A Survey of Description Styles

Hsiao-Ping Juan
Daniel D. Gajski

Technical Report #93-37
August 13, 1993

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

hjuan@ics.uci.edu
gajski@ics.uci.edu

Abstract

In this report, issues related to design descriptions are discussed. Several existing description styles including timing diagrams, algorithmic-state-machine charts, state-action tables, VHDL, SpecCharts, and signal-flow graphs are investigated, and their features are compared. Five examples (addressed handshake read protocol, parallel counters, controlled counter, finite-impulse-response (FIR) filter, and a simple computer system) described in different styles are also presented.

Contents

List of Figures

1 Introduction

The issue of design description has been around for almost as long as hardware. A cursory glance at any standard component databook shows that, traditionally, a design is often specified in plain English annotated with flowcharts, state graphs, timing diagrams and block diagrams. As designs become increasingly more complex, so does the need for the description languages to raise beyond the present levels of abstraction in order to cope with the increased design complexity.

This survey will address the features of different description styles and relate these description styles to different design styles. We can divide the design styles with respect to complexity into three different categories:

1. **Interface.** This refers to low complexity designs used for communication between components. The design descriptions specify signal changes, sequences of events, and timing relationships between events. No data transformation is used in the description.

2. **FSMD (Finite State Machines with Datapaths).** This refers to medium complexity designs, which are usually described with sets of register transfers. FSMD is well-suited for modeling a design with up to several hundred states. Beyond that, it becomes incomprehensible to human designers.

3. **System.** This refers to designs of highest complexity, which are usually described with programming languages and implemented with one or more communicating processors or FSMDs.

The differences between the design styles impose vastly different requirements on the description style. To be effective, a description style should have features which can greatly reduce the effort required by the designer to describe a particular aspect, such as timing. Obviously, having a close match between the description style and the design style simplifies the task of synthesis and produces higher quality designs.

This short survey examines several description styles, including timing diagrams, Algorithmic-State-Machine Charts (ASM Charts), State-Action Tables, VHSIC Hardware Description Language (VHDL), SpecCharts, and Signal-Flow Graphs. They are believed to be representative of design description styles currently in use. We compare their capability with respect to specifying actions, states, hierarchy, concurrency, sequentiality, timing constraints, pipelining, clocking, chaining, asynchrony, synchronization, and memory. To demonstrate their features, five examples (addressed handshake read protocols, parallel counters, controlled counter, Finite-Impulse-Response filter, and a computer system) are described using different description styles.

In the next section, we briefly introduce each of the above description styles. In Section 3, different features of these styles are compared. The examples are presented in Section 4, and Section 5 contains concluding remarks.

2 Description Styles

2.1 Timing Diagram

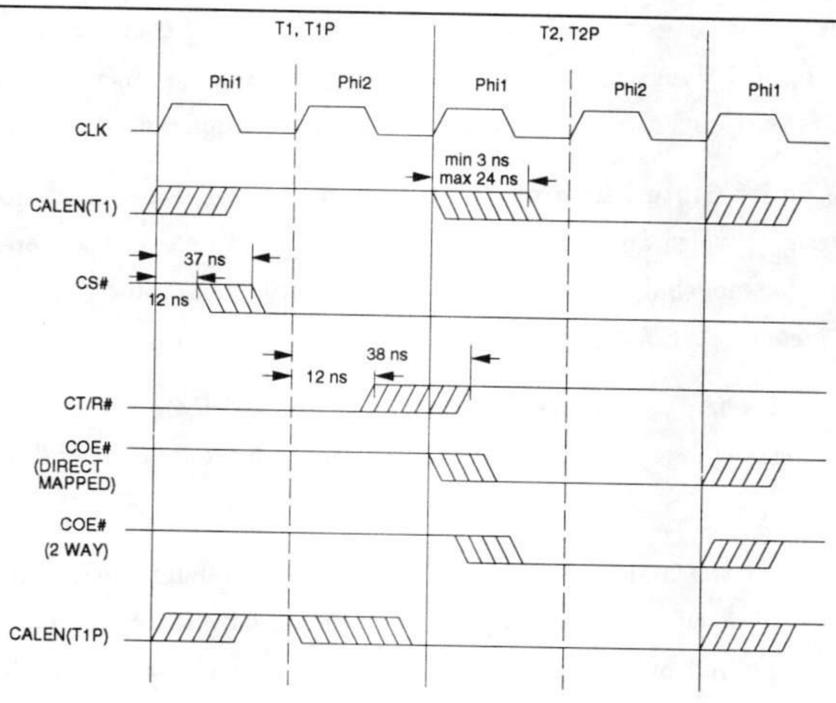


Figure 1: Example of timing diagram

Timing Diagrams are a very common description style and can be found in any component databook. They show various signals in the design as a function of time. Several signals are usually plotted with the same time scale so that the time at which these signals change with respect to each other can be easily observed. Typically, timing diagrams specify timing constraints such as clock rates, setup times, and timing relationships in terms of "ns" (nanosecond) or clock cycles. Because it can clearly show the rising and falling of signals (called *events*), sequencing of events and also timing relationships between events, it is generally used to specify protocols and I/O relationships. On the other hand, designers can not specify data transformations using timing diagrams because they lack the notation for expressions. Figure 1 shows an example of a timing diagram from the specification of the Intel 32-bit cache controller 82385 [Inte91]. This example shows how to specify signal changes related to multi-phase clocking and timing relationships such as minimum and maximum time range in timing diagrams.

2.2 Algorithmic State Machine Chart

The Algorithmic-State-Machine (ASM) chart [Clar73] is a diagrammatic description of the output function and the next-state function of a Finite State Machine (FSM). It resembles a conventional flow chart: control flow is expressed graphically, while operational behavior is described using textual assignment statements.

The chart is composed of three basic elements (Figure 2(a)): the state box, the decision box, and the conditional box. A state in the control sequence is indicated by a state box which contains a list of register operations or output signal names that the controller generates while being in this state. The exit path of the state box leads to other state boxes, decision boxes or conditional output boxes.

The decision box describes the effect of an input on the controller. Each decision box has two exit paths. One path is taken when the enclosed condition is true and the other when the condition is false. These two paths are usually indicated by 1 for true and 0 for false.

The conditional box describes register assignments or outputs which are dependent on one or more inputs in addition to the state of the FSM. The rounded corners of a conditional box differentiate it from a state box.

An ASM block is a structure consisting of one state box and all the decision and conditional boxes connected to its exit path (Figure 2(a)). Each block has one entrance and any number of exit paths represented by the structure of the decision boxes. An ASM chart consists of one or more interconnected ASM blocks. One ASM block describes the FSM operation during one state. It should be stressed that the exit paths of decision boxes, in

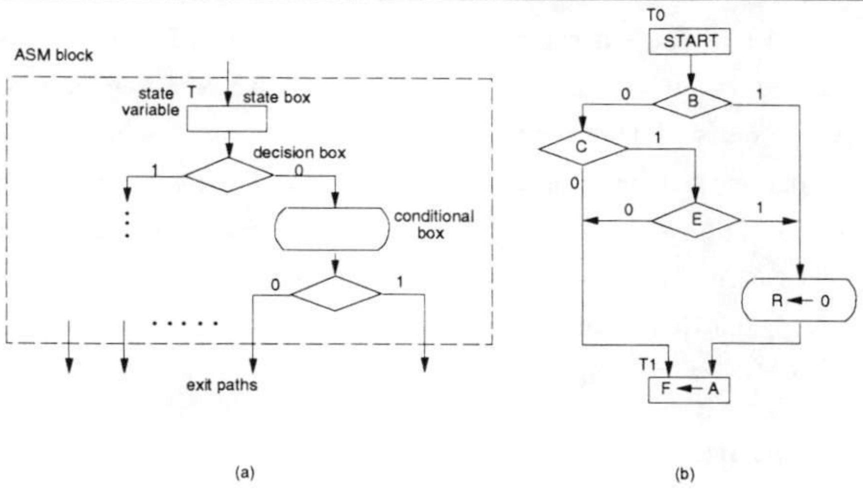


Figure 2: Example of ASM chart

no way, describe time dependence. They only represent logic relationship. The state box is the only element representing time in the ASM chart since each state box indicates one state. The timing model used in the ASM chart is a simple one-phase synchronous clocking scheme; therefore, asynchronous logic cannot be described using ASM charts. Moreover, designers cannot specify timing constraints such as component delay in ASM charts. Figure 2(b) gives an example of the ASM chart. This example shows two states T0 and T1. In T0, signal *START* is asserted. At the same time, if Boolean expression *B* is true or *B* is false and *C, E* are both true, then *R* is cleared.

2.3 State-Action Table

State-Action Table [HaCG93] provides a concise tabular notation for state-based design descriptions, where the state sequencing of the design can be clearly expressed in a state table and the datapath operations can be expressed using textual assignment statements in each state.

The State-Action Table (Figure 3) contains several different fields: *PS* (the present state), *SCOND* (the condition for a next-state transition), *NS* (the next state), *ORDER* (a level ordering of the actions within a given state), *CV* (the condition vector under which the results of the action will be used), *ACOND* (the assignment condition for each action), and *ACTIONS* (all operations in the behavior).

