initiation condition for the 8086 cycles, for example, is *next_cycle* = <cycle_name>.

A cycle may be of type *master*, or *slave*. If the initiation condition of a cycle is dependent on the value of a cycle-control signal that is a primary-input signal then the cycle is of type *master*, and if not the cycle is of type *slave*.

[2]Primary input and output signals are shown in lower case.

A cycle may initiate *multiple* unique signal activities. If so, it is decomposed into one or more *states*, each of which initiate only one unique signal activity; by default, a cycle is composed of one state if it initiates only one unique signal activity. Sequencing within a cycle, between states, is specified at the *state level* through a state diagram such as that shown in Figure 2.3 for the 8086 *read* cycle. A cycle may have one or more start and final states. The start state for the 8086 *read* cycle is *T1* and the final state *T4*. Signals that are used to specify sequencing at the state level are called *state-control* signals, and are disjoint from cycle-control signals. The state-control signal for the 8086 is *ready*. State diagrams are represented using a *state-transition graph*.

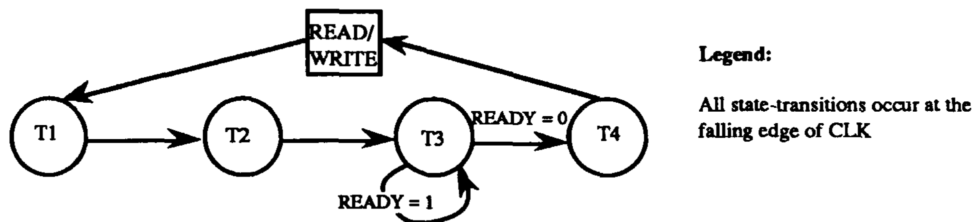

Figure 2.3: Example state diagram

Signal activity associated with a state, in general, specifies the following:

- The occurrence of output events as a function of input events, and input signal values. For example, the *RD* signal, in Figure 2.4 (signal activity for all states in the 8086 *read* cycle), goes to level 1 when the *read* cycle is in state *T4*, which, in turn, is dependent on the *READY* signal value.

- *Causal links* that specify the minimum *and* maximum temporal separation of an *output event* e_{to} relative to an event e_{from}[3]. The timing link T_{clrh} associated with *RD* going to level 1, in Figure 2.4, is an example of a causal link.

- *Timing constraints* that specify the minimum (*min-constraint links*) *or* maximum (*max-constraint links*) permissible temporal separation of an *input event* e_{to} relative to an event e_{from} for an interface to *function* correctly. In Figure 2.4, the timing link T_{clcl} yields a min-constraint and max-constraint link that specify the period of the *CLK* signal.

- *Sampling constraints* that are composed of a pair of min-constraint links, t_s (set-up) and t_h (hold), and specify temporal separations that have to be satisfied to *correctly capture signal values*. These links apply on *input events* e_1, e_2, and e_s such that: e_1 and e_2 are consecutive events (e_1 occurs before e_2) on a signal s_1, and have timing links t_s and t_h, respectively, to a common event e_s on a signal s_2 ($s_1 \neq s_2$); e_s is called the sampling event, and captures the level l_s, between e_1 and e_2. In Figure 2.4 links T_{ryhcx} (t_s), and T_{chryx} (t_h) specify the sampling constraints associated with the *READY* signal.

Signal activity associated with a state is represented using an *event graph* [7, 10]. An event graph is a directed acyclic graph (DAG) whose nodes, $E = \{e_1, e_2 ... ,e_n\}$, represent events, and arcs, $A = \{t_1, t_2 ... ,t_m\}$, represent timing links between pairs of events. A causal link is represented by a directed arc from e_{from} to e_{to}. A constraint link is represented by a directed arc from e_{to} to e_{from}. The events and arcs associated with a sampling constraint are grouped to form a single entity. A subset of the events on an event graph serve as temporal reference points for output events. These *reference events* define the *range of occurrence* of output events relative to reference events. The range of occurrence, $t_p(e_o)$, of an output event e_o has the form, $t_{min}(e_o) \leq t_p(e_o) \leq t_{max}(e_o)$. The time $t_{min}(e_o)$ ($t_{max}(e_o)$) depicts the earliest (latest)

[3] The event e_{from} occurs, or is required to occur, before e_{to}.

that event e_0 could occur relative to a reference event e_r. Figure 2.5 shows the event graph for the state *T4* of the 8086 *read* cycle.

