BIF:
A Behavioral Intermediate Format For High Level Synthesis

BY

Nikil D. Dutt
Tedd Hadley
Daniel D. Gajski

Technical Report 89-03 (**Revised 9/19/89**)

Information and Computer Science
University of California at Irvine
Irvine, CA 92717

Abstract

BIF is a new intermediate format for behavioral synthesis systems, based on annotated state tables. It is unique in supporting user control of the synthesis process by allowing specification of partial design structures, user-bindings and user modification of compiled designs. It captures synchronous and asynchronous behavior, permits hierarchy and concurrency, and serves as a good interface to the user by linking behavior and structure at each level of abstraction in the behavioral synthesis process. BIF's simple and uniform syntax allows it to be used as an intermediate exchange format for various behavioral synthesis tools.

TABLE OF CONTENTS

CHAPTER

CHAPTER

LIST OF FIGURES

CHAPTER 1.

Introduction

1.1. Problem Description

The task of high level synthesis spans the continuum from the automatic generation of a design, starting with a purely behavioral specification, down to the compilation of a completely specified structural design consisting of a set of components from a given library and their interconnections. In the first case (automatic generation), the behavior is specified as a set of assignment statements to variables, possibly with timing constraints for input-output pairs. There is no binding of operations to time or to functional units, no binding of variables to storage elements, and the description does not have any connectivity specified between storage and functional units. At the other extreme, compilation of structure consists of mapping generic components (or components from one library) to components derived from another library. The main objective is optimization of that mapping to satisfy technology constraints such as time, area, power, testability, etc.

The traditional view of behavioral synthesis ([GrKP85] [Thom86] [McPC88] etc.) assumes that the synthesis system automatically generates the structural design from a user specified abstract behavior. Such systems do not permit the user to interact in the design synthesis and evaluation loop. The major drawback with this approach is that the user cannot impose structural constraints (in the form of an initial design structure), or provide design hints (in the form of behavioral operators and variables *bound* to structural components and connections). The need for such user input is evidenced by the fact that

research in behavioral synthesis algorithms is still in its infancy. Existing algorithms for synthesis tasks like state allocation, component binding, etc. are either limited to certain narrow application areas (e.g. Digital Signal Processing applications), or use restrictive and simple models which fail to generate realistic designs (e.g. unit-delay, unit-cost models). Since the user cannot directly interact with the design process, it becomes difficult to incorporate the designer's knowledge and expertise for guiding the synthesis tasks.

An attempt to rectify this drawback is described in [ThBR87], where "links" are maintained between the abstract behavioral entities (variables, operators) and the resultant structural design (state, component, connection). These links are a useful representation for performing multi-level simulation, enabling behavioral verification of the synthesized structure. But on closer examination, this behavior-to-structure linking does not really help the designer explore different design alternatives. If the synthesized design does not meet the constraints, the user is forced to re-synthesize the design automatically from the abstract behavior by changing some high level constraint.

For instance, knowing that variable "A" in some statement of the behavior is bound to register R1 in state 2 of the synthesized design doesn't really help the user decide on how to improve the design; it merely serves as a debugging aid to verify the correctness of the synthesis algorithms that generated that particular design. Instead, what is really needed is a mechanism that permits the user to selectively specify the binding of certain behavioral variables and operators to specific structural components and connections. The user is then able to directly influence synthesis of the structural design to meet the desired constraints.

Several requirements emerge from the previous discussion:

(1) *partial design specification*: the user should be able to specify a partially designed structure as an initial constraint; the synthesis tools should then be able to complete the rest of the design.

(2) *user-bindings*: the user should be able to selectively bind behavioral operations to particular states, behavioral operations to components, and behavioral variables to storage components (e.g. registers) or connections (e.g. buses, wires).

(3) *modification of compiled designs*: the user should be able to modify a structural design during or after synthesis [1].

(4) *modification of synthesis tools*: a consistent and readable intermediate format is required to enable the addition of new tools and the modification of existing synthesis tools; the format must allow description of the complete design with links to the behavior *at each stage of the design process.*

In this report, we describe a new intermediate representation using annotated textual state tables which supports the above requirements. We will show how this representation can be used to describe the design at each level of abstraction in the synthesis process. It facilitates easy translation to and from the internal data structures of synthesis algorithms, thereby allowing interchangeability (and upgrading) of synthesis tools. It also serves as a useful linking mechanism between the behavior and the structure. Furthermore, users can interact with the representation at each of the intermediate levels, allowing for user modification of the partial designs. The state-table based format is flexible yet simple with an overall consistency throughout the levels of abstraction; designers will find this to be a

[1] Modification of compiled design is described in more detail in Section 5, User Interaction Scenarios.

convenient interface mechanism for interacting with a behavioral synthesis system. Furthermore, its model is general, since it can express hierarchy, concurrency, timing relationships and asynchronous behavior in a single, unifying intermediate form.

CHAPTER 2.

Behavioral Synthesis Framework

2.1. Synthesis Tasks

Behavioral synthesis is the process of mapping an abstract behavioral specification to a structural design that satisfies the behavior within some design constraints. The behavior is normally described by a sequence of language variables and operators, while the structural design is an interconnection of functional units and storage elements operating on a state-by-state basis. The functional units, storage and connection elements are normally drawn (or *allocated*) from a given library. There are several behavior-to-structure mappings that comprise this synthesis task; these mappings are called *bindings*. Some of the important synthesis tasks are mentioned below.

Resource allocation is the task of determining the type and number of functional units, storage elements and connections to be used in the ensuing design. The allocated resources must satisfy the designer's high level constraints.

State binding is the temporal assignment of operation sequences in the behavior to states of the structural design. A state, in this context, may be delimited by a clock or signal event. A signal event, associated mostly with asynchronous circuit behavior, refers to a change in a signal value.

Unit binding is the task of assigning functional and storage units to particular behavioral operators and variables.

Connection binding is the task of providing connections between structural components to effect the data transfers specified in the behavior.

Control synthesis is the task of generating control logic which sequences the design through the final states of the design, and which produces control signals for performing operations within each state.

Each of these tasks can be followed by an optimization phase, where, for instance, unit merging follows unit binding, or connection merging follows connection binding. These allocation, binding and optimization tasks are closely inter-related; there is no optimal ordering of these tasks and current research in this area is still attempting to understand their interaction. This underscores the need for a standard intermediate format that captures the complete design at each of these levels.

2.2. A Typical Synthesis Environment

We will use the environment shown in Figure 1 as a representative synthesis framework to show the utility of the intermediate form. The figure is organized into three columns: the synthesis tasks on the left, the user interface on the right, and the intermediate representation in the middle. The intermediate representation is composed of four basic components: the state table, the unit list, the connections list, and the symbol list.

The user typically specifies the behavior of the design in a behavioral specification language like VHDL [LiGa88] or EXEL [DuGa89]. The language compiler parses the input into a data structure which is captured in the first level of the intermediate form, by creating the the symbol list and a hierarchical operations table. In addition, if the user has

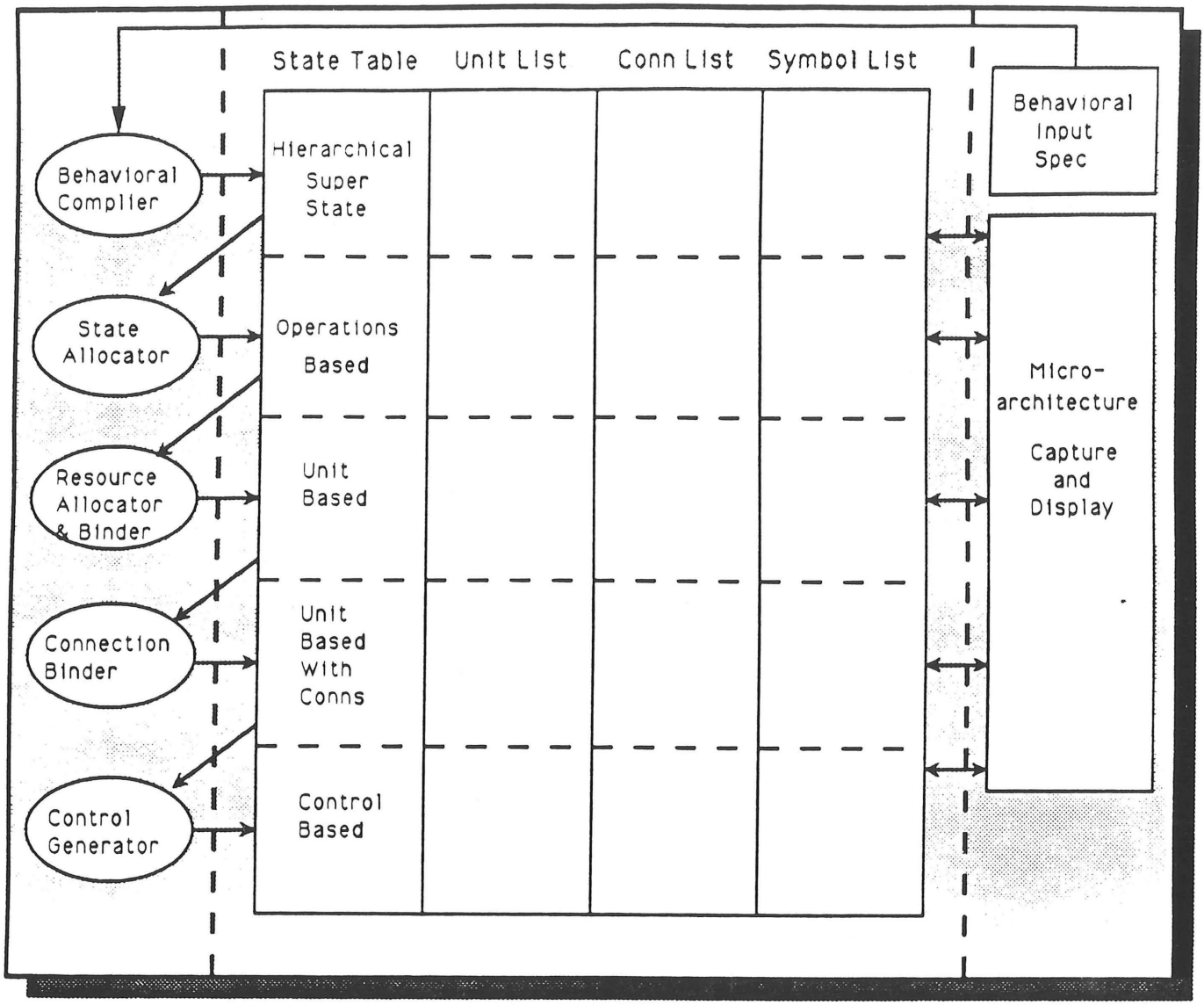

Figure 1. A Typical Behavioral Synthesis Environment

specified some structure along with the behavior, this structure is captured in the unit and connectivity lists. Since the designer may not know the optimal state encoding while describing the behavior, the input is naturally described using sequences of groups of operations. This description forms a multi-level hierarchy, where an operations group at one

level of hierarchy can be defined by a sequence of operations groups at a lower level. The lowest level of hierarchy consists of sequences of operations. Each operations group is much like a basic-block in a standard programming language; it may span several states depending upon the state encoding scheme used. We call an operations group a *super-state*, and we refer to this form as the *hierarchical super-state table.* This table describes the operations performed in each super-state, and the sequencing between super-states.

The lowest level of hierarchy in the hierarchical super-state table is composed of sequences of operations arbitrarily allotted to states. These state assignments may be based on a behavioral view of the design. In the next level of design, we use a state allocator to "slice" the states into hardware states of the design. Operation sequences are assigned to specific states of the final design. This task is also called *state scheduling* in the literature [PaKn87] [PaGa87]. The *op-based state table* is generated by this synthesis task. This table uses conditional triplets to capture the behavior of the design on a state-by-state basis. Each triplet describes the condition tested, the operations performed and the event-controlled next state to be executed. The next state is entered only if the controlling event occurs. In synchronous designs the controlling event is the clock. Note that the previous hierarchy of super-states defined in the hierarchical super-state table is also represented in the ops-based state table. The only difference between the two is the operations-to-state assignments at the lowest level of hierarchy. At this point in the synthesis framework, the temporal ordering of operations has been fixed, but we have not specified how exactly these operations are to be performed in hardware; this is determined by the tasks of resource allocation and binding.

Resource allocation determines the type and number of structural components needed to implement the structural design. These components are typically drawn from a generic library [Dutt88], which contains information about each type of component. Since buses are also treated as components, this task updates both the unit list and the connection list.

Resource binding assigns *specific* instances of functional and storage components to abstract operations and variables in the op-based state table. At this point, the design is stored in the *unit-based state table.* This table uses triplets to describe the structural operation of the design on a state-by-state basis. Each triplet describes the unit generating the conditional, the units performing the conditional operations, and the event-controlled next state to be executed. The *operations* in the unit-based state table only specify which components are to be used as inputs for the operation; they do not specify the connection paths for these inputs.

The task of connection binding adds these connection paths to the unit-based state table to create a *unit-based state table with connections.* This table describes the complete structure of the synthesized data path, but lacks the control signals for the components.

Finally, the task of control generation creates control lines for every functional or storage unit that needs to be controlled. The *control based state table* captures this functionality with triplets that describe the control lines conditionally activated in each state, and the subsequent event-controlled next state.

At each level of the synthesis process, the appropriate synthesis task can be performed automatically (by a set of algorithms and rules), or can be performed manually by the user through the user interface. The user interface displays the units and connections in the

form of a schematic, and displays the state tables visually. This permits the user to comprehend the complete behavior and structure of the design at each level.