PS	SCOND	NS	ORDER	CV	ASCOND	ACTIONS	AC #	TIMING			
								AB	EB	SB	TB
1	T	2	0	(b==2)|!(b==2)&(b==1)	T	Z1=b−2	1	30ns		70ns	1,200ns,2
			0	T	T	F=D*E+E	2				
2	T	1	0	(b==2)	(b==2)	X=Y+Z1	3				
			0	!(b==2)	T	Z2=Y−Z1	4		4,80ns,5		
			1	T	!(b==2)&(b==1)	X=Y*Z2	5				
			1	T	!(b==2)&!(b==1)	X=0	6				

Figure 3: Example of state-action table

The assignment condition for each action (*ACOND*) can be an asynchronous input-signal, a clock-signal change or any valid expression resulting in a Boolean value. In the field *ACTIONS*, the designer can specify functions as well as simple assignments. The functions which define multiple operations may take place over multiple time steps, and may have multiple return values. Using functions, the designer can specify *operator pipelining* and *multi-cycle operators*. Operator pipelining allows a single operator to be mapped to a pipelined component which will take one or more states to complete an operation. Multi-cycling allows a single operation to be divided into some number of sequential time steps. By specifying the execution sequence using the field *ORDER*, the state-action table also allows several operations to be chained together into a single state (called *chaining*).

The state-action table also allows the designer to specify four kinds of timing constraints. *Action-Based constraints* specify timing values over a single action. *Expression-Based constraints* specify timing values across several sequential actions. *State-Based constraints* show timing values for an entire state. And *Transition-Based constraints* show timing values across several sequential states. In Figure 3, an action-based (AB) constraint is specified on the action $Z1 = b - 2$. Similarly, a state-based (SB) constraint is shown for state 1. And an expression-based (EB) constraint indicates that the time constraint from action 4 ($Z2 = Y - Z1$) to action 6 ($X = Y * Z2$) is 80 ns. Similarly, the transition-based (TB) constraint indicates that the time constraint from state 1 to state 2 is 200 ns.

Multiple clock phases can be represented in state-action tables by introducing a hierarchy between states and atomic actions. Figure 4 shows the same example as shown in Figure 3, but using a two-phase clock. The clock phases (*PHASE*) are always offset from the current state and proceed in the order they are numbered. The phase order (*PORDER*) column is equivalent to the *ORDER* column with the numbers offset from the current phase.

PS	SCOND	NS	PHASE	ORDER	PORDER	CV	ASCOND	ACTIONS
1	T	1	1	0	0	(b==2)||(b==2)&(b==1)	T	Z1=b-2
				0	0	T	T	F=D*E+E
			2	1	0	(b==2)	(b==2)	X=Y+Z1
				1	0	!(b==2)	T	Z2=Y-Z1
				2	1	T	!(b==2)&(b==1)	X=Y*Z2
				2	1	T	!(b==2)&!(b==1)	X=0

Figure 4: Two-phase clock representation in state-action table

2.4 VHDL

VHDL (VHSIC Hardware Description Language) [IEEE88] is an industry standard language used to describe hardware from the abstract to the concrete level. It exhibits semantics common to high-level programming languages, such as data abstraction, behavioral operators, assignment statements, and control and execution ordering constructs to express conditional and repetitive behavior. For instance, the data types allowed in VHDL consist of everything from scalar numeric types, to composite arrays, records, and file types.

The primary hardware abstraction in VHDL is an *entity*. It represents a portion of the design which performs a specific function and possesses well-defined inputs and outputs. A design entity may represent any abstraction level from a logic gate to a complete system, and it can be described in terms of a hierarchy of subentities. The relationship between the inputs and outputs of a design entity may be described in terms of behavior, dataflow, structure or any combination thereof.

In behavioral descriptions, a design can be modeled as a set of concurrently executing processes. Statements within a process are executed sequentially. The execution of each process can be controlled by making the process sensitive to changes on certain synchronous or asynchronous signals. Figure 5(a) shows the behavioral description of a simple modulo-8 counter. To model pipelining, a process can be partitioned into successive, synchronized stages such that multiple processes, each in a stage different from others, can be executed in parallel.

In data-flow descriptions, a design can be modeled as a hierarchy of data-flow blocks. Statements within a data-flow block are executed concurrently. Each statement assigns a value to a signal based on a set of conditions or control expressions. VHDL allows a guard condition to be associated with a data-flow block, which enables conditional assignments of selected statements in the block. Figure 5(b) show the data-flow style description of the counter.

In structural descriptions, the designer can instantiate components which represent previously defined design entities and specify their interconnections by mapping component ports to the same signal. The design structure can still be maintained hierarchically. The structural description of the counter is given in Figure 5 (c).

```
entity COUNTER is
  port(CLK : in bit; CNT : out integer);
end COUNTER;

architechture A of COUNTER is
begin
 process
 begin
   wait until (CLK='1') and not CLK'stable;
   if (CNT=7) then
      CNT <= 0;
   else
      CNT <= CNT + 1;
   end if;
 end process;
end A;
```

(a)

```
entity COUNTER is
   port (CLK: in bit; CNT: out integer);
end COUNTER;

architecture A of COUNTER is
begin
 block (CLK='1' and not CLK'stable)
 begin
   CNT <= guarded 0 when CNT=7 else
                              CNT+1;
 end block;
end A;
```

(b)

```
entity COUNTER is
   port (CLK: in bit; CNT: out integer);
end entity;

architecture A of COUNTER is
  component REG
    port (D: in integer; CLK: in bit; Q: out integer);
  end component;

  component ADD
    port (A, B: in integer; O: out integer);
  end component;

  component MUX
    port (I0, I1: in integer; SEL: in bit; O: out integer);
  end component;
  ..............................

  signal ONE: integer := 1;
  signal ZERO: integer := 0;
  signal SEL, ADD_OUT, MUX_OUT: integer;
  ..............................
begin
 REGISTER: REG
    port map(MUX_OUT, CLK, CNT);
 ADDER: ADD
    port map(CNT, ONE, ADD_OUT);
 MUX: MUX
    port map (ZERO, ADD_OUT, SEL, MUX_OUT);
  ............................
end A;
```

(c)

Figure 5: Example of VHDL description styles: (a)behavioral; (b) data-flow; (c) structural

VHDL also provides timing specifications: *wait* statement and *after* clause. The wait statement can be used in several ways such as delay specification ("wait for") , synchronization ("wait until", "wait on") and timeout clause ("wait on" *signal* "for" *time*). The after clause specifies the time into the future when the value of the signal is to be updated with the new value, which in a way, places a timing constraint on the assignment statement.

In a description with several concurrent processes, synchronizing the processes is often required because two processes may need to exchange data or certain actions must be performed by different processes at the same time. *Wait* statements of VHDL in two processes with a common signal will synchronize the two processes when an event occurs on the common signal. In addition, any event on a signal which occurs in the sensitivity list of two or more processes will start the processes simultaneously. An example is a global clock signal which can be used to synchronize the entire system.

In asynchronous interprocess communication, global signals are declared which can be accessed by concurrent processes. The sending processes update the global signals which are then read by all the receiving processes. The process which is ready first usually waits for the other processes to be ready for the communication. The protocol to implement the transfer has to be specified as a part of the description of the processes.

2.5 SpecCharts

The SpecCharts language [NaVG91] [VaNG91] consists of a hierarchy of states, represented in combined graphical and textual form, while catering to the expression of concurrent behavior and specification of constraints.

The basic object in SpecCharts is a behavior (i.e., a state). A behavior can be expressed in one of the three ways. First, a behavior may itself consist of several sub-behaviors sequenced by transition arcs. This corresponds to traditional state diagrams. Secondly, a behavior may consist of concurrent sub-behaviors (often called *processes*). Thirdly, a behavior may be a leaf behavior which is not further decomposed into sub-behaviors. Such a behavior has sequential VHDL code only.

Briefly, behaviors are represented as boxes, sequential behaviors are sequenced by transition arcs and concurrent behaviors are separated by dotted lines. Each behavior may contain declarations identical to those in VHDL. Figure 6 shows a CPU described in SpecCharts. At the topmost level, the description SYSTEM consists of two concurrently executing sub-behaviors, the processor CPU and the clock generator CLK_GEN. The CPU in turn consists of two major states: RESET, which is the default entry state for the CPU, and ACTIVE, where the processor is operational. The ACTIVE state is further composed of three sequential sub-behaviors representing instruction fetch, decode and execution, respectively. CLK_GEN is a leaf state which contains sequential VHDL code only.