Figure 2.4: Example timing diagram

Legend:

- Nodes, and arcs, that are part of a sampling constraint are shown in bold.
- Causal arcs are shown
- Constraint arcs are shown

Figure 2.5: Example event graph.

A truth table is used to specify signal values, and those aspects of interface behavior that are a Boolean or arithmetic function of signal values. For each row, the LHS of a truth table specifies conditions that need to hold for the functions specified on the RHS to be applicable. Boolean and arithmetic operations between signals may be specified in the RHS of a truth table. An example truth table, necessary to specify the multiplexing of data values on the 8086 signal AD as a function of address alignment, is shown in Figure 2.6.

LHS		Operations
BHE	addr[0]	
0	0	data_out[15:0] = AD[15:0]
0	1	data_out[7:0] = AD[15:8]
1	0	data_out[7:0] = AD[7:0]
1	1	data_out[15:8] = AD[7:0]

Figure 2.6: Example truth table

3. Specification Methodology

In this section, we define the sequence of steps performed in specifying component-interface behavior using SpecIT.

A user first defines the high-level model of a component. This requires specifying whether the component is asynchronous, or synchronous (in this situation a user also specifies the *clock* signal). In addition a user provides the following information on a component interface: internal and external signals, cycles and their initiation conditions, and *operating conditions*. By operating conditions, we refer to parameters such as operating frequency that affect only the delay values associated with timing links, and no other aspect of interface behavior. For example, the 8086 has operating frequencies of 5, 8, and 10 MHz.

Next, a user specifies sequencing within a cycle using Xstate. Multiple cycles may be specified through the same state diagram if they are composed of the same number of states and sequencing between states is the same. A user defines the states associated with a cycle and enters the state-transition arcs and their conditions. In addition, for a synchronous interface, a user specifies the event on the clock signal (rise or fall) that triggers state-transition

This is followed by specifying the timing diagrams associated with each state using Xwave. For each diagram a user selects the signals for which activity is defined by the state, and then draws the waveforms associated with each signal. A user specifies the timing links between events, and defines their values for each operating condition. Xwave uses the definitions of causal links, constraint links, and signal sampling, (refer Section 2) to automatically cull these semantics from a timing diagram.

Finally, a user specifies the truth tables associated with cycles on a component interface using Xtable.

4. Model Generation

HIDE generates BIMs in VHDL [11] and Verilog [12][4]. The modeling style employed by HIDE closely resembles the *process-model graph* structure outlined by Armstrong [3]. A BIM is implemented as a set of processes that execute concurrently. A process is generated for the timing diagrams associated with a cycle, each truth table and each state diagram. Code generation for each process is modular and independent of the code generated for other processes.

Each process has a set of signals on its sensitivity list. An event on any of these signals results in the execution of the process. Depending on the status of the input signals on a component during a simulation run, any number of processes may be active at a given time.

---

[4] In this section, examples of HIDE generated code will be given using VHDL syntax.

Processes communicate with each other through buffers that contain the value driven or received by a signal. A buffer (*<signal-name>_buf*) exists for every external signal on a component interface. Only processes generated from timing diagrams directly drive or receive values from signals on an interface. When driving a signal, a process sends the value contained in the signal buffer to a signal. When sampling a value on a data or control signal, a process stores the received value in the signal buffer. Processes generated from state diagrams use the buffer value of signals to update the state of a cycle. Processes generated from truth tables use and affect the buffer value of signals.

In addition to signal buffers, the following predefined signals are contained in a model:

- A flag (*<signal-name>_sample*), associated with all input and bi-directional external signals, that is toggled when the signal is sampled.

- A signal (*state*) that stores the current state of a cycle.

- A flag (*trans*) that indicates the occurrence of a state transition.

- A signal (*cycle*) that indicates the current cycle being executed, and another (*next_cycle*) that indicates the next cycle to be executed.

The following sections discuss the structure of processes generated for timing and state diagrams, and truth tables.

4.1 Processes for Timing Diagrams

Code generation examines event graphs associated with each state of a cycle. For each event graph, code is generated for those events that have a causal or constraint arc leading out of them. It is also possible for an event to sample a signal or set of signals. In this case, code to implement sampling is generated for each of the sampled signals.

Code generation commences with a creation of the sensitivity list for the process. This list consists of signals whose events are either reference events, or input events with constraint links on any event graph associated with the cycle. If a cycle has a state diagram associated with it, then the signal *trans* is included in the sensitivity list.