Note that we have introduced this particular framework progression solely for the purpose of illustrating the use of the intermediate form. The tasks of state binding, resource allocation, unit binding, connection binding and control generation can be performed in different combinations; the annotated state tables described in this report can still be used as the intermediate exchange format between the various synthesis tasks, regardless of their order of invocation.

CHAPTER 3.

BIF : The Behavioral Intermediate Format

3.1. General Description

The Behavioral Intermediate Format (BIF) makes use of modified state tables annotated with a list of symbols, units, and connections to describe a design at each level of abstraction in the synthesis process. However, since the design spans several levels of abstraction (as illustrated in Figure 1) , we maintain a slightly different format for each abstraction level in the design process.

The state tables and associated information lists are progressively updated as the synthesis process proceeds. At each level of abstraction the user can, either directly or through the use of tools, modify the state table and/or any of the information lists.

In this section, we will describe some of the features of BIF that make it useful for a large class of designs. Chapter 4 will illustrate the use of BIF with two annotated examples.

3.2. Hierarchy

Hierarchy is a useful method for handling complexity in describing, designing, and managing large designs. BIF supports the concept of hierarchy in the state table format by allowing each state in the hierarchy to be decomposed into a number of sub-states. *Statecharts* [DrHa89] provide a method for representing this concept visually. Figure 2 shows how a state table is divided into levels of hierarchy to facilitate modular development and

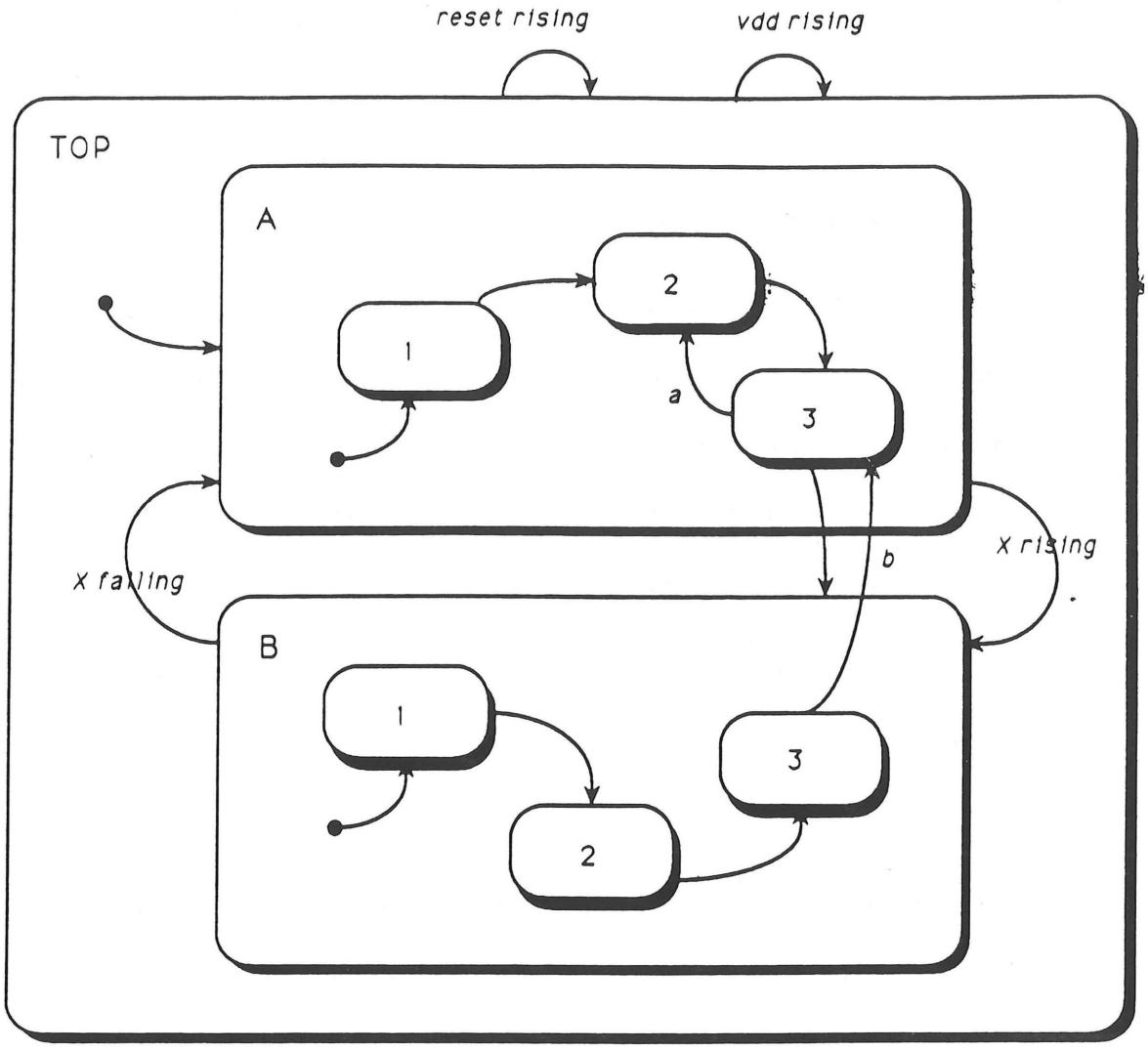

Figure 2. A Graphical Representation of Hierarchy

user comprehension of the overall design.

The highest level of hierarchy in Figure 2 is called *TOP* and the events *reset* and *vdd* cause a reentry into the initial state of *TOP*. *TOP* is composed of two states named *A* and *B*. *A* is the initial entry point for *TOP*; hence the events *reset* rising or *vdd* rising force *TOP* to enter state *A*. The event *X* rising forces a transition from state *A* to state *B*. When event *X* falls, control is again returned to *A*.

A is itself composed of 3 states, the initial state being labeled 1. On reaching state 3 in *A* condition **a** returns the design to state 2 in *A*, while condition **b** forces an exit from *A*, and enters *B* at its initial state, 1. State *B* operates in a similar fashion.

Figure 3 shows how BIF would represent the control portion of the design of Figure 2. The hierarchy is easily visualized by a multi-way tree of state tables with each state entry having at most one state table as an immediate descendant. Each state in the table at the highest level of hierarchy (*TOP* in this example) may be a parent node to a state table at a lower level (*A* and *B* in this example).

Events described in each table apply to all of the states of its descendants. For instance, the first entry in table *TOP* declares that events *reset* and *vdd* will effect a transition to the initial state if we are in *any* state of *TOP*. Since *TOP*'s two states, *A* and *B*, subsume all states in the lower hierarchies, these two events apply to all states at all levels of hierarchies.

3.3. Concurrency

Concurrency is the notion of separate processes running in parallel. BIF supports this notion by allowing multiple processes of a design to operate concurrently at a particular

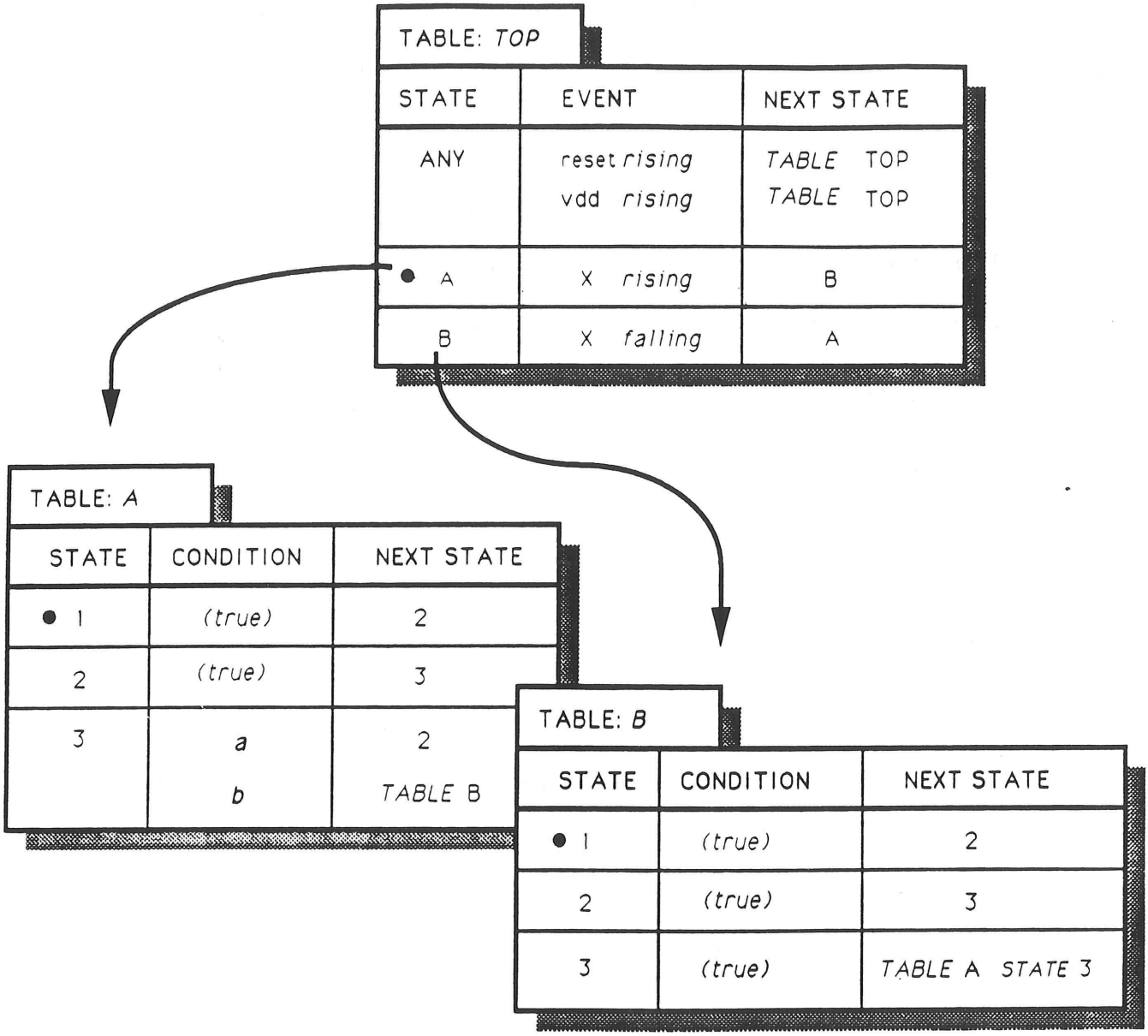

Figure 3. Hierarchy in BIF

level of hierarchy. Each process is represented by a state table. Figure 4 shows a graphical representation of concurrency within a design.

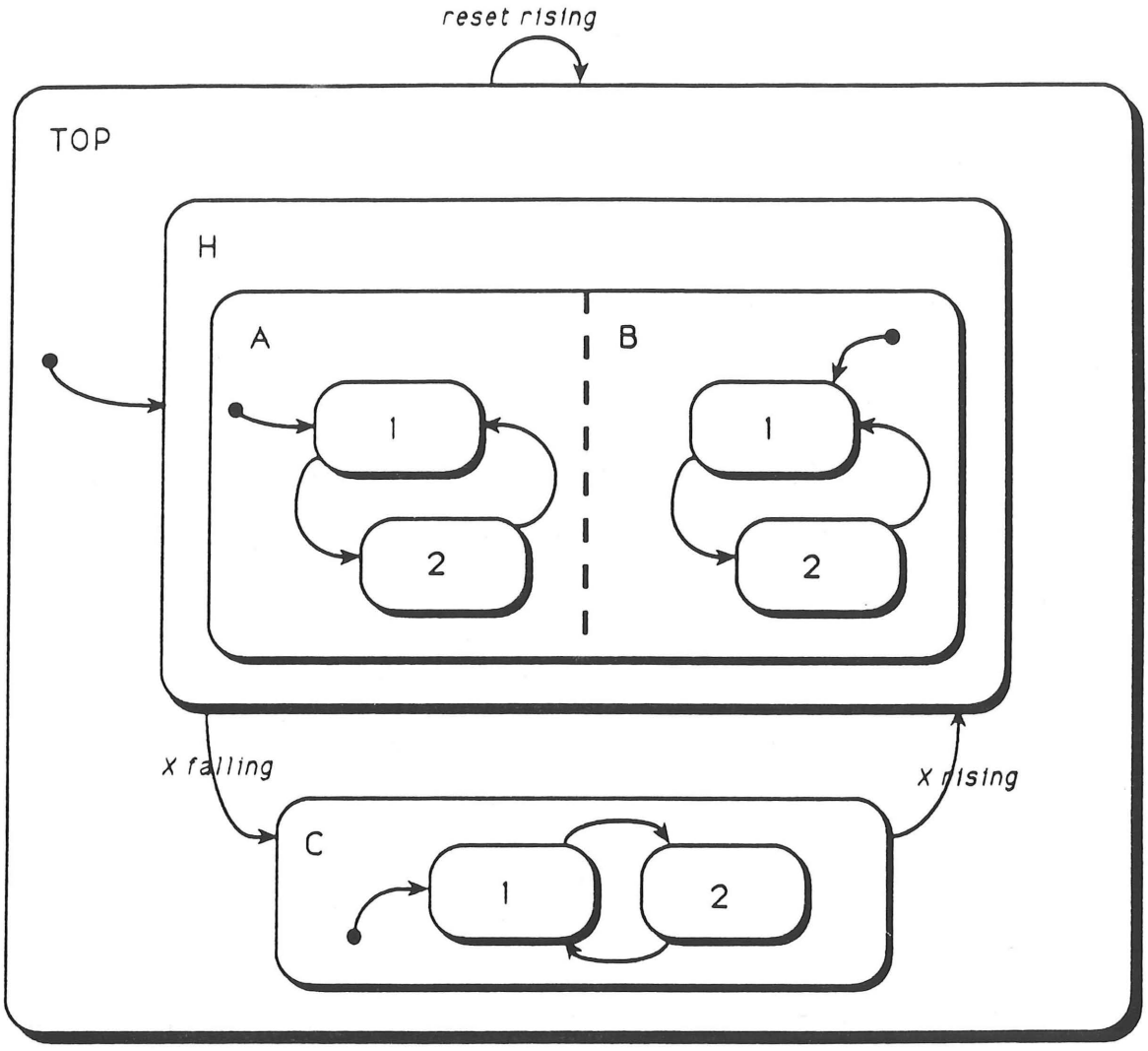

Figure 4. A Graphical Representation of Concurrency

In this example, *TOP* is the highest level of hierarchy, composed of states *H* and *C*. The event *reset* causes a reentry into the initial state *H* of *TOP*, regardless of the current

state. C is entered from H following event X falling. When X rises control is returned to H. H is composed of concurrent substates A and B and entry into H passes control to the two state tables, A and B, simultaneously.

