Essential Issues in Codesign

Daniel D. Gajski
Jianwen Zhu
Rainer Dömer

Technical Report ICS-97-26
June, 1997

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(714) 824-8059

gajski@ics.uci.edu
jzhu@ics.uci.edu
doemer@ics.uci.edu

Abstract

In this report we discuss the main models of computation, the basic types of architectures, and language features needed to specify systems. We also give an overview of a generic methodology for designing systems, that include software and hardware parts, from executable specifications.

Contents

List of Figures

Essential Issues in Codesign

Daniel D. Gajski, Jianwen Zhu, Rainer Dömer

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

Abstract

In this report we discuss the main models of computation, the basic types of architectures, and language features needed to specify systems. We also give an overview of a generic methodology for designing systems, that include software and hardware parts, from executable specifications.

1 Models

In the last ten years, VLSI design technology, and the CAD industry in particular, have been very successful, enjoying an exceptional growth that has been paralleled only by the advances in IC fabrication. Since the design problems at the lower levels of abstraction became humanly intractable earlier than those at higher abstraction levels, researchers and the industry alike were forced to devote their attention first to lower level problems such as circuit simulation, placement, routing and floorplanning. As these problems became more manageable, CAD tools for logic simulation and synthesis were developed successfully and introduced into the design process. As design complexities have grown and time-to-market requirements have shrunk drastically, both industry and academia have begun to focus on system levels of design since they reduce the number of objects that a designer needs to consider by an order of magnitude and thus allow the design and manufacturing of complex application specific integrated circuits (ASICs) in short periods of time.

The first step in designing a system is specifying its functionality. To help us understand and organize this functionality in a systematic manner, we can use a variety of conceptual models. In this chapter, we will survey the key conceptual models that are most commonly used for hardware and for software systems, while in the next section we examine the various architectures that are used in implementing those models, while in the third section we survey language features needed for specifying systems. In the fourth section we present a generic codesign methodology.

1.1 Model and architecture definition

System design is the process of implementing a desired functionality using a set of physical components. Clearly, then, the whole process of system design must begin with specifying the desired functionality. This is not, however, an easy task. For example, consider the task of specifying an elevator controller. How do we describe its functionality in sufficient detail that we could predict with absolute precision what the elevator's position would be after any sequence of pressed buttons? The problem with natural-language specifications is that they are often ambiguous and incomplete, lacking the capacity for detail that is required by such a task. Therefore, we need a more precise approach to specify functionality.

The most common way to achieve the level of precision we need is to think of the system as a collection of simpler subsystems, or pieces, and the method or the rules for composing these pieces to create system functionality. We call such a method a **model**.

To be useful, a model should possess certain qualities. First, it should be formal so that it contains no ambiguity. It should also be complete, so that it can describe the entire system. In addition, it should be comprehensible to the designers who need to use it, as well as being easy to modify, since it is inevitable that, at some point, they will wish to change the system's functionality. Finally, a model should be natural enough to aid, rather than impede, the designer's understanding of the system.

It is important to note that a model is a formal system consisting of objects and composition rules, and is used for describing a system's characteristics. Typically, we would use a particular model to decompose a system into pieces, and then generate a specification by describing these pieces in a particular language. A language can capture many different models, and a model can be captured in many different languages.

The purpose of a conceptual model is to provide an abstracted view of a system. Figure 1, for example, shows two different models of an elevator controller, whose English description is in Figure 1(a). The difference between these two models is that Figure 1(b) represents the controller as a set of programming statements, whereas Figure 1(c) represents the controller as a finite state machine in which the states indicate the direction of the elevator movement.

As you can see, each of these models represents a set of objects and the interactions among them. The state-machine model, for example, consists of a set of states and transitions between these states; the programming model, in contrast, consists of a set of statements that are executed under a control sequence that uses branching and looping. The advantage to having these different models at our disposal is that they allow designers to represent different views of a system, thereby exposing its different characteristics. For example, the state-machine model is best suited to represent a system's temporal behavior, as it allows a designer to explicitly express the modes and mode-transitions caused by external or internal events. The algorithmic model, on the other hand, has no explicit states. However, since it can specify a system's input-output relation in terms of a sequence of statements, it is well-suited to representing the procedural view of the system.

Designers choose different models in different phases of the design process, in order to emphasize those aspects of the system that are of interest to them at that particular time. For example, in the specification phase, the designer knows nothing beyond the functionality of the system, so he will tend to use a model that does not reflect any implementation information. In the implementation phase, however, when information about the system's components is available, the designer will switch to a model that can capture the system's structure.

Different models are also required for different application domains. For example, designers would model real-time systems and database systems differently, since the former focus on temporal behavior, while the latter focus on data organization.