SpecCharts provides two different kinds of transition arcs: TOC (Transition-On-Completion) and TI (Transition-Immediately). A TOC arc is traversed only if the source behavior has completed execution and the arc condition is true. A TI arc is traversed the instance its associated condition becomes true, regardless of the execution status of the source behavior

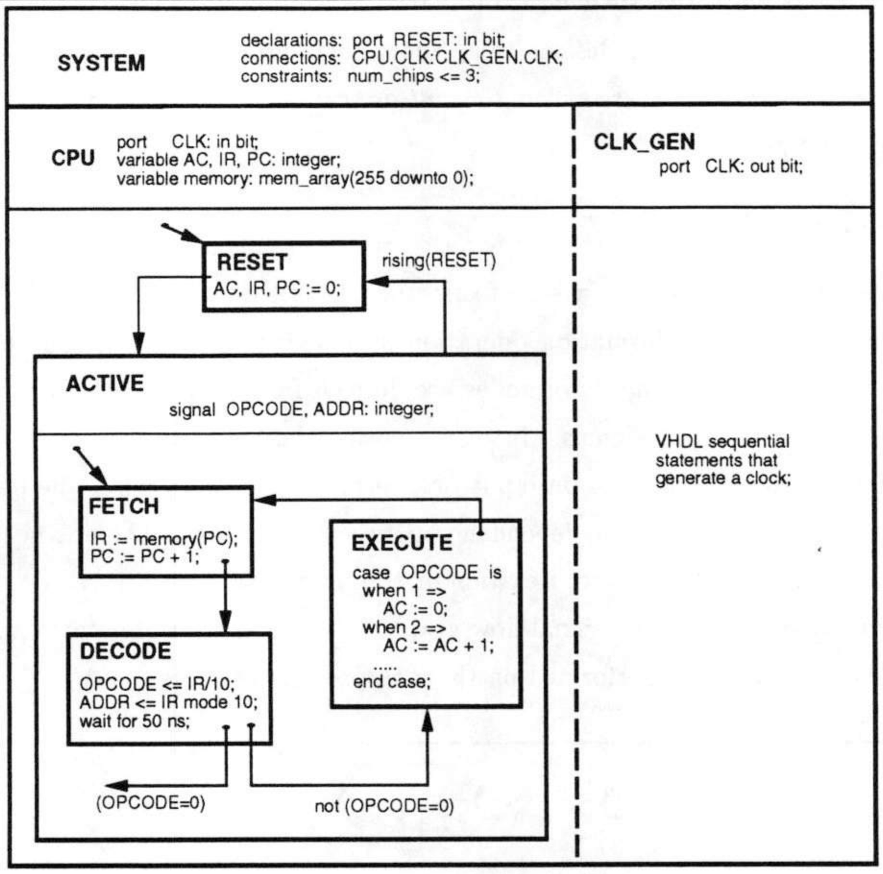


Figure 6: Example of SpecCharts

and any of its descendant behaviors. For instance, in Figure 6, the arc from ACTIVE to RESET is a TI arc; therefore, the event *rising(RESET)* forces the CPU process into the RESET state, no matter which ACTIVE substate it is currently in.

Interface information such as external ports and communication channels between concurrent sub-behaviors can be specified either by using global signals or by *channels*. A channel is a combination of ports and protocols. A protocol definition is identical to a procedure definition and contains data transfer statements such as placing address on the bus, generating a request, waiting for an acknowledgement, etc. The designer simply declares a channel at a module boundary, associating a descriptive label with it, and specifies the protocol of that channel. The channel label can then be used throughout the code as any procedure call.

Since SpecCharts semantics are identical to VHDL, the designer can use the same timing specifications as used in VHDL. Besides this, SpecCharts provides a special construct to represent timeout: a transition arc called *timeout arc* causes a transition on the expiry of the associated time period.

2.6 Signal-Flow Graph

A signal-flow graph is basically a set of directed paths that connect at nodes. Each node represents a data transformation operation such as an addition or multiplication. The flow paths show how the inputs of nodes are derived from appropriate outputs of the other nodes or external input signals. In other words, the nodes in a signal-flow graph represent functions to be performed on input or intermediate signals, while the flow paths represent data flows as well as time-dependence between operations. There is no notation for conditional expressions, recursion, iteration or timing constraints.

Figure 7 gives an example of a signal-flow graph. It shows the set of input data *a, b, c,* and *in*, and the computations performed on them to produce the signal *out*.

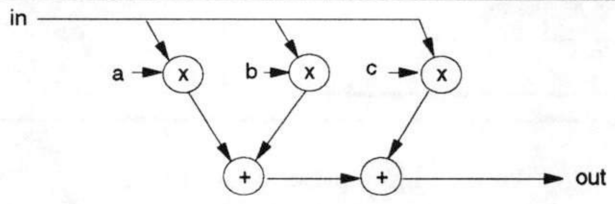


Figure 7: Example of signal-flow graph

3 Description Styles Evaluation

We have described the different description styles described in the previous section. Different features of these description styles are summarized in Figure 8 with respect to the capability of specifying *actions*, *states*, *hierarchy*, *concurrency*, *sequentiality*, *timing constraints*, *pipelining*, *chaining*, *clocking*, *asynchrony*, *synchronization* and *memory*. We will now evaluate and compare the styles with respect to these features.

	Actions	States	Hierarchy	Concurrency	Sequentiality	Timing constraints
Timing Diagram	events	No	No	concurrent event	sequence of events	timing constraints related to events
ASM	simple assignments	explicit state specification	No	assignments within each state	sequence of states	No
State–Action Table	simple assignments, functions	explicit state specification	procedural hierarchy	actions with the same order and in the same state	chaining within states; sequence of states	action–based; expression–based; state–based; transition–based
VHDL	programming constructs	using "case","loop" and state variable	structural hierarchy	process level; statement level	statements within process	component delay; timeout clause
SpecCharts	programming constructs	explicit state specification	behavioral hierarchy	concurrent sub–behavior at any level	sequence of states; sequential statements	delay specification; timeout arcs
Signal–Flow Graph	data transformation operators	No	No	operators without data dependency	data dependencies	No

	Pipelining	Clocking	Chaining	Asynchrony	Memory	Synchronization
Timing Diagram	No	multi–phase clock	No	asynchronous events	No	No
ASM	No	one–phase clock	No	No	No	No
State–Action Table	pipelined operators	multi–phase clock	Yes	actions executed on asynchronous input signal edge	No	No
VHDL	concurrent processes as stages of pipeline	multi–phase clock	Yes	asynchronous communication processes	declared as variable array	common events; global signal
SpecCharts	concurrent processes as stages of pipeline	multi–phase clock	Yes	asynchronous communication processes	declared as variable array	common events; global signal; communication channels
Signal–Flow Graph	No	No	No	No	No	No

Figure 8: Features of description styles

3.1 Timing Diagram

Timing diagrams present synchronous and asynchronous signals uniformly and make it trivial to identify signal events and interrelate them with timing constraints. Hence, timing diagrams can be used to describe the events and timing constraints on the IO pins. On the other hand, they lack notations to specify design implementation used to generate those events. This information is very useful in initial phases of design when complete design details are not known. One example is the specification of a system backplane bus protocol and the timing constraints that must be respected for proper operation.

There are also many limitations to using timing diagrams. There are no conventions for specifying conditional and looping behavior nor are there ways of expressing hierarchy. Therefore, a complete specification usually needs more than one timing diagrams to represent all possible cases of the same set of signals. For instance, the Intel cache controller 82385 specification [Inte91] consists of 3 diagrams: cache write hit cycle, cache read miss cycle, and cache read cycle (Figure 1). Since there is no data transformation specified in timing diagram, they can be implemented with FSMs. However, extracting control flow from timing diagrams is difficult since control constructs such as conditional behaviors cannot be specified explicitly in timing diagrams.

3.2 ASM Chart

The ASM chart has explicit state specification, which requires the designer to finish scheduling before the design can be described using the ASM chart. Every block in an ASM chart specifies the operations that are to be performed during the same clock period (state). The timing model used in the ASM chart is a very simple one-phase clocking scheme: every operation is synchronized by a global clock signal and no timing constraint can be specified. The requirements for the design of the datapath are specified inside the state and conditional boxes. The control logic is determined from the decision boxes and the required state transitions.

Because operations are written at the register-transfer level, datapath synthesis can be done simply by doing allocation. By following the control flow in the chart, boolean expressions for every datapath control signal can be generated, from which the control logic is synthesized. For example, consider the ASM chart shown in Figure 2. While in state T0, the control circuit evaluates the Boolean expressions *B, C, E* and generates control signals to clear register R. Suppose the control signal which resets register R is called *clear*, then we know that

$$clear = \mathrm{T0} \wedge (\mathrm{B} \vee (\bar{\mathrm{B}} \wedge \mathrm{C} \wedge \mathrm{E})).$$

As shown above, the design implementation can be obtained from the ASM chart description relatively easy. On the other hand, highly hierarchical designs will increase the number of states in ASM charts exponentially ASM charts are inherently flat and sequential. Therefore, ASM chart is well-suited to describe small FSMDs resulted by decomposing complex systems in some intermediate design phases.

3.3 State-Action table

State-action table is a state-based description, and it has many similarities compared to the ASM chart. For instance, scheduling is automatically achieved by describing an operation in its appropriate state; it shows exponential increase in number of states when describing hierarchical systems; the requirements for the design of datapath are specified as actions and written in register-transfer level, and thus can be synthesized without much effort, etc. But it also differs from the ASM chart in several ways.

The state-action table provides procedural hierarchy by using "functions" as a shorthand notation for describing repetitive behavior. This procedural hierarchy permits grouping of actions in a structured fashion and allows a concise representation. Using functions, the designer can specify pipelined and multi-cycle operators and any other hardware.

The state-action table also allows several operators to be chained together into a single state while ASM charts assume that all operations within a state are synchronized with the clock signal. The state-action table can also be used to specified asynchronous designs, but the ASM chart is limited to synchronous logic because of its one-phase clock timing model.