Figure 5 shows how BIF represents the design of Figure 4. Table H indicates that its two states, A and B, run concurrently, and are further defined by the their descendents, tables A and B.

3.4. Asynchronous Behavior

Traditionally, a state table is composed of entries which indicate what actions are performed in each state, the conditions under which those actions are to occur, and the next state to proceed to after completion of the actions. This idea assumes that a single unique event always triggers the transition from one state to the next. This is a valid assumption only if we make the restriction that our designs are to be synchronous and controlled by a single clock. To adapt state tables to represent asynchronous designs and/or multiple clocking schemes we must include a field that specifically describes the event that activates the next state. For the state table format used in BIF we associate each next-state with an event that causes the transition. Taken together, we call this pair the *event-controlled* next state. BIF uses a separate *event* column in the state tables to describe the event condition which activates the next state. The user may also include a specification of a *time-out* in case the event does not occur. A *time-out controlled* next state is specified by labeling the event signal with a duration value and the keyword *timeout*, and including the appropriate next state.

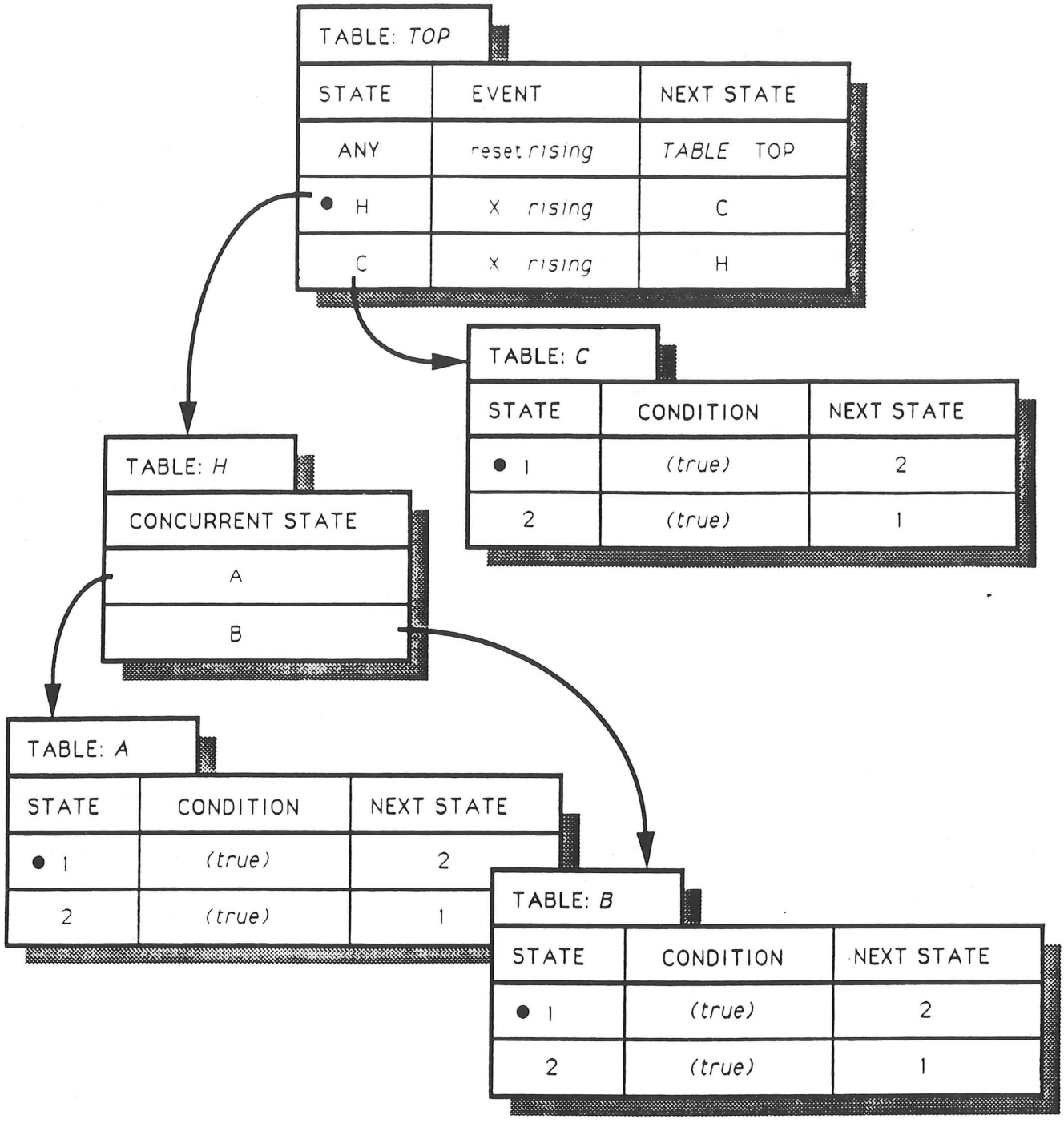

Figure 5. Concurrency in BIF

Chapter 4 will illustrate the use of the event-controlled next state with an asynchronous design example.

Very often, in both synchronous and asynchronous designs, certain events apply to every state. A simple machine reset, for example, would require that the design proceed to a prespecified state, regardless of the current state, on the rising edge of the reset event. To represent this in BIF, we provide a "wild-card" state specifier which matches all states in the current level of hierarchy. The event-controlled next state entry in that state applies to all states.

In synchronous designs, the system clock is the default event that sequences the design through different states of the machine. We therefore omit the event field in purely synchronous design descriptions. However, when a design exhibits a mixture of synchronous and asynchronous behavior, the clock can be used as an explicit event to indicate states that are entered synchronously.

3.5. Interface Specification

If a particular design receives events and control signals from the outside world, it may be necessary to describe these signals so that synthesis tools can verify the correctness of the design or make appropriate modifications. This information can also involve descriptions of interface protocols to allow some degree of interface synthesis.

In a situation where a signal, originating from outside a target design's representation, interacts with that design, BIF must be able to describe certain attributes of the signal. Information about the *duration* of the signal may be important if it is known that the signal shows *pulse* characteristics. If the signal is a clock, then information about the *frequency* and *rise/fall* time is important. If a group of signals interact with the design in a predefined manner, a *protocol* can be specified. For instance:

X: ... (*INTERFACE*: *active high*, *pulse*, *duration*: $30ns$) ...;

Y: ... (*INTERFACE*: *active low*, *edge-triggered handshake*) ...;

CLOCK1: ... (*INTERFACE*: *frequency*: $60ns$, *rise*: $10ns$, *fall*: $10ns$) ...;

Here, the keyword *INTERFACE*, in the symbols list, introduces a number of attributes for each signal, **X**, **Y**, and **CLOCK1** . For **X**, *active high* means that the signal is enabled high, *pulse* indicates that the signal remains high for an amount of time defined by its originator, and *duration* gives the amount of time the signal remains high. For **Y**, *active low* means that the signal is enabled low, and *level-triggered handshake* indicates that the sender of **Y** expects an acknowledge signal before it enables **Y** again. **CLOCK1**'s interface description shows its frequency and rise and fall time duration.

Interface specifications make it possible for synthesis tools to modify the representation to ensure correct design. **X**, in the previous example, is high for a duration of $30ns$. If a particular synthesis tool determines that **X**'s pulse may be missed by the target design, it may be necessary to insert a latch between **X** and our target design to ensure that the signal is caught. In **Y**'s interface description, *edge-triggered handshake* is specified, which contains the implicit notion of an acknowledging signal from the target design which follows the edge transitions of **Y**. Again, a particular synthesis tool will identify this case and ensure that an acknowledging signal exists and interacts appropriately with **Y**.

3.6. Control/Data Implementation

BIF allows the user (or a synthesis system) to specify how conditionals are to be implemented in the resulting design. Conditionals that are specified in the condition field of a

BIF state table are assumed to be implemented in control, while conditionals (such as IF statements) specified in the operation field of a BIF statement are implemented in the data path. This notion is similar to EXEL's interpretation of conditional evaluation [DuGa89]. This gives the user (or a synthesis system) direct control over how to implement the conditionals, and also provides a mechanism for exercising control/data tradeoffs.

CHAPTER 4.

Three Illustrative Examples

4.1. Quotient Accumulator Example

In this section, we will use a simple example to illustrate the basic format of the annotated state tables at each level in the design process. A brief tutorial of the intermediate form is described in Appendix I, while the detailed syntax is given in Appendix II.

Figure 6 shows a flowchart for a design that performs the function:

$$\sum_{i=1}^{IREG} (LIMIT \textbf{ div } i)$$

The design accumulates the sum of all quotients for an externally specified value (LIMIT), with respect to every number equal to and below the value set in an internal register IREG. In this specification, we assume that the user has already specified the states of the machine. The initial structure specified by the user is shown in Figure 7. LPORT is the port through which the external limit is specified, while TPORT and DPORT are used to output the accumulated sum and a flag signifying the end of the task. Internally, the user has specified the several components and connections: registers IREG, CREG, DONE and LIMIT; the counter TICK; and the connections between ports LPORT, TPORT, DPORT and LIMIT, TICK and DONE respectively. IREG is set to a pre-determined start value for the algorithm, while CREG functions as a temporary register, and TICK keeps track of the accumulated quotients. LIMIT is loaded with the external value with respect to which the