Each causal arc results in the generation of a *signal-assignment statement* for the non-reference event e_i connected to the reference event e_r . A constraint arc results in the generation of an *assertion statement* that checks the temporal separation between a pair of events. An event that is used to sample a signal results in code that stores the value of the sampled signal in that signal's buffer. The value of the corresponding *<signal_name>_sample* signal is toggled to indicate that the signal has been sampled. Each of these statements is nested within a conditional statement that tests for the occurrence of the event from which these arcs fan out. An illustration of these different statements is shown in Figure 4.1 for the event graph shown in Figure 2.5.

4.2 Processes for State Diagrams

The signals placed in the sensitivity list of a process generated from a state diagram depend on whether the component interface is synchronous or asynchronous. For a synchronous interface the sensitivity list contains only the clock signal. For an asynchronous interface the sensitivity list contains all signals that are used in state-transition conditions. The sensitivity list of the process generated for the state diagram in Figure 2.3 contains the signal *CLK*.

```
READ : process (AD, READY, trans)
begin
    if cycle = 'READ' then
        •••
        if trans'event and state = T4 then        /* code associated with state T4 */
            RD <= transport '1' after Tclrh;       /* signal assignment statement */
            RD_buf <= transport '1';
            DEN <= transport '1' after Tcvctv     /* signal assignment statement */
            DEN_buf <= transport '1';
            assert AD'last_event > Tdvcl          /* assertion statement */
            report "Timing Violation";
            AD_buf := AD;                          /* code as a result of a sampling arc */
            AD_sample <= not AD_sample;          /* used to indicate that AD has been sampled */
        end if;
        •••
end process READ;
```

Figure 4.1 Process generated from a timing diagram.

Each statement generated for a state diagram checks the current state and evaluates the state-transition conditions associated with the state. If a state-transition condition is satisfied then the state is updated to its new value. Also, the *trans* signal is toggled to indicate that a state transition has been made. For a transition that indicates the completion of a cycle as many conditional statements are generated as cycles. Each of these conditional statements evaluate the initiation condition of cycles to determine the next enabled cycle. The process generated for the state *T1* in Figure 2.3 is shown in Figure 4.2.

```
READ_STATE_TRANSITION : process (CLK)
begin
    if cycle = Read or cycle = Write then
        if CLK'event and CLK = '0' and state = T1 then
            state <= T2;
            trans <= not trans;                /* indicate that a state transition has been made */
        end if;
        •••
    end if;
end READ_STATE_TRANSITION;
```

Figure 4.2: Process generated from a state diagram.

4.3 Processes for Truth Tables

The sensitivity list for a process generated from a truth table consists of sampling flags associated with the following signals:

1. Those signals on the LHS of a truth table that are input to a component.
2. Those signals on the RHS of an action associated with a row in a truth table.

These signals represent *data flow* constraints on the actions in a truth table. Recall that a sampling flag for a signal is set when that signal is sampled by a process generated from a timing diagram. When all sampling flags associated with a process generated from a truth table have been set, then the actions associated with a row in the truth table are performed. The code generated for each row of the truth table tests whether the row conditions are satisfied, and if so executes the actions associated with that row. A signal whose value is X for a row in the truth table does not form part of the

conditional statement. The process generated for a portion of the truth table in Figure 2.6 is shown in Figure 4.3.

```
TRUTH_TABLE_READ  : process (AD_sample)
    variable AD_sample_flag: integer := 0;          /* flag used to indicate whether AD has been sampled */
begin
    if  AD_sample'event then
        AD_sample_flag := 1;
    end if;
    •••
    if AD_sample_flag = 1 then
        if bhe_in= '1' and addr(0) = '0' then
            data_out(7 downto 0) := AD_buf (7 downto 0);
        end if;
        •••
    end if;
    AD_sample_flag := 0;
end TRUTH_TABLE_READ;
```

Figure 4.3: Process generated from a truth table.

5. Related Work

General approaches have been taken toward the automated generation of simulation models. HUM [13] generates VHDL models representing the behavior of both the computation and interface engine of a component. HUM requires a spreadsheet-like specification of the behavior of a component. XBIF [14] is a user interface that provides both graphical and tabular specification formats of component behavior, which may then be converted to VHDL or retained in an internal [15] representation for synthesis tasks. Both XBIF and BIF rely on the specification and representation, respectively, of behavior using hierarchical annotated state tables. While the use of state tables is appropriate for specifying control flow, we believe that timing diagrams are necessary to capture the temporal behavior that is an integral part of an interface specification.