3.4 VHDL

VHDL has a rich set of language constructs, such as behavioral operators, which allow it to succinctly capture behavior that contains assignment statements with complex data transformations carried out by an algorithm.

The concepts of states, state hierarchy, state transitions, etc., are not present in VHDL.

Thus, for state-based designs, the state activation and transitions need to be explicitly specified in detail using state variables and control constructs such as "case" statements. This can prove to be cumbersome, error-prone, and reduces understandability of descriptions.

To describe complex systems, VHDL supports structural hierarchy which allows the designer to decompose a large complicated system into smaller and more comprehensible subsystems. Each of the subsystems may represent behavior or structure. This capability becomes vital as design complexity grows.

It is noted that synthesis from VHDL descriptions needs more effort than from the ASM chart or the state-action table because it does not explicitly specify states, control and register transfers. For instance, since VHDL does not specify control steps explicitly, a VHDL description must be partitioned into control steps before it can be realized as FSMD structure.

3.5 SpecCharts

A state-based description with simple assignments is difficult to use for system level designs. In the absence of programming constructs, implementing even a simply looping mechanism will require a complicated set of state transitions, making design description cumbersome and less readable. Therefore, description styles such as the ASM chart and the state-action table are limited to describe small FSMDs.

On the other hand, programming languages such as VHDL have a rich set of programming constructs, but they do not have state-based specification. Besides, VHDL provides only a single level of concurrency (processes) followed by one level of sequentiality (sequential statements in the process). Therefore, for highly hierarchical designs, the designer has to coerce the design specification into a description of a single-level of concurrent processes, which may not always be possible. Besides, designers do not usually think in terms of programming languages when designing system; instead, they draw state diagrams, and think of protocols, etc.

SpecCharts is essentially a combination of behavioral VHDL and hierarchical/concurrent state diagrams. It has programming constructs like in VHDL and thus inherits the merits from it. Furthermore, it includes state-based specification and supports behavioral hierarchy to concisely capture complex behavior in a way closer to the designer's conceptual view of the system. In short, SpecCharts is intended for system descriptions.

3.6 Signal-Flow Graph

A signal-flow graph shows data operations and data flow. It has no notation for control flow or conditional operations. Therefore, it is usually used in digital signal processing where very little control flow is needed and a relatively large amount of data transformations are performed.

Hardware designers do not think in terms of equations or algorithms when they design circuits. The signal-flow graphs can be derived by expanding equations in the algorithm or in the program. They also provide a good visual aid for designers to gain a better understanding of the equations. By analyzing the signal-flow graphs, the designer can obtain the datapath according to different design constraints. However, the signal-flow graphs do not have notations to specify control flows. Therefore, to obtain the control logic, the signal-flow graphs need to be annotated by some other description styles.

4 Examples

In this section, we describe a large variety of examples using different description styles. These examples are chosen to demonstrate the applications of the description styles. The first example is the *Addressed Handshake Read Protocol* which represents a typical interface protocol. The second example *Parallel Counters*, brings out the features of concurrent behaviors. The third example is an asynchronous circuit: *Controlled Counter. Finite-Impulse-Response (FIR) Filter* is a DSP application which is given by a mathematical formula. The last example is a *Simple Computer System*. It shows the importance of hierarchy when describing a complex system.

4.1 Example 1: Addressed Handshake Read Protocol

This example is a well-known read protocol used for memory access. The processor initiates the data transfer by placing the address on the *ADDRESS* bus and asserting the *READ* signal. The *READY* signal is activated by the memory after it places the data on the *DATA* bus. The processor deasserts the *READ* signal after it stores the data. The memory then deasserts its *READY* signal, and invalidates the data on the bus.

Timing diagram is a natural choice to describe this protocol. The timing diagram in Figure 9 clearly shows the exchange of signals between the processor and the memory. For

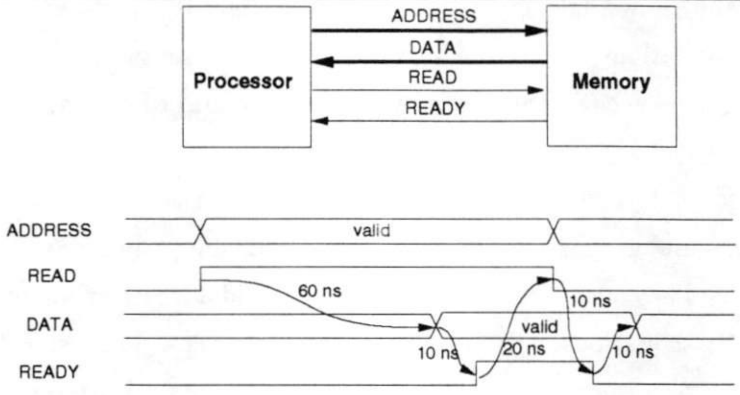


Figure 9: Description of addressed handshake read protocol using timing diagram

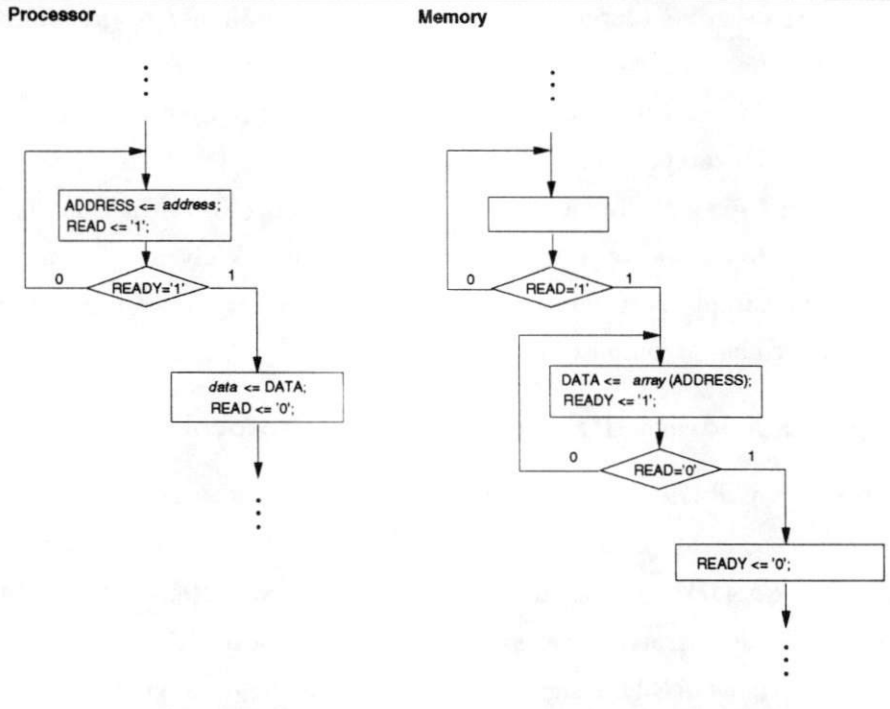


Figure 10: Description of addressed handshake read protocol using ASM chart

Processor

PS	SCOND	NS	ORDER	CV	ACOND	ACTIONS
⋮	⋮	⋮	⋮	⋮	⋮	⋮
i	READY	i+1	0	T	T	ADDRESS=address
	!READY	i	0	T	T	READ='1'
i+1	T	i+2	0	T	T	data=DATA
			0	T	T	READ='0'
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Memory

PS	SCOND	NS	ORDER	CV	ACOND	ACTIONS
⋮	⋮	⋮	⋮	⋮	⋮	⋮
i	READ	i+1	0	T	T	no–operation
	!READ	i	0	T	T	no–operation
i+1	!READ	i+2	0	T	T	DATA=memory(ADDRESS)
	READ	i+1	0	T	T	READY='1'
i+2	T	i+3	0	T	T	READY='0'
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Figure 11: Description of addressed handshake read protocol using state-action tables

the purpose of simulation, this timing diagram provides concise and adequate information about the interface behaviors. No unnecessary information such as internal circuit or the interface logic implementation is described. To implement the interface logic, the timing diagram needs to be partitioned into control steps, then it can be implemented with FSMs.

In comparison with the timing diagram, Figures 10 and 11 show the interface control logic descriptions using the ASM chart and the state-action table respectively. These two FSMs (the processor and the memory) can be synchronized with different clock signals.

```
process Processor
begin
  .....
  .....
  ADDRESS <= address;
  READ <= '1';
  wait until READY='1';
  data <= DATA after 20 ns;
  READ <= '0' after 20 ns;
  .....
  .....
end;
```

```
process Memory
  variable memory: mem_array;
begin
  wait until READ='1';
  DATA <= memory(ADDRESS) after 60 ns;
  READY <= '1' after 70 ns;
  wait until READ='0';
  READY <= '0' after 10 ns;
end;
```

Figure 12: Description of addressed handshake read protocol using VHDL

The designer can also describe the processor and memory as two concurrent processes in VHDL. The communication between these two processes is via addressed handshake protocol (Figures 12). To observe or modify the interface behaviors is more difficult now because they are splitted into two processes. The VHDL code can also be partitioned into control steps using scheduling technique and result in FSMs as described in Figure 10 and 11.