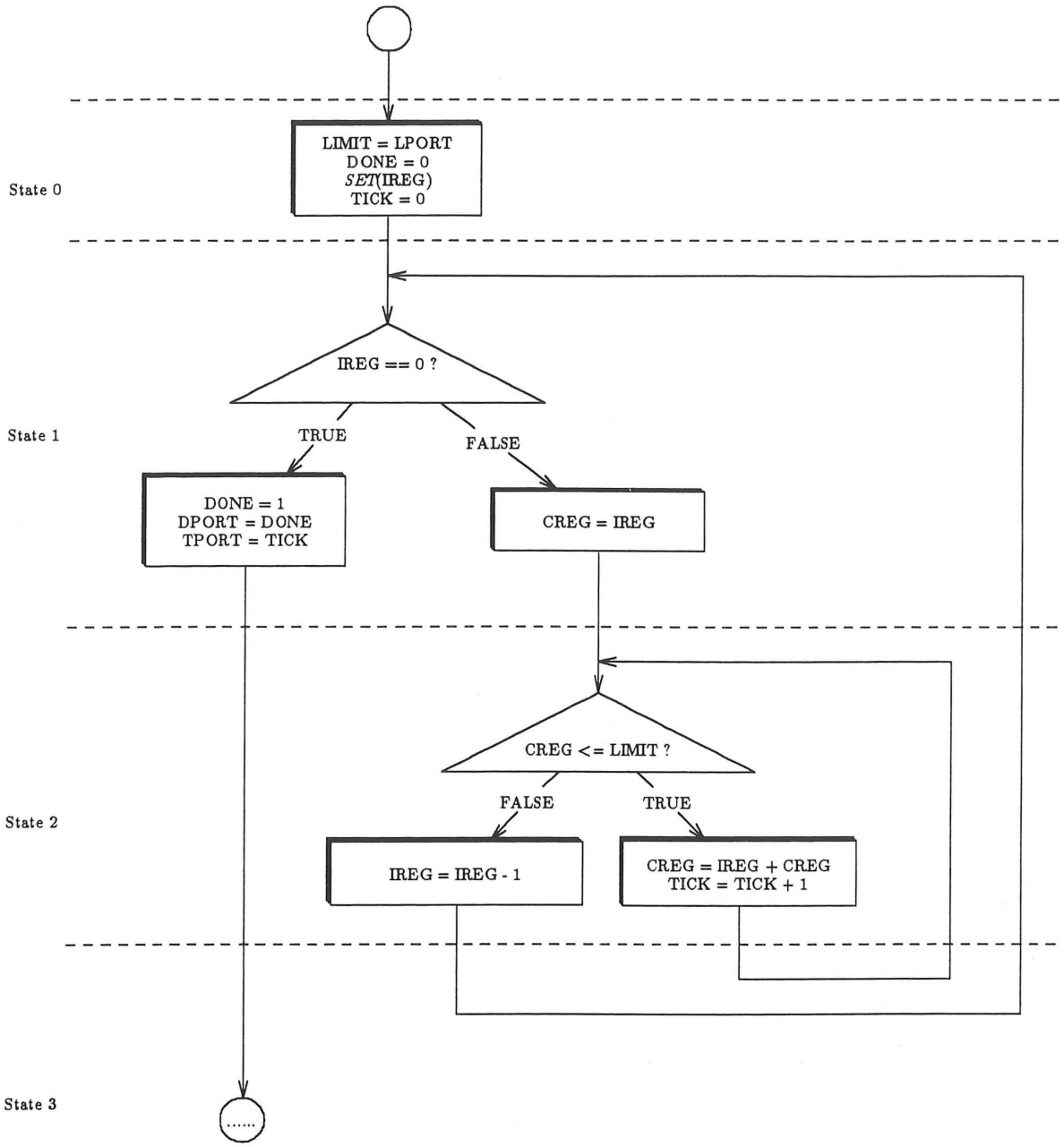

Figure 6. Quotients Accumulator Behavior

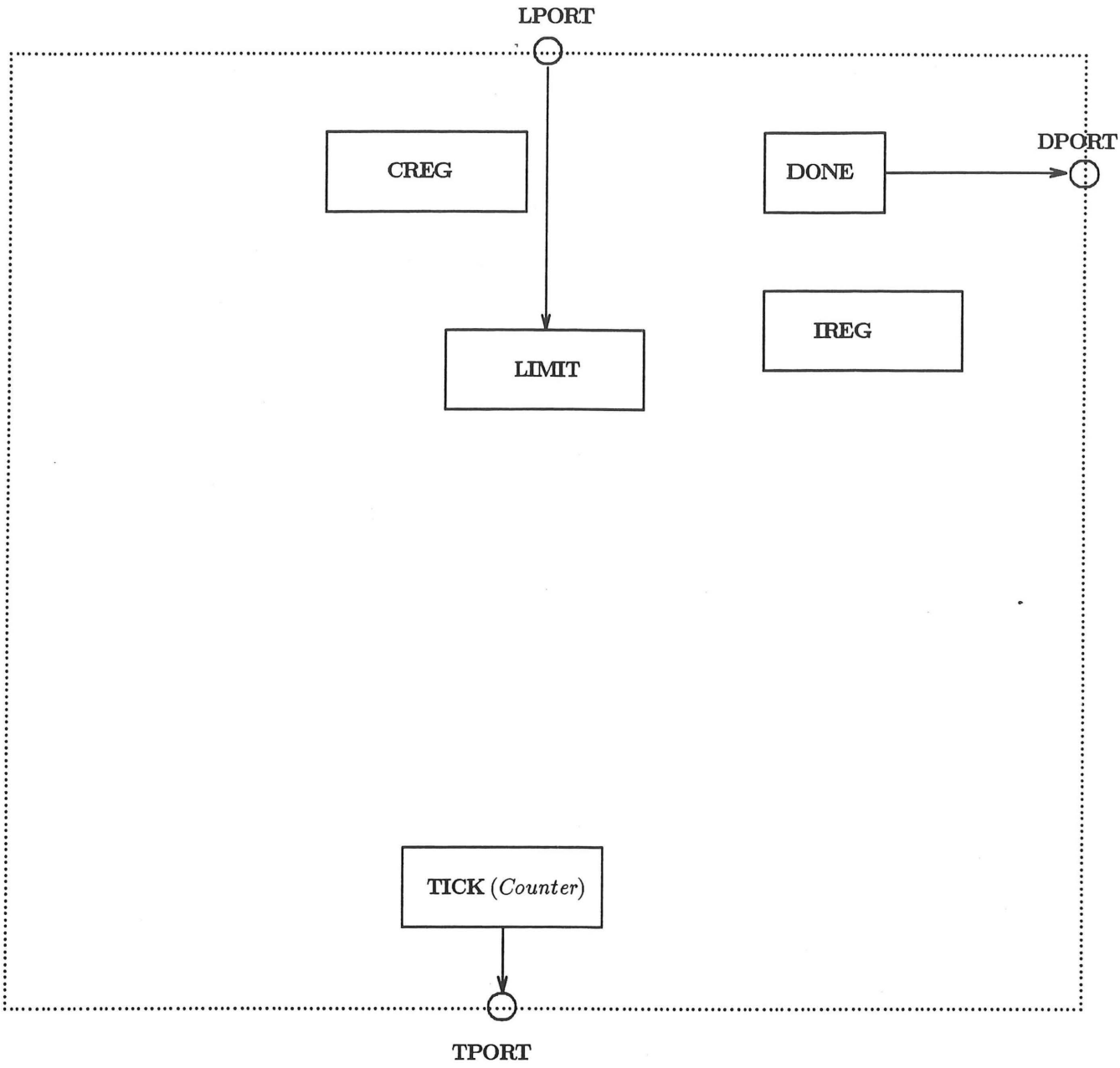

Figure 7. Quotients Accumulator Initial Structure

accumulated quotients is to be computed. DONE indicates the status of the completed task.

In state 0 of the behavior, we load the LIMIT register with the value on LPORT, clear DONE and TICK, and set IREG to the predetermined value.

States 1 and 2 describe a nested loop, where the outer loop decrements IREG by one, and the inner loop computes the quotient of LIMIT with respect to the current value of IREG.

When IREG is equal to 0, the task is completed. DONE is set to 1 and is asserted on DPORT, while the accumulated sum in TICK is sent out on TPORT.

4.1.1. The Operations-Based State Table Format

Since the input behavior already has states assigned to it, we capture initial behavior using the operations-based state table (OBST). The OBST contains triplets for each state, describing the condition tested, conditional operations performed, and the next state information.

Figure 8 shows the operations-based state table for the example shown in Figure 6. In this example, the user has also specified a partial structure consisting of the external ports and a few registers. This structure is stored in the symbol list and the unit list of the OBST. The structure is identical to that of Figure 7, since no additional units or connections have been allocated.

4.1.2. The Unit-Based State Table Without Connections

The task of unit allocation and unit binding assigns additional components (if necessary) and binds operations in the OBST to specific units. The output of this phase is the unit-based state table without connections (UBST). Each triplet in this table describes the condition tested, the unit name and the operation performed by the unit, the list of inputs

Present State	Condition	(Value)	Actions	Next State
0	-	TRUE	LIMIT = LPORT DONE = 0 *SET*(IREG) TICK = 0	1
1	IREG == 0	TRUE	DONE = 1	3
		FALSE	CREG = IREG	2
2	CREG <= LIM	TRUE	CREG = CREG + IREG TICK = TICK + 1	2
		FALSE	IREG = IREG - 1	1
3	....			

Figure 8. Operations-Based State Table

used for the operation, and the next state information. Figure 9 shows the UBST for the design after unit allocation and binding. Figure 10 is a schematic displayed from the unit and connection lists associated with the UBST. Note that at this point in the design, all the components have been allocated to the design by the synthesis system, but no connections have been generated.

Present State	Condition	(Value)	Actions	Next State
0	-	TRUE	LIMIT(REG; Ops: LOAD; Inps: LPORT) DONE(REG; Ops: CLEAR) IREG(REG; Ops: SET) TICK(COUNTER; Ops: CLEAR) NOR(GNOR_GATE; Ops: GNOR; Inps: IREG.OQ)	1
1	NOR.O0 == 1	TRUE	DONE(REG; Ops: SET)	3
		FALSE	CREG(REG; Ops: LOAD; Inps: IREG.OQ)	2
2	CMP1.OLEQ == 1	TRUE	ALU1(ALU; Ops: ADD Inps: CREG.OQ, IREG.OQ) CREG(REG; Ops: LOAD; Inps: ALU1.O0) TICK(COUNTER; Ops: UP) CMP1(CMP; Ops: LEQ; Inps: CREG.OQ, LIMIT.OQ)	2
		FALSE	ALU1(ALU; Ops: DEC Inps: IREG.OQ) IREG(REG; Ops: LOAD; Inps: ALU1.O0) NOR(GNOR_GATE; Ops: GNOR; Inps: IREG.OQ)	1
3	....			

Figure 9. Unit-Based State Table

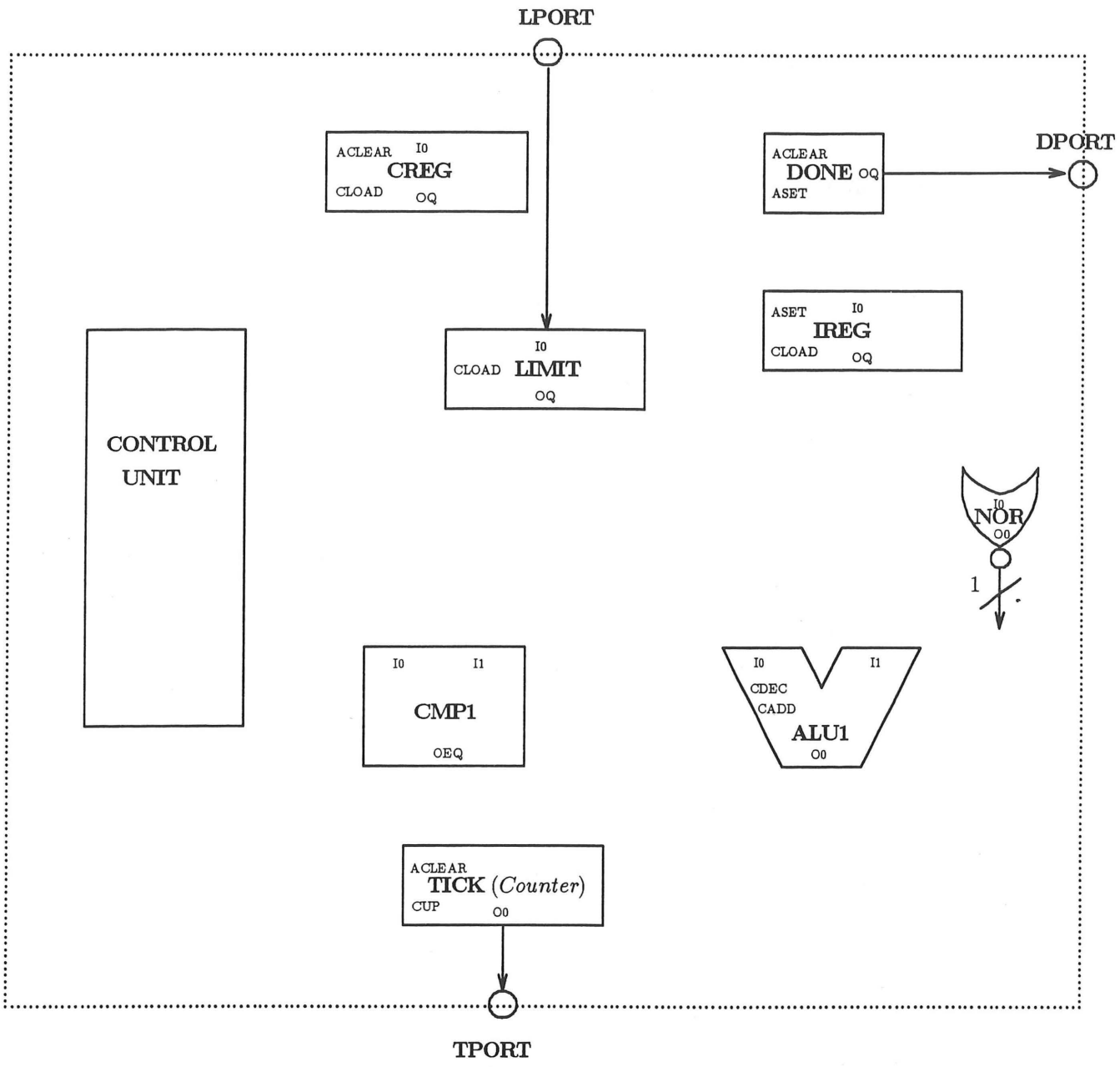

Figure 10. UBST Annotated Structure

4.1.3. The Unit-Based State Table With Connections

The task of connection binding traverses the UBST to determine the connections required to effect data transfers between various components in the design. The unit-based state table with connections (UBCST) is created after connection binding is performed.

The resulting design describes the complete data path excluding control signals for units and registers. For our running example, Figure 11 shows the UBST for the design after unit allocation and binding. Figure 12 shows the complete data path schematic generated from the unit and connection lists associated with the UBCST.

4.1.4. The Control-Based State Table

The control-based state table (CBST) is created in preparation for the task of control compilation. Like the previous state table, each entry is a triplet which describes the condition tested, the control signals asserted on that condition, and the next state information. Figure 13 shows the CBST for the design after control generation, while Figure 14 shows the schematic of this complete design generated from the unit and connection lists. The complete synthesized design is now represented by the CBST annotated with the unit and connection lists.

4.2. Mixed Synchronous and Asynchronous Example

In synchronous designs, the system clock is the default event that sequences the design through different states of the machine. We therefore omit the event field in purely synchronous design descriptions. However, when a design exhibits a mixture of synchronous and asynchronous behavior, the clock can be used as an explicit event to indicate states that are entered synchronously.