6. Results

HIDE has been used to generate models for the following components: Intel I8086 and I80386 CPUs, Motorola MC68020 and MC68332 CPUs, and a Cypress Semiconductor memory component, CY7C147. HIDE has also been used to generate BIMs for other non-commercial interface specifications, such as read and write cycles performed using a two-wire handshake protocol, that exercised the capabilities of the tool. The model-generation time using HIDE is a few *man-days*, in contrast to a few *man-months* when the same task is performed manually. Most of the development time is spent in understanding component-interface behavior; the execution time of HIDE is of the order of a few seconds. Figure 6.1 summarizes results of the model-generation process. Note that the MC68332 model, developed after the MC68020, required fewer man days to generate because of its similarity to the MC68020; much of the effort spent in understanding and specifying the MC68020 was applicable for the MC68332.

Verilog models were, on average, 10% longer than VHDL models. The reason for this is that processes generated from timing diagrams for Verilog models have only a single signal on their sensitivity lists, while VHDL models have no such limitation. Consequently, there are many more processes required to capture the signal activity of each cycle for Verilog models than for VHDL models. In addition to the processes described in Section 4, all VHDL models had additional statements that specified the time-multiplexing of signal values using a scheme outlined by Armstrong [3]; each statement required three

lines of code per external signal. Verilog models, in turn, each had a library of routines (75 lines of code) that checked for timing-constraint violations.

DEVICE	GRAPHICAL INPUTS			LINES OF CODE		MAN DAYS
	STATE DIAGRAMS	TIMING DIAGRAMS	TRUTH TABLES	VHDL	VERILOG	
MC68020	2	6	2	1345	1492	7
MC68332	3	8	2	1368	1524	3
I8086	1	3	4	503	560	3
I80386	2	8	4	1051	na	10
CY7C147	0	2	0	332	371	1
HANDSHAKE	0	2	0	316	405	1

Figure 6.1: Results of the Application of SpecIT and HIDE

7. Summary

In this paper, we have presented a tool HIDE which generates BIMs from a high-level specification of interface behavior (SpecIT) using constructs that are familiar to a hardware engineer. HIDE and SpecIT use a hierarchical model of interface behavior. Our methodology for interface specification is consistent with standard logic design methodologies. HIDE has been used to generate models for a wide range of components, and experimental results indicate that it significantly reduces the time taken to generate BIMs.

References

[1] Logic Automation, Inc. Processor Control Language, Users Guide.
[2] D. Coelho. The VHDL Handbook. *Kluwer Academic Publishers*, 1989.
[3] J. Armstrong. Chip-Level Modeling with VHDL. *Prentice-Hall Inc.*, 1989.
[4] W.I. Fletcher. An Engineering Approach to Digital Design. *Prentice-Hall, Inc.*, 1980.
[5] M. McFarland. CPA: Giving an Account of Timed System Behavior. In *Proceedings of TAU'90*, August 1990.
[6] W.P. Birmingham and Yew-Hong Leong. The Automatic Generation of Bus-Interface Models. In *Proceedings of the 29th Design Automation Conference*, June 1992.
[7] G. Borriello. A New Interface Specification Methodology and its Application to Transducer Synthesis. *Technical Report UCB/CSD 88/430 (PhD Dissertation)*, Computer Science Division, University of California at Berkeley, May 1988.
[8] H.S. Stone. Microcomputer Interfacing. Addison-Wesley Publishing Company, February 1983.
[9] Intel, Corp. Microprocessor and Peripheral Handbook (Volume 1), 1988.
[10] A.R. Martello and S.P. Levitan. Causal Timing Verification. In *Proceedings of TAU'90*, August, 1990.
[11] Institute of Electrical and Electronics Engineering, *IEEE Standard VHDL Language Reference Manual, IEEE Std. 1076-1987*, 31 March 1988.
[12] D.E. Thomas and Philip Moorby. The Verilog Hardware Description Language. *Kluwer Academic Publishers*, 1991.
[13] Lewis System, Inc. HUM Users' Manual, VHDL PC Version, July 1991.
[14] N. Dutt, J. Cho and T. Hadley. A User Interface for VHDL Behavioral Modeling. In *Proceedings of the Tenth International Conference on Computer Hardware Description Languages and their Applications (CHDL '91)*, IFIP April 1991.
[15] N.D. Dutt, D.D. Gajski and T. Hadley. BIF: A Behavioral Intermediate Format for High Level Synthesis. *ICS Technical Report 89-03*, University of California at Irvine, February, 1989.