Figures 10, 11 and 12 show the memory read operations only, but it should be stressed that these statements may be only part of the entire descriptions of the processor and the memory. This makes it more difficult to modify the interface descriptions since they are mixed with the internal circuit descriptions.

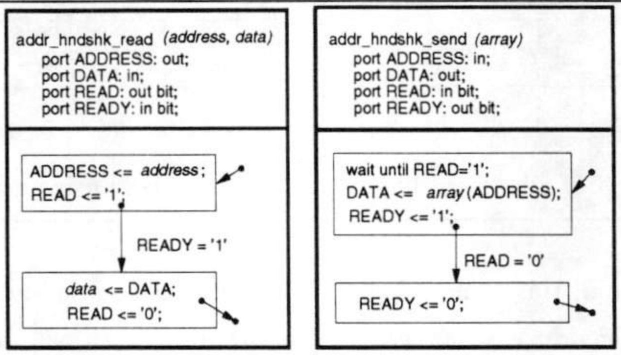
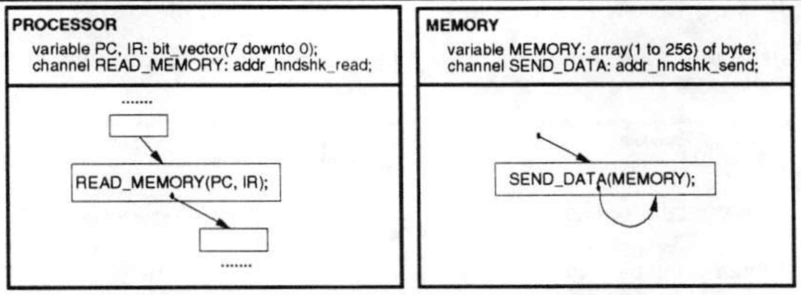


Figure 13: SpecCharts description of the address handshake protocol



PROCESSOR
variable PC, IR: bit_vector(7 downto 0);
channel READ_MEMORY: addr_hndshk_read;
READ_MEMORY(PC, IR);
MEMORY
variable MEMORY: array(1 to 256) of byte;
channel SEND_DATA: addr_hndshk_send;
SEND_DATA(MEMORY);

Figure 14: Channel declaration and parameterized protocol instantiation

Moreover, the designer can describe the protocol using SpecCharts. SpecCharts provides a special construct called *channel* for interprocess communication. In Figure 13, the two complementary halves of the protocol are shown as parameterizable SpecCharts states, with the actual low-level data transfer statements that constitute the protocol. The designer simply has to specify the protocol and the italicized parameters at each access point. For instance, a channel is declared on both the processor and the memory, and the appropriate protocol associated with each (Figure 14). This construct *channel* separates the descriptions of the interface and the internal behaviors as using the timing diagram, and the low-level descriptions of the protocol have been assigned to states as using the ASM chart or the state-action table. But comparing to the timing diagram, the signal exchanges between the processor and the memory is not as easy to observe.

4.2 Example 2: Parallel Counters

This example consists of N-independent counters where each counter has a separate *load* signal, *start* signal, M-bit *limit* signal and a *time_out* signal (Figure 15). The M-bit *limit* is loaded into the counter when the signal *load* is high. When the counter receives a *start* signal, it counts down on every clock cycle, and sets *time_out* signal high when it reaches zero.

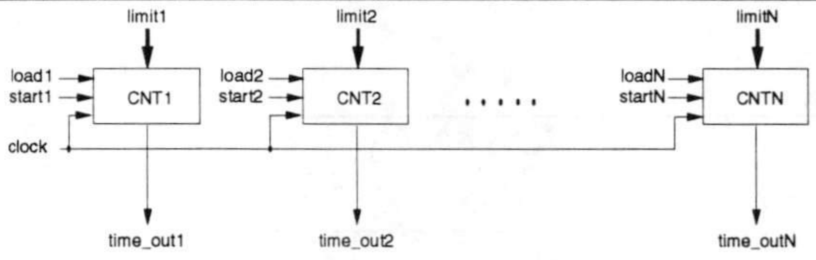


Figure 15: N-independent parallel counters

The functional requirements can be easily transformed into a VHDL process (Figure 16), an ASM chart(Figure 17), or a state-action table (Figure 18).

In VHDL, N-independent counters can be represented using one process with a loop as shown in Figure 16. Although the statements in the VHDL process are executed sequentially, the behaviors of the N counters described by the loop can still be viewed as concurrent since there is no time elapsed within the loop and all the statements are executed at the same time. It should be noted that if there is timing constraints specified using wait statements or after clauses for the individual counter, then the N parallel counters have to be

```
CNT1: process(clock)
begin
  for i in 0 to N-1 loop
    if load(i)=1 then
      CNT(i) <= limit(i);
    elsif start(i)=1 then
      count(i) := 1;
      CNT(i) <= CNT(i) - 1;
    elsif count(i)=1 then
      if CNT(i)=0 then
        count(i) := 0;
        time_out(i) <= 1;
      else
        CNT(i) <= CNT(i) - 1;
      end if;
    end if;
  end loop;
end process;
```

Figure 16: Description of N-independent parallel counters using VHDL

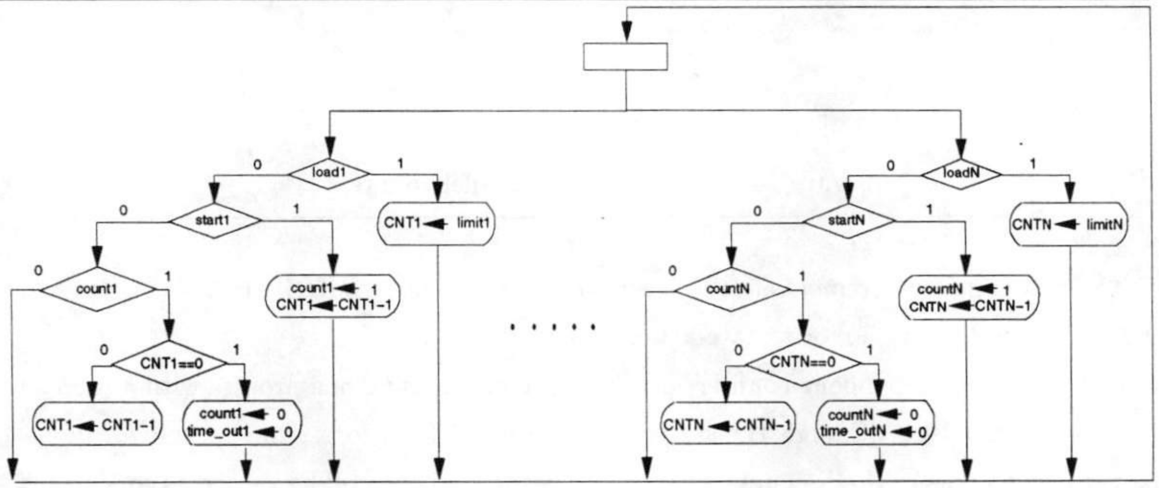


Figure 17: Description of N-independent parallel counters using ASM chart

PS	SCOND	NS	ORDER	CV	ASCOND	ACTIONS
1	T	1	0	(load1==1)	(load1==1)	CNT1=limit1
			0	!(load1==1)&(start1==1)	!(load1==1)&(start1==1)	count1=1
			0	!(load1==1)&(start1==1)	!(load1==1)&(start1==1)	CNT1=CNT1-1
			0	!(load1==1)&!(start1==1)&(count1==1)&!(CNT1==0)	!(load1==1)&!(start1==1)&(count1==1)&!(CNT1==0)	CNT1=CNT1-1
			0	!(load1==1)&!(start1==1)&(count1==1)&(CNT1==0)	!(load1==1)&!(start1==1)&(count1==1)&(CNT1==0)	count1=0
			0	!(load1==1)&!(start1==1)&(count1==1)&(CNT1==0)	!(load1==1)&!(start1==1)&(count1==1)&(CNT1==0)	time_out1=1
			0	(load2==1)	(load2==1)	CNT2=limit2
			0	!(load2==1)&(start2==1)	!(load2==1)&(start2==1)	count2=1
			0	!(load2==1)&(start2==1)	!(load2==1)&(start2==1)	CNT2=CNT2-1
			0	!(load2==1)&!(start2==1)&(count2==1)&!(CNT2==0)	!(load2==1)&!(start2==1)&(count2==1)&!(CNT2==0)	CNT2=CNT2-1
			0	!(load2==1)&!(start2==1)&(count2==1)&(CNT2==0)	!(load2==1)&!(start2==1)&(count2==1)&(CNT2==0)	count2=0
			0	!(load2==1)&!(start2==1)&(count2==1)&(CNT2==0)	!(load2==1)&!(start2==1)&(count2==1)&(CNT2==0)	time_out2=1
			⋮	⋮	⋮	⋮
			0	(loadN==1)	(loadN==1)	CNTN=limitN
			0	!(loadN==1)&(startN==1)	!(loadN==1)&(startN==1)	countN=1
			0	!(loadN==1)&(startN==1)	!(loadN==1)&(startN==1)	CNTN=CNTN-1
			0	!(loadN==1)&!(startN==1)&(countN==1)&!(CNTN==0)	!(loadN==1)&!(startN==1)&(countN==1)&!(CNTN==0)	CNTN=CNTN-1
			0	!(loadN==1)&!(startN==1)&(countN==1)&(CNTN==0)	!(loadN==1)&!(startN==1)&(countN==1)&(CNTN==0)	countN=0
			0	!(loadN==1)&!(startN==1)&(countN==1)&(CNTN==0)	!(loadN==1)&!(startN==1)&(countN==1)&(CNTN==0)	time_outN=1

Figure 18: Description of N-independent parallel counters using State-action table

described using N concurrent processes individually.