We will use a modified version of the quotients accumulator shown in Figure 6 to illustrate the use of an op-based event state table. Figure 15 shows a flowchart describing a

Present State	Condition	(Value)	Actions	Next State
0	-	TRUE	LIMIT(REG; Ops: LOAD; Inps : LPORT) DONE(REG; Ops: CLEAR) IREG(REG; Ops: SET) TICK(COUNTER; Ops: CLEAR) NOR(GNOR_GATE; Ops: GNOR; Inps: IREG.OQ)	1
1	NOR.O0 == 1	TRUE	DONE(REG; Ops: SET)	3
		FALSE	CREG(REG; Ops: LOAD; Inps: MUX2.O0) CMP1(CMP; Ops: LEQ; Inps: CREG.OQ, LIMIT.OQ) MUX2(MUX; Ops: I0; Inps: IREG.OQ)	2
2	CMP1.OLEQ == 1	TRUE	MUX1(MUX; Ops: I0; Inps: CREG.OQ) ALU1(ALU; Ops: ADD Inps: MUX1.O0, IREG.OQ) CREG(REG; Ops: LOAD; Inps: MUX2.O0) TICK(COUNTER; Ops: UP) CMP1(CMP; Ops: LEQ; Inps: CREG.OQ, LIMIT.OQ) MUX2(MUX; Ops: I1; Inps: ALU1.O0)	2
		FALSE	MUX1(MUX; Ops: I1; Inps: IREG.OQ) ALU1(ALU; Ops: DEC; Inps: MUX1.O0) IREG(REG; Ops: LOAD; Inps: ALU1.O0) NOR(GNOR_GATE; Ops: GNOR: Inps: IREG.OQ)	1

Figure 11. Unit-Based State Table With Connections

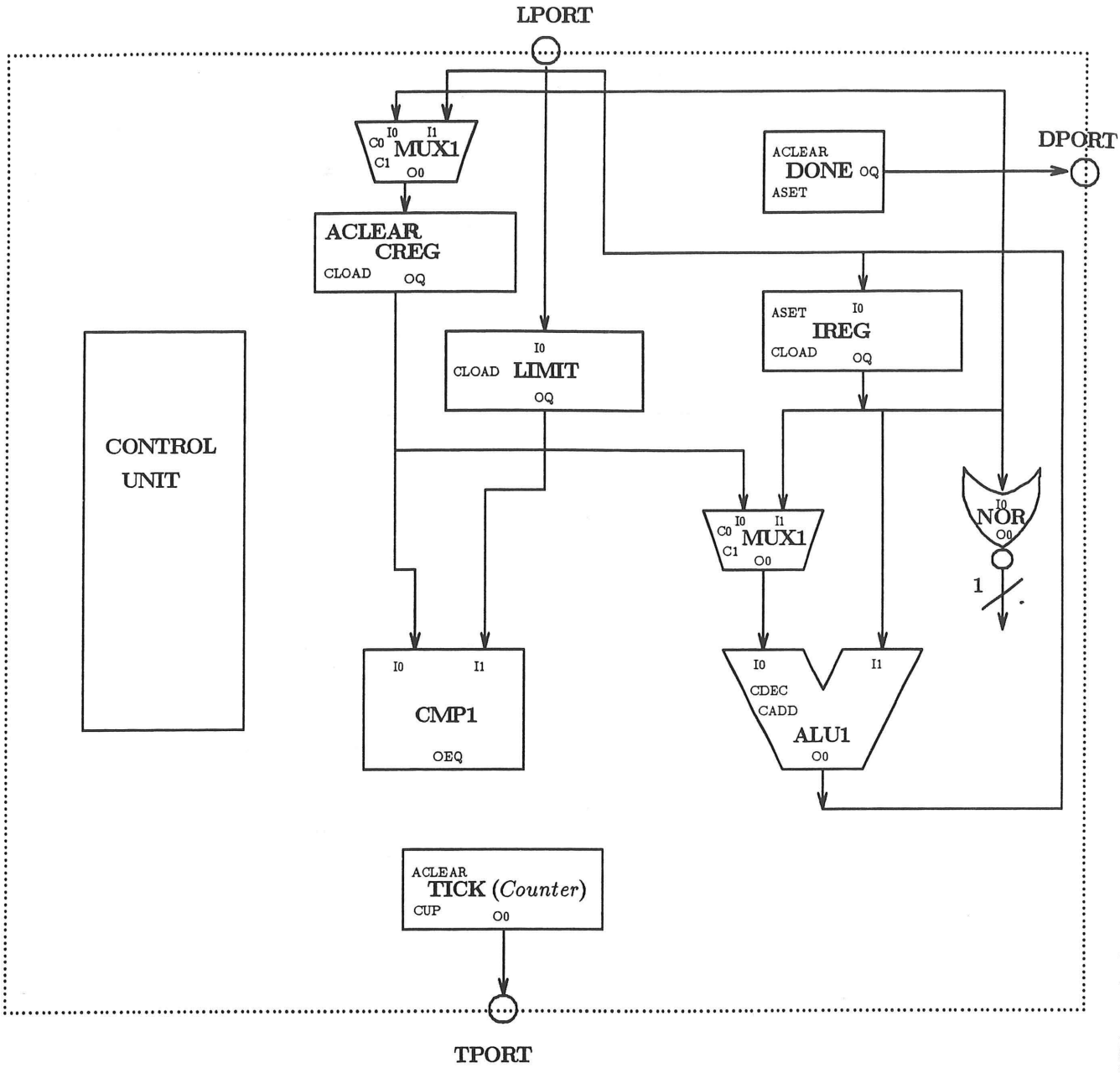

Figure 12. UBCST Annotated Structure

Present State	Condition	(Value)	Actions	Next State
0	-	TRUE	LIMIT.CLOAD = 1 DONE.ACLEAR = 1 IREG.ASET = 1 TICK.ACLEAR = 1	1
1	NOR.O0 == 1	TRUE	DONE.ASET = 1	2
		FALSE	CREG.CLOAD = 1 MUX2.CI0 = 1	3
2	CMP1.OLEQ == 1	TRUE	MUX1.CI0 = 1 ALU1.CADD = 1 CREG.CLOAD = 1 TICK.CUP = 1 MUX2.CI1 = 1	2
		FALSE	MUX1.CI1 = 1 ALU1.CDEC = 1 IREG.CLOAD = 1	1
3	....			

Figure 13. Control-Based State Table

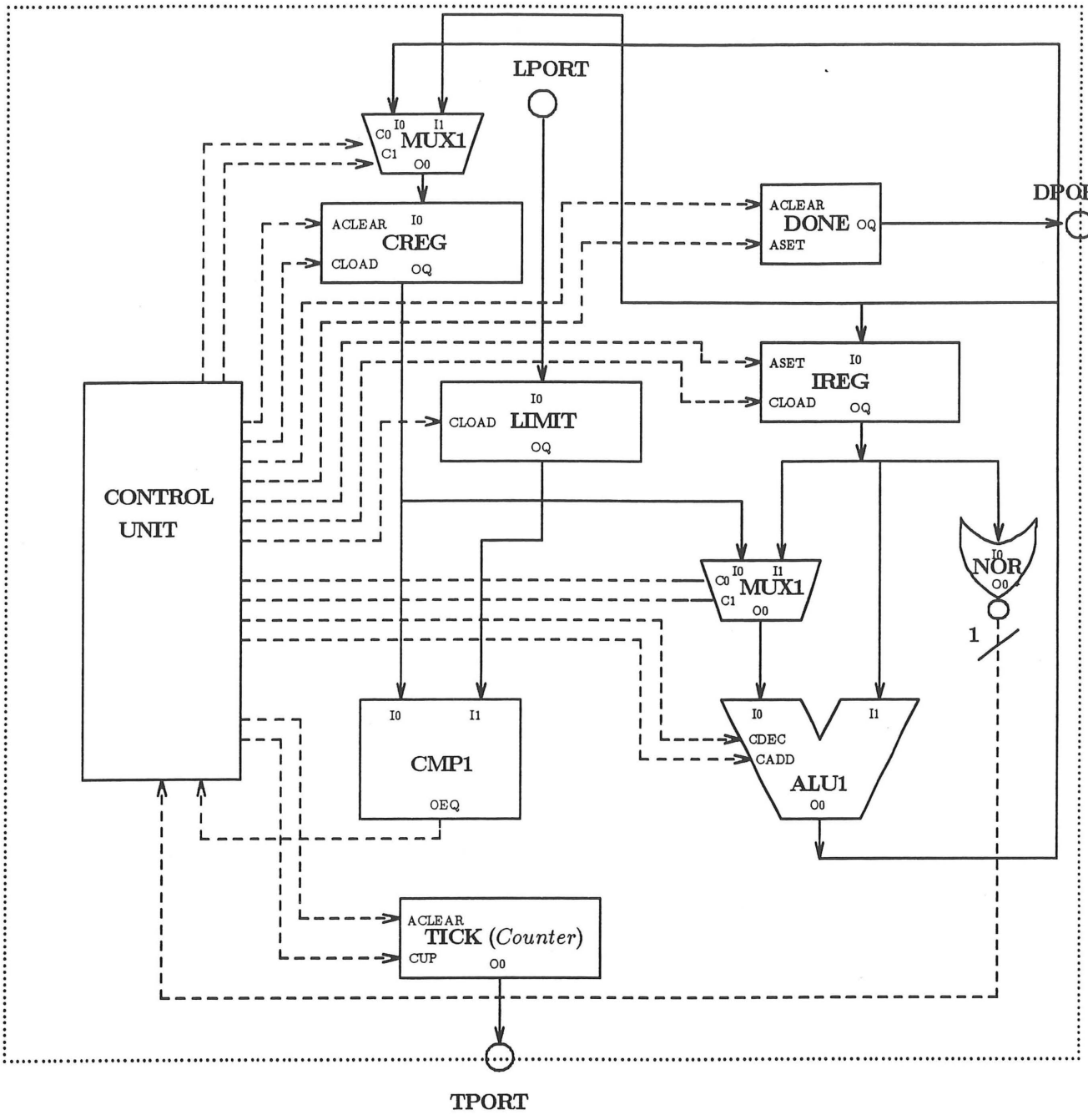

Figure 14. CBST Annotated Structure

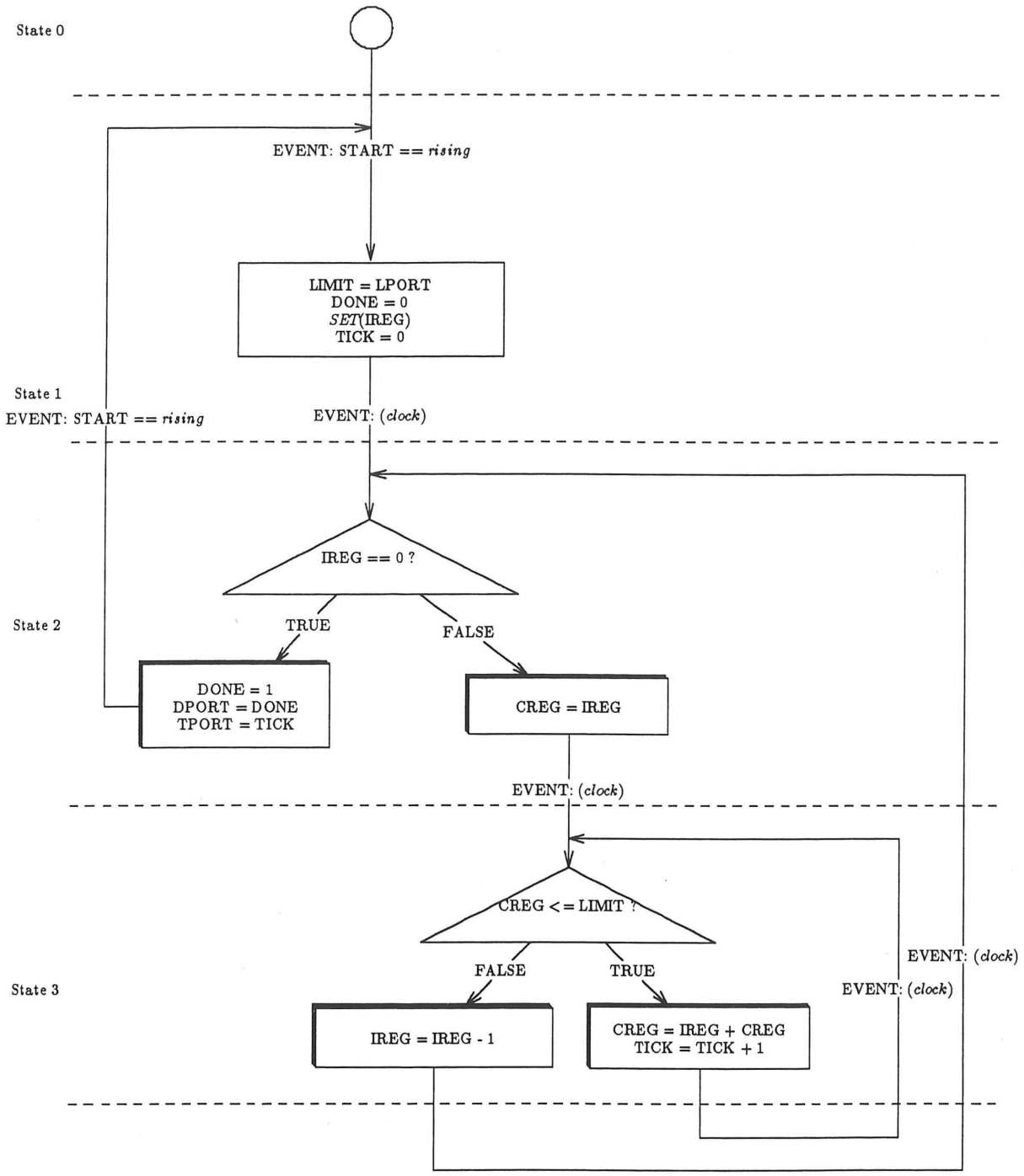

Figure 15. Quotients Accumulator with External Event

similar quotients accumulator which begins operation only when the signal on the port

START rises. State 0 of the design is entered when this event occurs. Subsequently, states 1 and 2 are synchronous with respect to the clock and therefore use the system clock as the default event.