The ASM chart and the state-action table are inherently sequential and they do not have any notation for higher level concurrent behaviors such as processes or blocks; therefore, it becomes cumbersome to describe N parallel counters using the ASM chart and the state-action table. The designer has to duplicate the description for one counter N times to specify N parallel counters. Figure 17 shows the description of this example using the ASM chart. Figure 18 gives the description of N-independent parallel counters in the state-action table.

The example shows that VHDL description is short and more readable because it benefits from the loop constructs. But to implement the design from this description, the designer has to recognize the parallelism across the loop iterations. Exploiting the parallelism and partitioning the VHDL code into control steps becomes a difficult task. On the other hand, the ASM chart and the state-action table descriptions are lengthy, but the parallelism is specified explicitly. It requires the designer less effort to obtain the implementation from the ASM chart and the state-action table descriptions.

4.3 Example 3: Controlled Counter

The controlled counter is a 4-bit counter which can count up or down on each rising clock, to a specified limit, and can be asynchronously cleared. It consists of three main components: a 2-bit control register (*CNTL*), a 4-bit limit register (*LIMIT*), and a 4-bit counter (*CNT*). The output of the counter is *OUT*. On the rising edge of the *STRB* signal, the counter stores the 2-bit control input signal *CON*. It then performs the following operations based on the stored value of *CON*:

1. "00": Clear the counter.
2. "01": Load inputs from the *DATA* signals into the *LIMIT* register on the falling edge of the *STRB* signal.
3. "10": Increment the counter *CNT* on the rising edge of the clock signal *CLK*.
4. "11": Decrement the counter *CNT* on the rising edge of the clock signal *CLK*.

The difference between the N-independent parallel counters and this controlled counter is that the behavior of the controlled counter consists of both synchronous and asynchronous parts. The synchronous part is performed on the rising edge of the *CLK* signal while the asynchronous part is performed on the rising edge of the *STRB* signal. State-action table, VHDL and SpecCharts can be used to describe the controlled counter because of their capability in specifying asynchrony. Here, we show the state-action table and VHDL descriptions only.

							TIMING(ns)	
PS	SCOND	NS	ORDER	ASCOND.e	ASCOND.l	ACTIONS	ACTION	EXPRESSION
1	T	1	0	STRB	T	CON=CNTL		40
			1	T	CON=="00"	CNT="0000"		
			1	STRB'	CON=="01"	LIMIT=DATA		
			1	CLK	CON=="10"&CNT!=LIMIT	CNT=CNT+"0001"	30	
			1	CLK	CON=="11"&CNT!=LIMIT	CNT=CNT-"0001"	30	
			2	T	T	OUT=CNT		

Figure 19: Description of controlled counter using state-action table

Figure 19 shows the state-action table description of the controlled counter. The field *ASCOND* is split into two subfields: ASCOND.e specifies the edge-triggered conditions, and ASCOND.l specifies the level-triggered conditions. The VHDL description is shown in

Figure 20. Wait statements are used in VHDL code to represent the asynchrony while the state-action table does not need additional constructs. Hence, the state-action table gives a more concise description.

```
process
begin
  wait until not STRB'stable;

  if STRB='1' then CON := CNTL; end if;

  case CON is
     when "00" =>
           CNT := "0000";
     when "01" =>
           wait until STRB='0' and not STRB'stable;
           LIMIT := DATA;
     when "10" =>
           wait until CLK='1' and not CLK'stable;
           if CNT/=LIMIT then
             CNT := CNT+"0001" after 30 ns;
           end if;
     when "11" =>
           wait until CLK='1' and not CLK'stable;
           if CNT/=LIMIT then
             CNT := CNT-"0001" after 30 ns;
           end if;
     when others => null;
   end case;

   OUT <= CNT;
end process;
```

Figure 20: Description of controlled counter using VHDL

We can also specify the component delays in VHDL using after clause. For instance, Figure 20 shows that the increment and decrement operations need 30 ns each. In state-action table, the component delays can be specified using action-based constraints as shown in Figure 19. On the other hand, using expression-based constraints, the state-action table can specify timing values across sequential actions while VHDL cannot.

4.4 Example 4: Finite-Impulse-Response (FIR) filter

```
process
begin
  i := 0;
  for i in 0 to 15 loop
    for k in 0 to 3 loop
       if (i-k>0) or (i-k=0) then
         y[i] <= b[i]x[i-k] + y[i];
       end if;
    end loop;
  end loop;
end process;
```

Figure 21: Description of FIR filter using VHDL

The FIR filter response is given by the following equation:

$$y[i] = \sum_{k=0}^{M-1} x[i-k]b[k]$$

where x[0..N-1] is the input stream, y[0..N-1] is the output stream and b[0..M-1] is the array of filter coefficients (M is the order of the filter, N is the number of samples, and i, k are the filter sample index and the filter order index respectively). This equation can be naturally written as an iterative algorithm using VHDL (Figure 21) because of the programming constructs it provides.

The VHDL description shows that this example has little control but a large amount of data transformations, and we know a signal-flow graph is suitable to describe designs with this nature. Figure 22 shows the signal-flow graph for an FIR filter ($M = 4$, $N = 16$). The signal-flow graph represents the most parallel design resulted from unfolding the loops in the algorithm.

To demonstrate how signal-flow graphs help designers analyzing data flows and determining datapath, we now assume there are four multipliers available. Figures 22 and 23 shows four different ways of utilizing these four multipliers. The four multiplications grouped in the dashed boxes (called *slice*) represent the operations to be performed on the four multipliers concurrently. The dashed arcs pointing from one slice to another represent dataflow which requires data storage. There are a number of selection rules for slicing depending on different design constraints[BaGa93]. For instance, here, to minimize the data buffering requirements, designers should choose horizontal slicing since it has minimal number of connections crossing the slices.

Now the datapath is chosen, the algorithm can be rewritten to reflect the datapath (Figure 24). To obtain the implementation, we need to schedule the VHDL process into control steps. Since VHDL cannot specify control steps explicitly, state-based description styles are more effective to describe the design after scheduling. Here, we describe the design using both the state-action table (Figure 25) and the ASM chart (Figure 26). Comparing the state-action table and the ASM chart, it is noted that the ASM chart has more states (9) than the state-action table (3) because it cannot specify action chaining.

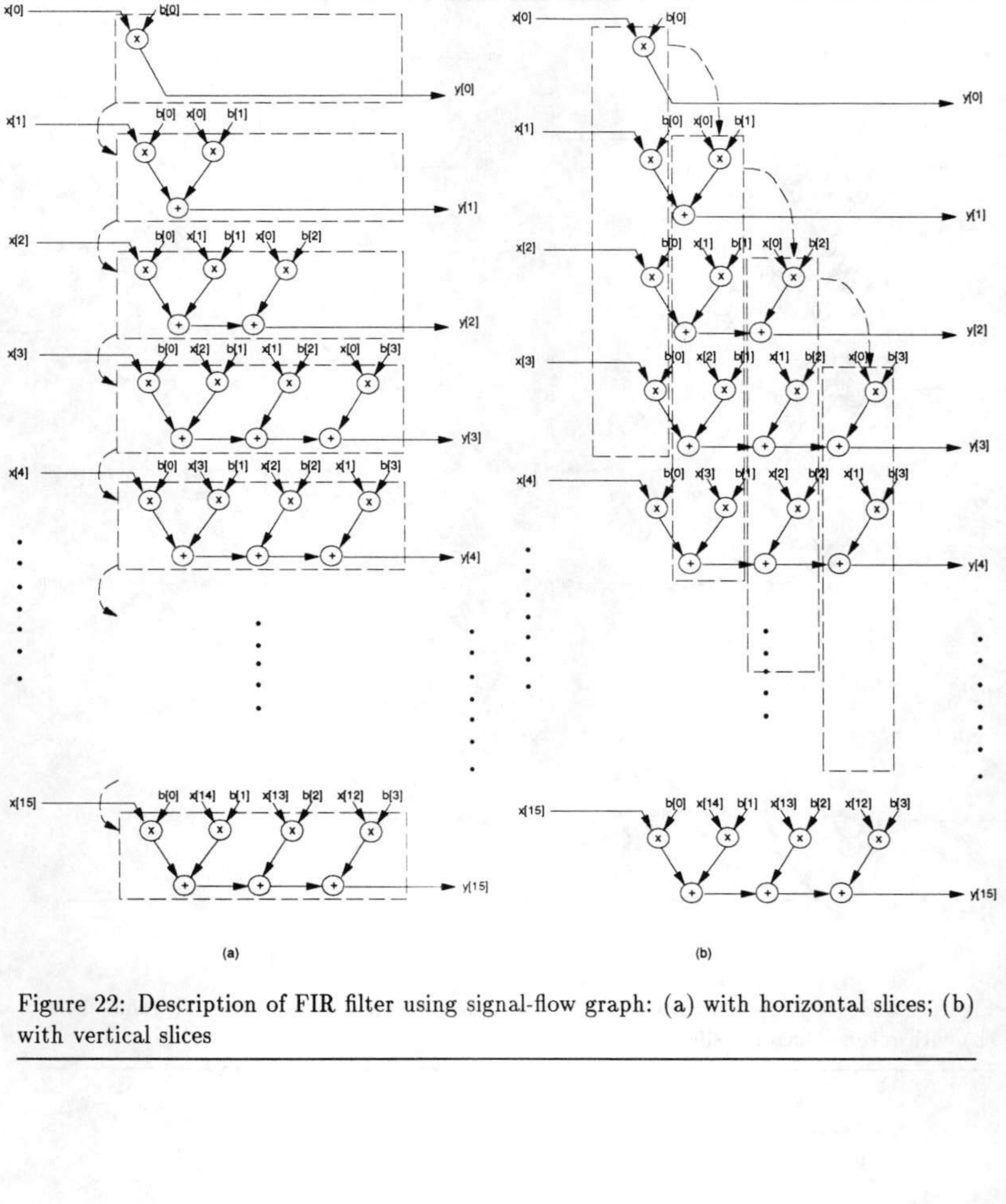


Figure 22: Description of FIR filter using signal-flow graph: (a) with horizontal slices; (b) with vertical slices

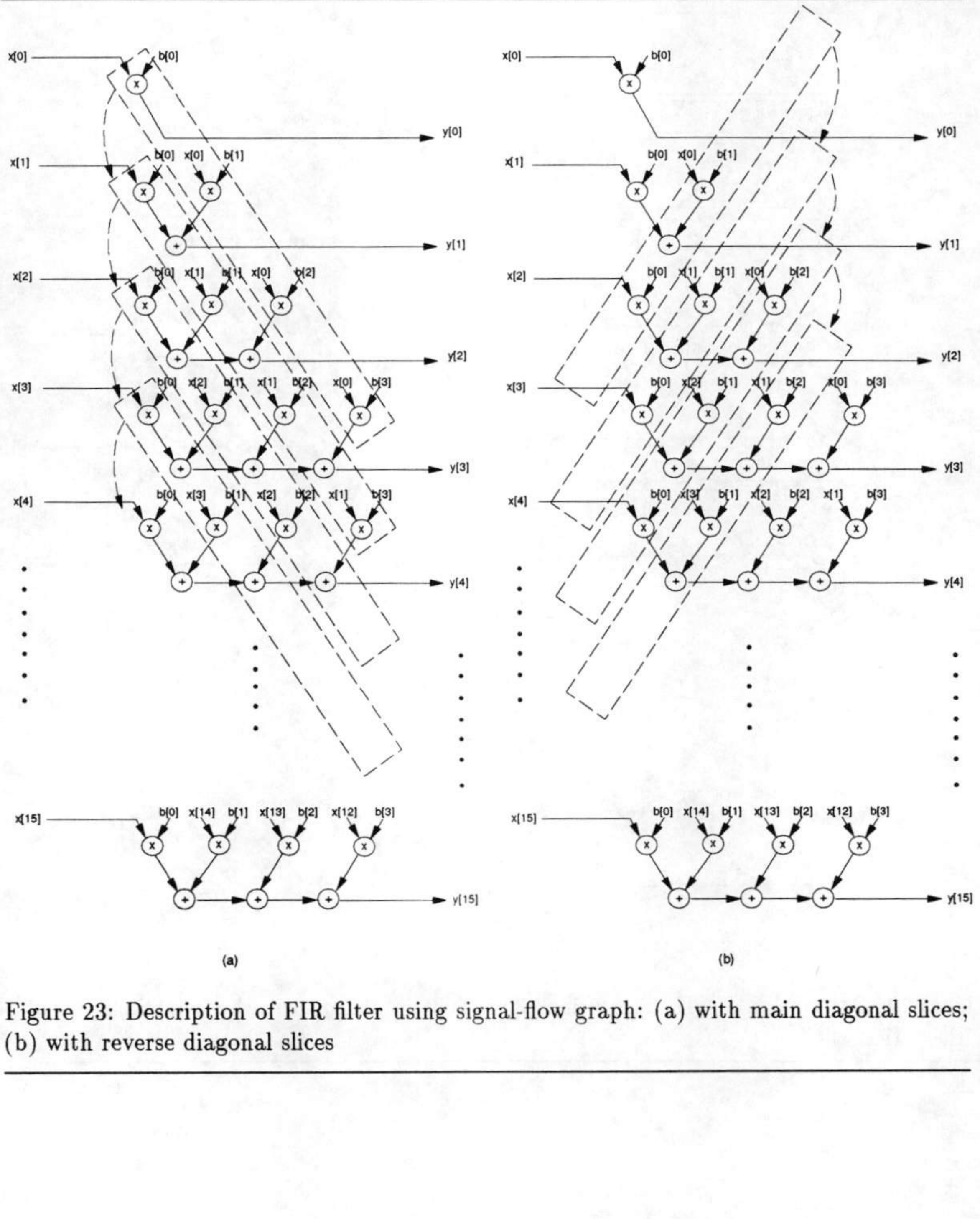

x[0]
b[0]
y[0]
x[1]
b[0] x[0] b[1]
y[1]
x[2]
b[0] x[1] b[1] x[0] b[2]
y[2]
x[3]
b[0] x[2] b[1] x[1] b[2] x[0] b[3]
y[3]
x[4]
b[0] x[3] b[1] x[2] b[2] x[1] b[3]
y[4]
x[15]
b[0] x[14] b[1] x[13] b[2] x[12] b[3]
y[15]
(a)
(b)

Figure 23: Description of FIR filter using signal-flow graph: (a) with main diagonal slices; (b) with reverse diagonal slices

```
process
 variable R0, R1, R2, DR: register_type;
 variable a, b, c, d, e, f: register_type;
begin
 R0 := 0;
 R1 := 0;
 R2 := 0;
 i := 0;
 for i in 0 to 15 loop
   DR := x[i];
   a := b(0)*DR;
   b := b(1)*R0;
   c := b(2)*R1;
   d := b(3)*R2;
   e := a+b;
   f := c+d;
   y := e+f;
   R2 := R1;
   R1 := R0;
   R0 := DR;
 end loop;
end;
```

Figure 24: Description of FIR filter using VHDL

							TIMING (ns)	
PS	SCOND	NS	ORDER	CV	ASCOND	ACTIONS	ACTION	STATE
1	(i==15)	END	0	T	T	R0=0	10	10
			0	T	T	R1=0	10	
	!(i==15)	2	0	T	T	R2=0	10	
			0	T	T	i=0	10	
2	T	3	0	T	T	DR=x[i]	100	300
			1	T	T	a=b[0]*DR	100	
			0	T	T	b=b[1]*R0	100	
			0	T	T	c=b[2]*R1	100	
			0	T	T	d=b[3]*R2	100	
			2	T	T	e=a+b	50	
			1	T	T	f=c+d	50	
			3	T	T	y=e+f	50	
3	(i==15)	END	0	T	T	R2=R1	10	50
			1	T	T	R1=R0	10	
	!(i==15)	2	2	T	T	R0=DR	10	
			0	T	T	i=i+1	50	

clock period: 300 ns
total execution time: 10,200 ns

Figure 25: Description of FIR filter using state-action table

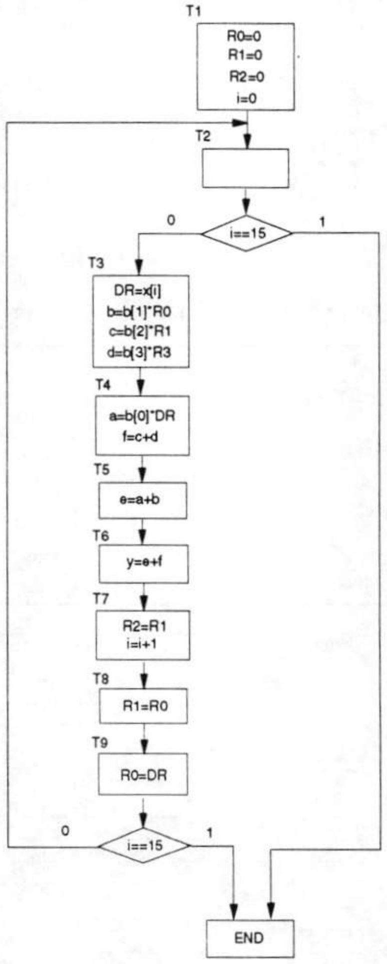


Figure 26: Description of FIR filter using ASM chart

4.5 Example 5: Computer System

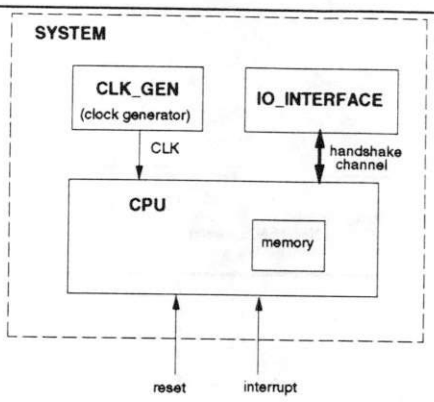


Figure 27: Block diagram of an example computer system

Figure 27 shows the block diagram of an example computer system. In Figure 28, a partial SpecCharts of the system is shown. At the topmost level, the specification consists of the state SYSTEM, which consists of three concurrent states: CLK_GEN, IO_INTERFACE, and CPU. CPU in turn contains two sequential states: RESET and NORMAL, whereas NORMAL contains three sequential states represents the instruction fetch, execution, and interrupt handling respectively. An 8K memory is declared as a variable array in CPU.