Figure 16 shows the operations-based event state table for this new quotients accumulator behavior. The table has an extra column which describes the event triggering entry into the next event-state. State 0 in this table is entered only on the event *START*

Present State	Condition	(Value)	Actions	Next State	Next State Event
0	-	TRUE	-	1	START == RISING
1	-	TRUE	LIMIT = LPORT DONE = 0 *SET*(IREG) TICK = 0	2	(*clock*)
2	IREG == 0	TRUE	TPORT = TICK DONE = 1 DPORT = DONE	1	START == RISING
		FALSE	CREG = IREG	3	(*clock*)
3	CREG <= LIM	TRUE	CREG = CREG + IREG TICK = TICK + 1	3	(*clock*)
		FALSE	IREG = IREG - 1	2	(*clock*)

Figure 16. Operations-Based Event State Table

RISING, while states 1 and 2 have the clock as the default events.

4.3. Asynchronous Bus Interface Example

Figure 17 shows a schematic of the bus interface section of a typical memory board.

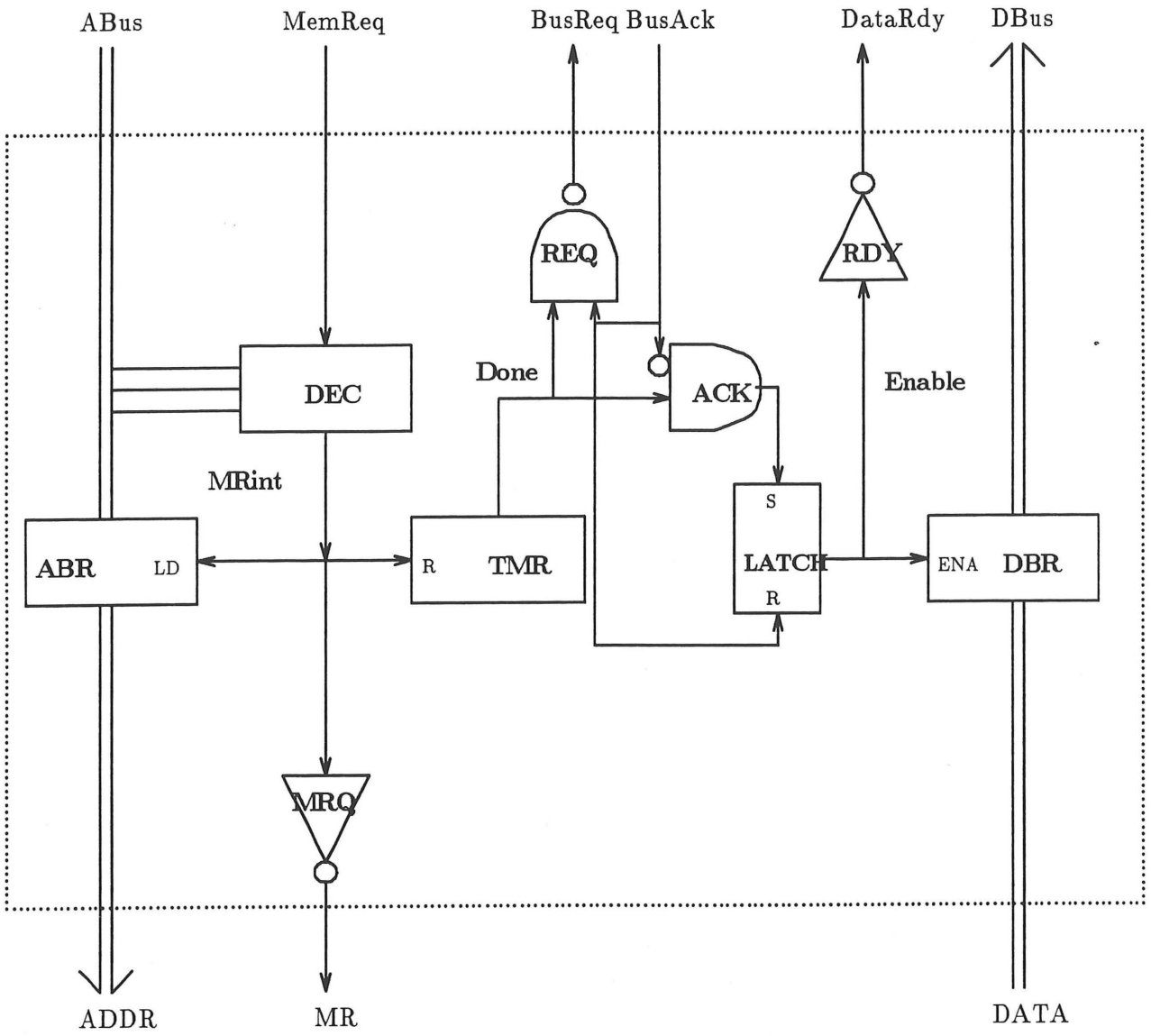

Figure 17. Schematic diagram of the Bus Interface

The timing diagram is show in Figure 18 We will first describe the bus protocol and then show how we can represent this asynchronous design in BIF.

The bus interface has three ports that connect to a ROM array: **ADDR**, which drives the address inputs of the ROMS; **MR**, connected to the chip select lines; and **DATA**, which receives the ROM output data. The rest of the ports connect to the external bus. **ABus** is connected to the address lines of the external bus, **DBus** is connected to the data lines, and the remaining signals participate in the bus handshaking logic. All of the bus handshaking

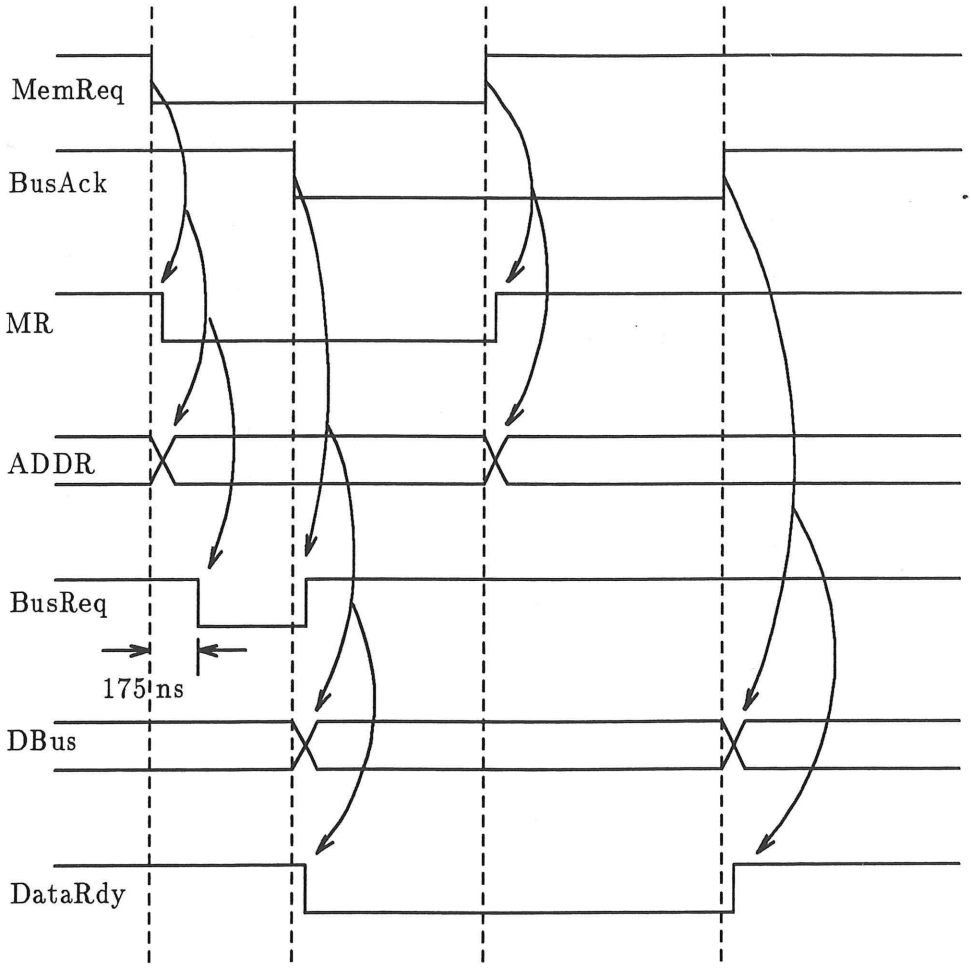

Figure 18. Timing diagram of the Bus Interface

signals are active low.

The memory access cycle begins when **MemReq** goes low and bits 16 to 18 of the address on **ABus** match the board number. This combination of events causes decode **DEC** to drive the on-board signal **MRint** high. **MRint** drives the load input of the address latch **ABR**, the trigger input of the timing element **TMR**, and the buffer **MRQ**.

MRQ drives the chip select inputs of the ROMS. The **TMR** output **Done** goes high 175 ns after the rising edge of **MRint** and falls again as soon as **MRint** drops. Thus the request for the bus is delayed until 175 ns after the **ROM** address lines are set up.

Done holds **BusReq** low until **BusAck** is received from the bus master. The bus signal **BusAck** is inverted and then anded with **Done** to set **LATCH**. **LATCH**'s output drives **RDY**, and at the same time enables the tristate outputs of **DBR**. The data inputs of **DBR** are driven by the ROM data outputs. The **DBR** outputs will remain enabled until **BusAck** rises again, resetting **LATCH**.

Figure 19 shows the operations-based state table for the bus interface. Since the design is completely asynchronous all states are exited on either external event **MemReq** and **BusAck**. There is no clock.

State 1 shows the conditional test for the board id (*BOARD_ID* is a constant defined in the symbols list). The condition is only tested if **MemReq** falls. If the condition is *false* then the design waits in that state until **MemReq** falls again. If the board id matches the address bits then the bus interface performs as described above.

In state 1, the action of setting **BusReq** low is described in terms of delay with respect to the event **MemReq** *falling*. This captures the requirement that the signal **BusReq** be

Present State	Condition	(Value)	Actions	Next State	Next State Event
0	-	TRUE	DBus = 'X' DataRdy = 1 BusReq = 1	1	MemReq = FALLING
1	ABus{16..18} == BOARD_ID	TRUE	MR = 0 Addr = ABus BusReq|(delay 175ns after MemReq FALLING) = 0	2	BusAck = FALLING
		FALSE	-	1	MemReq = FALLING
2	-	TRUE	BusReq = 1 DBus = DATA DataRdy = 0	3	MemReq = RISING
3	-	TRUE	MR = 1 Addr = 'X'	0	BusAck = RISING

Figure 0. Operations-Based State Table for the Bus Interface

delayed 175ns after **MemReq** is enabled.

CHAPTER 5.

User Interaction Scenarios

The annotated state table representation described in this report serves as a standard exchange format for use by various synthesis tasks. This format not only permits modification, upgrading and replacement of the tools for various synthesis tasks, but also provides a "manual override" feature by allowing the user to perform any or all of these synthesis tasks manually. This is a unique advantage of the state table representation over flowgraph-based representations, which only capture the abstract behavior, or netlist-based representations, which capture pure structure. In this chapter we describe several user interaction scenarios that demonstrate the utility of the state table format as a convenient intermediate representation.

5.1. User Specified Structural Constraints

Quite often, the designer may want to specify some initial hardware allocation or some partial design structure as a starting point for the synthesis tasks. By doing this, the user is specifying structural constraints before the task of synthesis begins.

5.1.1. Partial Resource Allocation

If the user partially allocates resources such as a certain number of functional units, storage elements and buses, these resources are stored in the unit list. These pre-specified units constrain the task of resource allocation (see Figure 1).

5.1.2. Partial Design Structure

The user may wish to specify a partial design structure consisting of an interconnection of pre-allocated functional units, storage elements and buses. The components in the partial design are stored in the unit list, while the connections are captured in the connection list. This partial design constrains both the task of resource allocation and connection binding.

5.2. User Specified Bindings

In addition to specifying partial resources and their connections, the user may wish to selectively bind certain behavioral operations and variables to components and connections. This is useful, for instance, when the designer has determined the critical path in the design, and wants to force the binding of fast components along the critical path in the behavior. Some input behavioral languages like EXEL [DuGa89] have special constructs that allow the user to selectively bind resources to abstract variables and operations.

This type of binding is a user specified behavior-to-structure "link" that must be used as a constraint through all the synthesis levels. The pre-allocated components constrain the resource allocation task, while the user-bindings constrain both the resource and connection binding tasks. We represent each such binding explicitly in the state table by annotating the corresponding behavioral variable or operator with the structural component or connection it is bound to.

5.3. Modification of Compiled Designs

An experienced designer who sees the structure generated by an automatic synthesis system can, quite often, identify parts of the synthesized structure that are inefficient, unrealistic or which just seem odd to the experienced eye. The designer may want to correct the design by manually modifying parts of the compiled structural design. This type of user modification is a unique feature supported by BIF; existing behavioral synthesis systems do not permit such modifications.

A typical example would be an automatically synthesized structure where a register A is cleared by loading the value "0" from a constant register [1]. If the register A is loaded through another source, the design also has a mux at the input to register A to switch between the two sources. For this design, the designer would like to modify the generated design manually by replacing loading of the zero register with the activation of the asynchronous clear input on the register. This eliminates the zero register, as well as the mux at the input to register A.

These kinds of changes are handled very cleanly in BIF. Structural changes to the compiled design are updated in the unit list and the connection list. Since there is no guarantee that the design will still function correctly after user-modification, the behavior must be verified on this new design structure by simulation. If the simulation does not satisfy the intended behavior, the complete synthesis process must be restarted from the beginning, using the user specified structural changes as an additional structural constraint.

---

[1] The design model behind most existing synthesis tools cannot handle asynchrony, and hence cannot generate the asynchronous signal required to clear the register.

If the designer modifies the synthesized design by changing the unit list, the synthesis process *must* start from the state binding phase. If only the connections have been modified in the structure, then resynthesis can begin at the connection binding phase.