On the other hand, this system can also be described using VHDL (Figure 29). The top-level entity SYSTEM consists of three concurrent processes: CLK_GEN, IO_INTERFACE, and CPU. The CLK_GEN process is identical to the CLK_GEN state in the SpecCharts description since the CLK_GEN state is a leaf behavior and contains VHDL code only. Since VHDL can represent only one level of concurrency and sequentiality, the hierarchy of sub-behaviors RESET, FETCH, EXECUTE, and INTERRUPT in the SpecCharts descriptions are all flattened in VHDL description. This results in a less readable description.

5 Conclusion

In the previous sections, we have presented six description styles and examined some of their features. Figure 8 summarizes the various features of the descriptions we have discussed. We also described five examples using different description styles.

The timing diagram is an unambiguous canonical form which can be used to describe

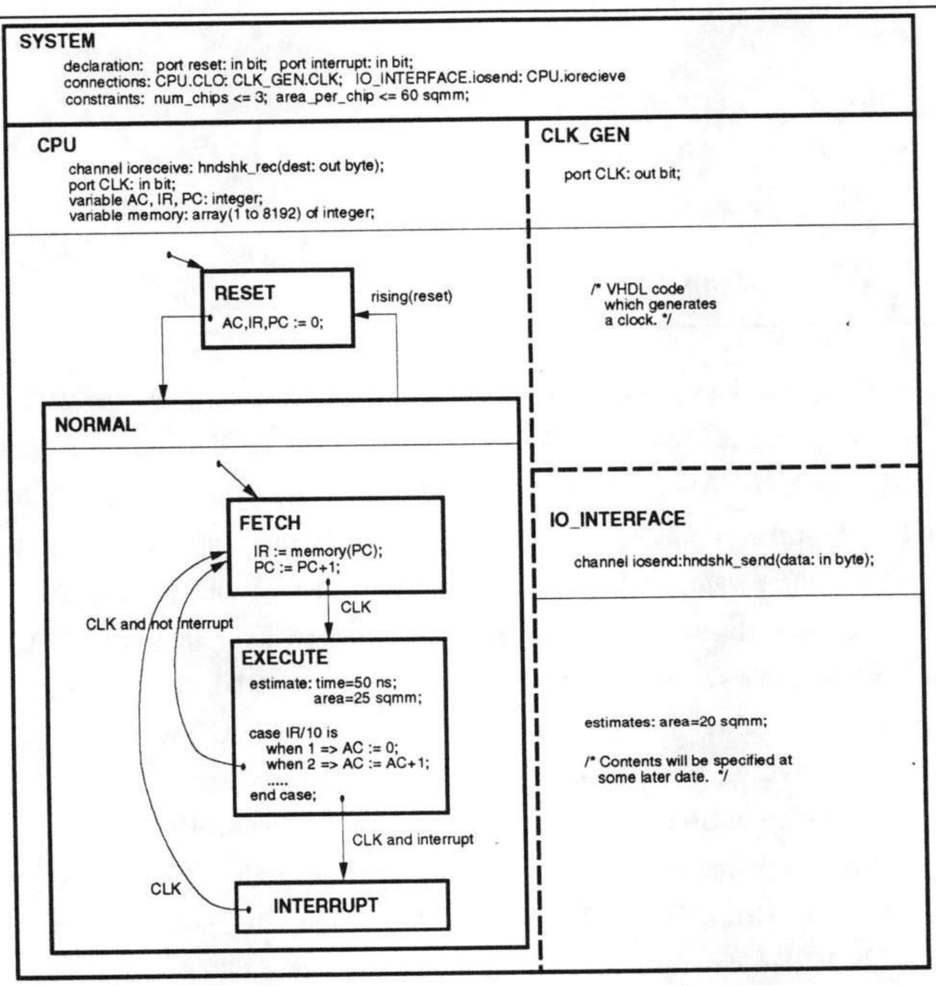

SYSTEM
declaration: port reset: in bit; port interrupt: in bit;
connections: CPU.CLO: CLK_GEN.CLK; IO_INTERFACE.iosend: CPU.iorecieve
constraints: num_chips <= 3; area_per_chip <= 60 sqmm;
CPU
channel ioreceive: hndshk_rec(dest: out byte);
port CLK: in bit;
variable AC, IR, PC: integer;
variable memory: array(1 to 8192) of integer;
RESET
AC,IR,PC := 0;
rising(reset)
NORMAL
FETCH
IR := memory(PC);
PC := PC+1;
CLK
CLK and not interrupt
EXECUTE
estimate: time=50 ns;
area=25 sqmm;
case IR/10 is
when 1 => AC := 0;
when 2 => AC := AC+1;
.....
end case;
CLK and interrupt
CLK
INTERRUPT
CLK_GEN
port CLK: out bit;
/* VHDL code which generates a clock. */
IO_INTERFACE
channel iosend:hndshk_send(data: in byte);
estimates: area=20 sqmm;
/* Contents will be specified at some later date. */

Figure 28: A partial SpecCharts description of the example computer system

```
entity SYSTEM is
  port (reset: in bit;
       interrupt: in bit);
end SYSTEM;

architecture A of SYSTEM is
  signal CLK: bit;
begin

CLK_GEN: process
      .....
      — clock generation
      .....
end process;

IO_INTERFACE: process
      .....
      .....
      .....
end process;

CPU: process(reset, interrupt, CLK)
  variable state: integer:=0;
  variable AC, IR, PC: integer;
  variable memory: array(1 to 8192) of integer;
begin
  if reset='1' and not reset'stable then
    AC := 0;
    IR := 0;
    PC := 0;
  elsif CLK='1' and interrupt='0' and then
    if state=0 then
      IR := memory(PC);
      PC := PC+1;
      state := 1;
    elsif state=1 then
      case IR/10 is
        when 1 => AC := 0;
        when 2 => AC := AC+1;
          ........
      end case;
      state := 0;
  elsif CLK='1' and interrupt='1' then
    .....
    .....
  end if;
end process;
```

Figure 29: A partial VHDL description of the example system

an interface without implying any implementation. It is well-suited in the initial phases of design when the implementation details are not known. For implementation purpose, the timing diagram becomes insufficient since the designer cannot use it to specify any implementation method. For instance, to implement the interface using FSMs, the designer has to describe the design using state-based description styles such as the ASM chart or the state-action table to specify the control steps. The ASM chart and the state-action table are useful in describing small FSM-based designs because they have the notion of states built into them.

The signal-flow graph shows the data operations and data flow graphically, but it cannot specify any control constructs. The FIR filter example indicates that it can help describing and scheduling DSP applications because they have a large amount of data computations but very little control flow is needed.

VHDL provides programming constructs, and can succinctly capture behavior that contains assignment statements with complex data transformations. On the other hand, it needs to be scheduled into control steps before the implementation can be obtained because it does not have built-in state specifications.

Hierarchy becomes vital when the design complexity increases. The SpecChart language, designed for system-level description, provides hierarchy as well as programming constructs and state-based specifications.

Obviously, one description style cannot fit all possible design styles. Hence, a framework is needed to enable designers to describe designs in a variety of styles, each suitable for a particular abstraction or model of design.

6 Acknowledgements

This work was partially supported by the Semiconducter Research Corporation (grant # 92-DJ-316), and we gratefully acknowledge their support. We also extend our gratitude to Viraphol Chaiyakul, Tedd Hadley, Frank Vahid and Sanjiv Narayan for their insightful comments regarding state-action tables and SpecCharts language. Furthermore, we would like to thank Smita Bakshi for her valuable assistance in designing the Finite-Impulse-Response filter example.

References

[BaGa93] Bakshi, S., and Gajski, D. D., "A Strategy for Design State Exploration," Info. & Computer Science Dept., UCI, Tech. Rep. 93-10, March 1993.

[Clar73] Clare, C. R., *Designing Logic Systems using State Machines,* McGraw-Hill Inc., 1973.

[HaCG93] Hadley, T., Chaiyakul, V., and Gajski, D. D., "A Data Structure for Interactive Synthesis," Info. & Computer Science Dept., UCI, Tech. Rep. 93-6, January 1993.

[Inte91] *Microprocessors*, Intel Corporation, 1991.

[IEEE88] *IEEE Standard VHDL Language Reference Manual*, 1988.

[NaVG91] Narayan, S., Vahid, F., and Gajski, D. D., "System Specification and Synthesis with the SpecCharts Language," in *Proc. of the International Conference on Computer-Aided Design,* 1991.

[VaNG91] Vahid, F., Narayan, S., and Gajski, D. D., "SpecCharts: A Language for System Level Synthesis," in *Proc. of the International Symposium on Computer Hardware Description Languages and their Applications,* 1991.