5.4. State Table Modification

If we allowed complete freedom for the user to perform any or all of the synthesis tasks in the design process, the user could modify the state table in addition to the unit and connection lists.

Since this type of modification can easily cause the behavior of the design to be violated, state table modification must immediately be followed by a simulation to verify that the functionality of the original specification has not changed. Following verification, synthesis tasks can begin from the level where the user change is effected.

For instance, if the user modifies the op-based state table, we first require a verification of the new op-based state table. If the behavior of the new table is unchanged, we use this new state table as a starting point for the ensuing synthesis tasks of resource allocation, resource binding, connection binding and control generation.

CHAPTER 6.

Summary

In this report, we described **BIF**, an intermediate representation format that captures the complete behavior and structure of a design at each level of the behavioral synthesis process. This representation obviates the need to maintain complex behavior-to-structure links from the abstract behavior down to the final structure, by capturing these links *only where necessary*: at each level of the design process.

BIF can express hierarchy, concurrency, timing relationships, asynchronous behavior, and interface protocols in a single, unifying intermediate format.

BIF is an intermediate form which supports several novel design scenarios: specification of partial design structures, user binding of behavioral constructs to structural elements, and user modification of compiled designs. This permits synthesis tools to be interchanged, and also allows the user to manually replace the task of a synthesis tool.

The resulting design paradigm allows an evolution towards completely automatic synthesis, where synthesis tasks that are not fully understood may be performed manually by a designer, while well understood tasks are performed using synthesis algorithms. Synthesis algorithms can therefore be easily incorporated, modified, upgraded or replaced as necessary.

CHAPTER 7.

References

[ChGa89] G. D. Chen and D. D. Gajski, "An Intelligent Component Database for Behavioral Synthesis," Technical Report (in preparation), U.C. Irvine, February, 1989.

[DrHa87] D. Druzinsky and D. Harel, "Using Statecharts for Hardware Description," Proc. ICCAD, Nov. 1987.

[DuGa89] N. D. Dutt and D. D. Gajski, "A Language for Interactive Behavioral Synthesis," *Ninth International Symposium on Computer Hardware Description Languages (CHDL*89), Washington D.C., June, 1989.

[Dutt88] N. D. Dutt, "GENUS: A Generic Component Library for High Level Synthesis," Technical Report 88-22, University of California at Irvine, September, 1988.

[GrKP85] J. Granacki, D. Knapp, A. Parker, "The ADAM Advanced Design Automation System: Overview, Planner and Natural Language Interface," 22*nd Design Automation Conference* (June, 1985).

[LiGa88a] Y.-L. Lin and D. D. Gajski, "LES: A Layout Expert System," *IEEE Trans. on Computer-Aided Design*, Vol. CAD-7, Number 8, Aug. 1988.

[LiGa88b] J. S. Lis and D. D. Gajski, "Synthesis from VHDL," *Proc. ICCD*, Oct. 1988.

[McPC88] M.C. McFarland, A.C. Parker and R. Camposano, "Tutorial on High Level Synthesis," 25*th Design Automation Conference*, July 1988

[PaGa86] B. Pangrle, D. Gajski, "Slicer: A State Synthesizer for Intelligent Silicon Compilation" *Proceedings ICCAD*86 Santa Clara, CA, (Oct, 1986).

[PaKn87] P.G. Paulin and J.P. Knight, "Force Directed Scheduling in Automatic Data Path Synthesis," *Proc.* 24*th IEEE Design Automation Conference*, Miami, FL, June 1987.

[PaPM86] A. C. Parker, J. Pizarro, M. Milnar, "MAHA: A Program for Datapath Synthesis" 23*rd Design Automation Conference* IEEE, Las Vegas, NV (July, 1986).

[THBR87] D. E. Thomas, R. L. Blackburn and J. V. Rajan, "Linking the Behavioral and Structural Domains of Representation for Digital System Design," *IEEE Trans. CAD*, Vol. CAD-6, No. 1, January 1987.

[Thom86] D. E. Thomas, "Automatic Data path Synthesis," *Design Methodologies*, (S.

Goto, editor), Chapter 13, Elsevier Science Publishers, 1986.

[VaGa88] N. Vander Zanden and D. D. Gajski, "MILO: A Microarchitecture and Logic Optimizer," *Proc. 25th D. A. C.*, Anaheim, CA, June 1988.

APPENDIX A.

A Tutorial Introduction to BIF

A.1. General Description

This section is devoted to a syntactical description of the text-based form for state table representation. The row-column approach to state table display (e.g. Figure 4) does not work well for text viewers or editors. It is necessary to provide an alternate format that is easy to enter or edit using a common text editor such as *vi* or *emacs*. This format depicts row entries as successive vertical entries in a text file with corresponding key words representing the various state table constructs.

Each of the four state tables has a common structural format composed of a constant ordering of keywords and delimiters.

- **Table Identifier**

At the beginning of a given table there is a keyword which identifies which of the four state tables it is.

```
OPS_BASED      /* operations-based. */
...            /* table entries */
UNIT_BASED_NC  /* unit-based without connections. */
...            /* table entries */
UNIT_BASED     /* unit-based with connections. */
...            /* table entries */
CONTROL_BASED  /* control-based. */
...            /* table entries */
```

- **State Entries**

Following the table identifier are any number of state entries composed of the keyword **STATE**, a colon (':'), a number identifying the state, a possible unconditional action entry, and a number of triplets describing the conditions, the actions to be performed in that state, and the next state, along with an optional event for the next state. Commas (',') separate all entries following state number, and a semicolon (';') terminates the list of entries. (The ellipses ('...') in all of the following examples indicate entries omitted for readability).

```
STATE: 2       /* state two. */
...,           /* first entry */
...,           /* nth entry  */
```

```
...;            /* last entry */
```

- **Unconditional Action Entry**

The unconditional action entry specifies an action that is to always take place in that state. It is delimited by curly brackets ('{}') and is composed of the keyword **UNCOND_ACTIONS**, a colon, and a list of actions in a format identical to the actions list in a triplet entry (See below).

```
{
  UNCOND_ACTIONS:
  ...
}
```

- **Triplets**

Each triplet is delimited by curly brackets and is composed of three parts: condition, actions, and next state information.

```
{
  COND: ...;          /* condition */
  ACTIONS: ...;       /* actions   */
  NXTSTATE: ...;      /* next state */
}
```

- **Conditions**

Conditions are indicated by the keyword **COND**, a colon, an expression possibly enclosed in parentheses (()), and a semicolon.

```
COND: (...);  /* Expression is represented by ellipsis */
```

- **Expressions**

The expression in the condition is any kind of logical construct that will evaluate to either TRUE (non-zero) or FALSE (zero). (Keywords **true** and **false** are legitimate expressions). Currently, sum-of-products form of boolean equations, comparison to constants, and equality checks against constants are allowed, with variables having slightly different meanings in the operations-based state table form. Operators are too numerous to describe here. See appendix B for a BNF description of expressions and operators.

```
COND: (X OR Y);       /* Operations-based state table */
...
COND: (X OR Y > 4);   /* Operations-based state table */
...
COND: (AU1.SUM > 64);         /* any other state table */
...
COND: (AU1.sum AND CMP1.ogt); /* any other state table */
...
```

- *Else* **Expression**

The *else* special-case is evaluated uniquely among the expressions. If the expression in a condition is the keyword **ELSE** then all conditions in previous triplets up to a previous **ELSE** expression (or the beginning of the state entry) are considered to be relevant to this condition. That is, if all previous conditions fail then the **ELSE** condition evaluates to TRUE. If one or more previous conditions do not fail then the **ELSE** condition evaluates to FALSE. (NOTE: This may require restructuring of the entries by the user to ensure correct condition grouping).

```
{
  COND: (X != 0);
  ...                /* next state and actions */
},
{
  COND: (Y != 0);
  ...                /* next state and actions */
},
{
  COND: (E);         /* TRUE only if X==0 and Y==0 */
  ...                /* next state and actions */
};
```

- **Next State Specification**

The state to proceed to after completing the list of actions is indicated by the keyword **NXTSTATE**, a colon, a state number, an optional event specification and a terminating semicolon.

```
NXTSTATE: 4;      /* Proceed to state 4 after actions */
```

- **Events**

The optional event triggering the next state transition is specified by the keyword **EVENT** followed by a colon and an expression using the **EXEL** [DuGa88] syntax form for asynchronous event timing. For sequential designs where states are activated by the clock the keyword **CLOCK** can be used.

```
NXTSTATE: 4, EVENT: GPORT == rising;

NXTSTATE: 5, EVENT: CLOCK;
```

Note that omitting the event specification for the next state implies a transition on the next clock.

- **Actions List**

Actions to perform in a given state are indicated by the keyword **ACTIONS**, a colon, and a comma separated action list terminated by a semicolon.

```
ACTIONS:
  ...,        /* First action */
  ...,        /* nth action */
  ...;        /* Last action */
```

● **Actions**

The specification format for a single action is differs among the four state table formats. See the specification for each table format under heading *Actions*.

Fields or entries that are not used in a particular state table can be left blank, or the keyword **null** can be used.

C-style commenting (i.e. /* *comment* */) is allowed anywhere in the state table.

A.2. Specific Descriptions of Each State Table Format

A.2.1. Operations-based State Table

The operations-based state table describes actions to be performed in each state in terms of assignment statements. Variable names are not bound to units and instead represent values to be input and output at various stages of the design. ● **Actions**

Actions, listed following the keyword **ACTIONS**, are expressions, variables, and constants combined by logical or arithmetical operators. See the **EXEL** [DuGa88] input language description for a complete description of the expression format.

```
ACTIONS:
  X = Y + 32,     /* Addition          */
  X{0..3} = 0,    /* Selector function */
  Z = X * Y;      /* Multiplication    */
```

Unique to the operations based table is component binding specifications. Optionally immediately following any variable name can be a component name surrounded by curly brackets. This will be interpreted to mean that that variable will be represented by that component in that particular action.

A.2.2. Unit-based State Table With and Without Connections

Both the unit-based state table (UBCST) and the unit-based state table without connections (UBST) have the same syntax. Their differences are conceptual and external only.

- **Actions**

Actions, listed following the keyword **ACTIONS** in the state table, are represented by a unit name followed by a group of attributes delimited by parentheses.

```
ACTIONS:
  CNT2 (...),       /* Counter named CNT2     */
  MUX1 (...),       /* Multiplexor named MUX1 */
  ALU1 (...);       /* ALU named ALU1         */
```

- **Unit Attributes**

Unit attributes describe a unique unit name, the operations performed by the unit, and the inputs to the unit. They are listed within the parentheses as unit name, semicolon, the keyword **OPS**, a colon, a list of operations corresponding to control input names, a semicolon, the keyword **INPS**, a colon, and a list of output pin names from other units.

CNT (counter; **OPS**: inc,dec; **INPS**: ALU1.sum, CTR.I[0])

- **Pin Names**

Pin names are formed by concatenating the actual pin name of the unit with the unique unit name.

COND: (ALU1.sum == 0) /* unit-based state table */

A.2.3. Control-based State Table

The control-based state table describes actions in terms of the values of each unit's control input lines. At each state the pin names of each unit are given with the values they are to assume in that state, either 1 or 0.

```
ACTIONS:
  ADD1.carryin = 0,
  ALU1.czero = 1,
  ALU2.crinhi = 1,
  SHF1.cen = 1;
```

APPENDIX B.

BIF Table Syntax

B.1. Global Table Syntax

File

```
file            :   file_tables ';'
                ;
```

File Tables

```
file_tables     :   file_table | file_tables ';' file_table
                ;
```

File Table

```
file_table      :   table_header '{' table '}' |
                    table_header '{' concurrent_table '}'

table_header    :   TABLE identifier |
                    TABLE identifier ':' STATE identifier
                    OF TABLE identifier
```

B.2. Concurrency Table Syntax

Concurrency Table

```
concurrent_table    :   CONCURRENT '{' concurrent_states '}'
```

Concurrent States

```
concurrent_states   :   concurrent_state | concurrent_states ','
                        concurrent_state
```

Concurrent State

concurrent_state : STATE ':' state

B.3. Operations Based State Table Syntax

State Table

```
table          :       table_ident entries ';'
        ;
```

State Table Identifier

```
table_ident    :       OPS_BASED
        ;
```

State Table Entries

```
entries        :       entry | entries ';' entry
        ;
```

Single State Table Entry

```
entry          :       STATE ':' state triplets |
                       STATE ':' state UC_ACTIONS
                       uncond_actions triplets
        ;
```

Present State

```
state          :       identifier
        ;
```

Unconditional Actions

```
uncond_actions         :       action | uncond_actions ',' action
        ;
```

Triplets

```
triplets:      triplet | triplets ',' triplet
        ;
```

Single Triplet

```
triplet        :       empty |
                       '{'
                       COND ':' condition ';'
                       ACTIONS ':' actions ';'
                       next_state_event  ';'
                       '}'
```

```
        ;

        Next State Information

next_state_event:   NXTSTATE ':' next_state |
                    NXTSTATE ':' state; event
        ;

        Next State

next_state          TABLE identifier ',' STATE identifier
        ;

        Asynchronous Event

event       :       EVENT ':' cond_expr
        ;


        Condition

condition   :       '(' cond_expr ')'
        ;

        Condition Expression

cond_expr   :       variable compare_op expr | pinname compare_op expr |
                    bool_expr
        ;

        Compare Operation Types

compare_op  :       '==' | '!=' | '<' | '>' | '<=' | '>='
        ;

        Actions

actions     :       action | actions ',' action
        ;

        Single Action

action      :       empty | unit_action | ops_action
        ;

        Unit Based Action
```

```
unit_action     :       comp_name '(' comp_type ';' operations ';' inputs ')'
        ;

        Operations Based Action

ops_action      :       simple_assign | cond_assign
        ;

        Simple Assignment

simple_assign   :       variable ':=' expr
        ;

        Conditional Assignment

cond_assign     :       IF cond_expr THEN simple_assign
        ;

        Component Name

comp_name       :       identifier
        ;

        Component Type

comp_type       :       identifier
        ;

        Operations

operations      :       empty | OPS ':' op_list
        ;

        Operations List

op_list         :       op | op_list ',' op
        ;

        Single Operation

op              :       empty | op_type
        ;

        Operation Type

op_type                 :       identifier
        ;
```

Inputs

```
inputs          :       empty | INPS ':' inp_list
        ;
```

Input List

```
inp_list:       input | inp_list ',' input
        ;
```

Single Input

```
input           :       empty | variable | pinname
        ;
```

Expression

```
expr            :       arith_expr | bool_expr | shift_expr
        ;
```

Arithmetic Expression

```
arith_expr      :       '(' arith_expr ')' |
                        arith_expr '+' arith_expr | arith_expr '-' arith_expr |
                        arith_expr '*' arith_expr | arith_expr '/' arith_expr |
                        variable | pinname | dig_seq
        ;
```

Boolean Expression

```
bool_expr       :       lgbl_expr | btbl_expr
        ;
```

Logical Boolean Expression

```
lgbl_expr       :       '(' lgbl_expr ')' |
                        lgbl_expr LAND  gbl_expr | lgbl_expr LOR   gbl_expr |
                        lgbl_expr LNOT  gbl_expr | lgbl_expr LNAND gbl_expr |
                        lgbl_expr LXOR  gbl_expr | lgbl_expr LXNOR gbl_expr |
                        variable | pinname | dig_seq
        ;
```

Bitwise Logical Boolean Expression

```
btbl_expr       :       '(' btbl_expr ')' |
                        btbl_expr '&' btbl_expr | btbl_expr '|' btbl_expr |
                        btbl_expr '^' btbl_expr | btbl_expr '~' btbl_expr |
```

```
                    btbl_expr '~&' btbl_expr | btbl_expr '~|' btbl_expr |
                    btbl_expr '^~' btbl_expr |
                    variable | pinname | dig_seq
        ;
```

Shift Expression

```
shift_expr      :   '(' shift_expr ')' |
                    shift_expr SHL shift_expr | shift_expr SHR shift_expr |
                    shift_expr ROTR shift_expr | shift_expr ROTL shift_expr |
                    variable | pinname | dig_seq
```

Variable

```
variable        :   value_ident
        ;
```

Pin Name

```
pinname         :   comp_name '.' portname
        ;
```

Port Name

```
portname        :   value_ident
        ;
```

Port or Variable Identifier

```
value_ident     :   identifier |
                    identifier '[' dig_seq ']' |
                    identifier '{' dig_seq '..' dig_seq '}' |
                    identifier '{' bound_component '}'
        ;
```

Bound Component

```
bound_component     :   identifier
```

Identifier

Lex Format: [a-zA-Z][a-zA-Z0-9_]*

```
identifier      :   IDENTIFIER
        ;
```

Digit Sequence

Lex Format: [0-9$x$$X$]+

dig_seq : DIGSEQ

B.4. Unit Based State Table Syntax

State Table

```
table          :       table_ident entries ';'
        ;
```

State Table Identifier

```
table_ident    :       UNIT_BASED
        ;
```

State Table Entries

```
entries        :       entry | entries ';' entry
        ;
```

Single State Table Entry

```
entry          :       STATE ':' state triplets | STATE ':' state UC_ACTIONS
                       uncond_actions triplets
        ;
```

Present State

```
state          :       identifier
        ;
```

Unconditional Actions

```
uncond_actions         :       action | uncond_actions ',' action
        ;
```

Triplets

```
triplets:      triplet | triplets ',' triplet
        ;
```

Single Triplet

```
triplet        :       empty |
                       '{'
                       COND ':' condition ';'
                       ACTIONS ':' actions ';'
                       next_state_event  ';'
                       '}'
        ;
```

Next State Information

```
next_state_event    :    NXTSTATE ':' next_state |
                         NXTSTATE ':' state; event
    ;
```

Next State

```
next_state          TABLE identifier ',' STATE identifier
    ;
```

Asynchronous Event

```
event          :    EVENT ':' cond_expr
    ;
```

Condition

```
condition      :    '(' cond_expr ')'
    ;
```

Condition Expression

```
cond_expr      :    pinname compare_op expr | bool_expr
    ;
```

Compare Operation Types

```
compare_op     :    '==' | '!=' | '<' | '>' | '<=' | '>='
    ;
```

Actions

```
actions        :    action | actions ',' action
    ;
```

Single Action

```
action         :    empty | comp_name '(' comp_type ';' operations ';' inputs ')'
    ;
```

Component Name

```
comp_name      :    identifier
    ;
```

Component Type

```
comp_type    :      identifier
        ;
```

Operations

```
operations   :      empty | OPS ':' op_list
        ;
```

Operations List

```
op_list      :      op | op_list ',' op
        ;
```

Single Operation

```
op           :      empty | op_type
        ;
```

Operation Type

```
op_type      :      identifier
        ;
```

Inputs

```
inputs       :      empty | INPS ':' inp_list
        ;
```

Input List

```
inp_list:    input | inp_list ',' input
        ;
```

Single Input

```
input        :      empty | pinname
        ;
```

Expression

```
expr         :      arith_expr | bool_expr | shift_expr
        ;
```

Arithmetic Expression

```
arith_expr   :      '(' arith_expr ')' |
                    arith_expr '+' arith_expr | arith_expr '-' arith_expr |
```

```
                arith_expr '*' arith_expr | arith_expr '/' arith_expr |
                pinname | dig_seq
        ;
```

Boolean Expression

```
bool_expr       :       lgbl_expr | btbl_expr
        ;
```

Logical Boolean Expression

```
lgbl_expr       :       '(' lgbl_expr ')' |
                        lgbl_expr LAND  gbl_expr | lgbl_expr LOR   gbl_expr |
                        lgbl_expr LNOT  gbl_expr | lgbl_expr LNAND gbl_expr |
                        lgbl_expr LXOR  gbl_expr | lgbl_expr LXNOR gbl_expr |
                        pinname | dig_seq
        ;
```

Bitwise Logical Boolean Expression

```
btbl_expr       :       '(' btbl_expr ')' |
                        btbl_expr '&'  btbl_expr | btbl_expr '|'  btbl_expr |
                        btbl_expr '^'  btbl_expr | btbl_expr '~'  btbl_expr |
                        btbl_expr '~&' btbl_expr | btbl_expr '~|' btbl_expr |
                        btbl_expr '^~' btbl_expr |
                        pinname | dig_seq
        ;
```

Shift Expression

```
shift_expr      :       '(' shift_expr ')' |
                        shift_expr SHL shift_expr | shift_expr SHR shift_expr |
                        shift_expr ROTR shift_expr | shift_expr ROTL shift_expr |
                        pinname | dig_seq
```

Pin Name

```
pinname                 :       comp_name '.' value_ident
        ;
```

Port or Variable Identifier

```
value_ident     :       identifier | identifier '[' dig_seq ']' |
                        identifier '{' dig_seq '..' dig_seq '}'
        ;
```

Identifier

Lex Format: [*a-zA-Z*][*a-zA-Z*0-9_]*

identifier : IDENTIFIER
;

Digit Sequence

Lex Format: [0-9*xX*]+

dig_seq : DIGSEQ

B.5. Control Based State Table Syntax

State Table

```
table          :       table_ident entries ';'
        ;
```

State Table Identifier

```
table_ident    :       CONTROL_BASED
        ;
```

State Table Entries

```
entries        :       entry | entries ';' entry
        ;
```

Single State Table Entry

```
entry          :       STATE ':' state triplets | STATE ':' state UC_ACTIONS
                       uncond_actions triplets
        ;
```

Present State

```
state          :       identifier
        ;
```

Unconditional Actions

```
uncond_actions         :       action | uncond_actions ',' action
        ;
```

Triplets

```
triplets:      triplet | triplets ',' triplet
        ;
```

Single Triplet

```
triplet        :       empty |
                       '{'
                       COND ':' condition ';'
                       ACTIONS ':' actions ';'
                       next_state_event  ';'
                       '}'
        ;
```

Next State Information

```
next_state_event    :    NXTSTATE ':' next_state |
                         NXTSTATE ':' state; event
    ;
```

Next State

```
next_state          TABLE identifier ',' STATE identifier
    ;
```

Asynchronous Event

```
event       :       EVENT ':' cond_expr
    ;
```

Condition

```
condition   :       '(' cond_expr ')'
    ;
```

Condition Expression

```
cond_expr   :       pinname compare_op expr | bool_expr
    ;
```

Compare Operation Types

```
compare_op  :       '==' | '!=' | '<' | '>' | '<=' | '>='
    ;
```

Actions

```
actions     :       action | actions ',' action
    ;
```

Single Action

```
action      :       empty | pinname ':=' dig_seq
    ;
```

Expression

```
expr        :       arith_expr | bool_expr | shift_expr
    ;
```

Arithmetic Expression

```
arith_expr      :       '(' arith_expr ')' |
                        arith_expr '+' arith_expr | arith_expr '-' arith_expr |
                        arith_expr '*' arith_expr | arith_expr '/' arith_expr |
                        pinname | dig_seq
        ;
```

Boolean Expression

```
bool_expr       :       lgbl_expr | btbl_expr
        ;
```

Logical Boolean Expression

```
lgbl_expr       :       '(' lgbl_expr ')' |
                        lgbl_expr LAND  gbl_expr | lgbl_expr LOR   gbl_expr |
                        lgbl_expr LNOT  gbl_expr | lgbl_expr LNAND gbl_expr |
                        lgbl_expr LXOR  gbl_expr | lgbl_expr LXNOR gbl_expr |
                        pinname | dig_seq
        ;
```

Bitwise Logical Boolean Expression

```
btbl_expr       :       '(' btbl_expr ')' |
                        btbl_expr '&'  btbl_expr | btbl_expr '|'  btbl_expr |
                        btbl_expr '^'  btbl_expr | btbl_expr '~'  btbl_expr |
                        btbl_expr '~&' btbl_expr | btbl_expr '~|' btbl_expr |
                        btbl_expr '^~' btbl_expr |
                        pinname | dig_seq
        ;
```

Shift Expression

```
shift_expr      :       '(' shift_expr ')' |
                        shift_expr SHL shift_expr | shift_expr SHR shift_expr |
                        shift_expr ROTR shift_expr | shift_expr ROTL shift_expr |
                        pinname | dig_seq
```

Pin Name

```
pinname                 :       comp_name '.' value_ident
        ;
```

Port or Variable Identifier

```
value_ident     :       identifier | identifier '[' dig_seq ']' |
                        identifier '{' dig_seq '..' dig_seq '}'
        ;
```

Identifier

Lex Format: [a-zA-Z][a-zA-Z0-9_]*

```
identifier      :       IDENTIFIER
        ;
```

Digit Sequence

Lex Format: [0-9xX]+

```
dig_seq         :       DIGSEQ
```

APPENDIX C.

Intermediate List Syntax

C.1. General Description

This section describes, briefly, the textual format for the three lists: the units, connections, and symbols lists, associated with each state table format (see Figure 1)

C.2. The Units List

Recall that if the user partially allocates resources such as a certain number of functional units, storage elements and buses, these resources are stored in the unit list. The unit list is simply a list of unit names with details of the parameters used to instantiate the unit from a generic library such as GENUS [Dutt88]. For instance:

LU_1, *LU_2*: **GC_COMPILER_NAME**: LU, **GC_INPUT_WIDTH**: 8,
GC_NUM_FUNCTIONS:3, **GC_FUNCTION_LIST**: AND, NAND, NOR.

This fragment describes two components, *LU_1* and *LU_2*, defined to be 8 bit logic units that have the functions "AND", "NAND", and "NOR".

C.3. The Connections List

The connections lists was previously described as a method for storing a partial design structure consisting of interconnections of pre-allocated units. Though any structural netlist, VHDL, for example, would serve to accommodate this information, we have opted to define a simple format, compatible with the units list, that describes connections both by associating component pin names with nets and nets with component pin names. The following describes a simple connection list:

COMPONENTS {
LU_1,
INPUTS (*N1* => *I0*, *N2* => *I1*, ...),
OUTPUTS (*N3* => *O0*, *N4* => *O1*, ...);

LU_2,
INPUTS (*N3* => *I0*, *N4* => *I1*, ...),
OUTPUTS (...);

} **NETS** {
*N*3: **SOURCE** (*LU_*1.*O*0), **SINK** (*LU_*2.*I*0, ...);
*N*4: **SOURCE** (*LU_*1.*O*1), **SINK** (*LU_*2.*I*1, ...);
}

Each component instance has an entry in the **COMPONENTS** field describing the connections between each of its input and output pins and net junctions within the design. The **NETS** field describes connections between net junctions and component pins. Although this may seem redundant this makes the task of examining and/or restructuring the design from the connections list simpler and faster.

C.4. The Symbols List

The symbol list serves the function of a symbol table in a traditional compiler. It lists vertically all the names used in the operations fields of BIF descriptions, and horizontally the attributes associated with each name. For example:

R1 : (*SYM_type* : *COMPONENT*)
A : (*SYM_type* : *VARIABLE*, *NUM_BITS*: 16)