Once the designer has found an appropriate model to specify the functionality of a system, he can describe in detail exactly how that system will work. At that point, however, the design process is not complete, since such a model has still not described exactly how that system is to be manufactured. The

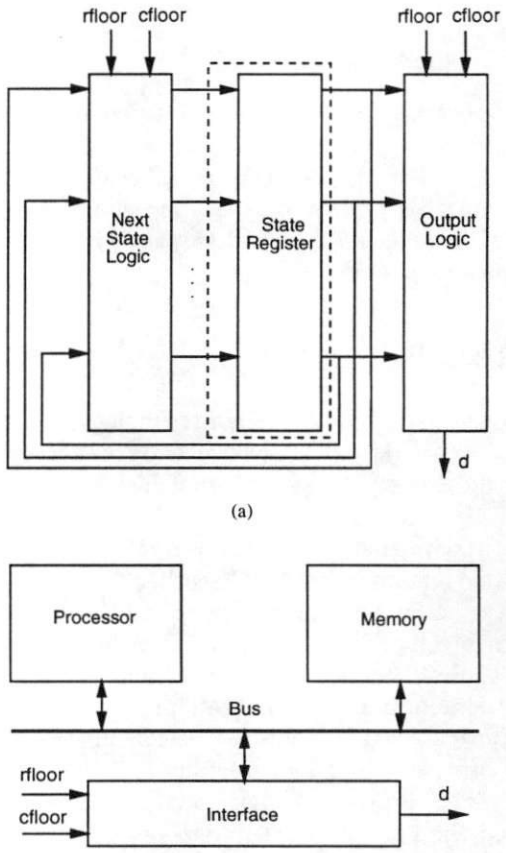


Figure 2: Architectures used in: (a) a register-level implementation, (b) a system-level implementation. (‡)

"If the elevator is stationary and the floor requested is equal to the current floor, then the elevator remains idle.

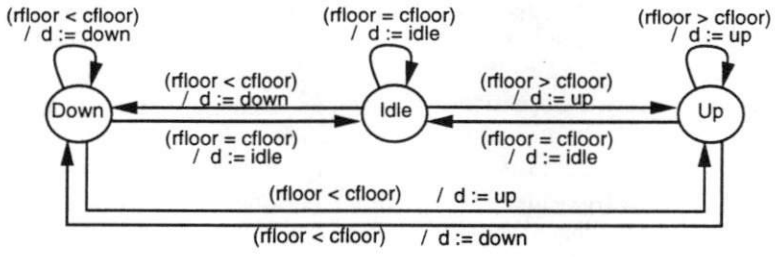
If the elevator is stationary and the floor requested is less than the current floor, then lower the elevator to the requested floor.

If the elevator is stationary and the floor requested is greater than the current floor, then raise the elevator to the requested floor."

(a)

```
loop
    if (rfloor = cfloor) then
        d := idle;
    elsif (rfloor < cfloor) then
        d := down;
    elsif (rfloor > cfloor) then
        d := up;
    end if;
end loop;
```

(b)


(rfloor < cfloor) / d := down
(rfloor = cfloor) / d := idle
(rfloor > cfloor) / d := up
(rfloor < cfloor) / d := down
(rfloor > cfloor) / d := up
Down
Idle
Up
(rfloor = cfloor) / d := idle
(rfloor = cfloor) / d := idle
(rfloor < cfloor) / d := up
(rfloor < cfloor) / d := down

(c)

Figure 1: Conceptual views of an elevator controller: (a) desired functionality in English, (b) programming model, (c) state-machine model. (†)

next step, then, is to transform the model into an **architecture**, which serves to define the model's implementation by specifying the number and types of components as well as the connections between them. In Figure 2, for example, we see two different architectures, either of which could be used to implement the state-machine model of the elevator controller in Figure 1(c). The architecture in Figure 2(a) is a register-level implementation, which uses a state register to hold the current state and the combinational logic to implement state transitions and values of output signals. In Figure 2(b), we see a processor-level implementation that maps the same state-machine model into software, using a variable in a program to represent the current state and statements in the program to calculate state transitions and values of output signals. In this architecture, the program is stored in the memory and executed by the processor.

Models and architectures are conceptual and implementation views on the highest level of abstraction. Models describe how a system works, while architectures describe how it will be manufactured. The **design process** or **methodology** is the set of design tasks that transform a model into an architecture. At the beginning of this process, only the system's functionality is known. The designer's job, then, is to describe this functionality in some language which is based on the most appropriate models. As the design process proceeds, an architecture will begin to emerge, with more detail being added at each step in the process. Generally, designers will find that certain architectures are more efficient in implementing certain models. In addition, design and manufacturing technology will have a great influence on the choice of an architecture. Therefore, designers have to consider many different implementation alternatives before the design process is complete.

1.2 Model taxonomy

System designers use many different models in their various hardware or software design methodologies. In general, though, these models fall into five distinct categories: (1) state-oriented; (2) activity-oriented; (3) structure-oriented; (4) data-oriented; and (5) heterogeneous. A **state-oriented model**, such as a finite-state machine, is one that represents the system as a set of states and a set of transitions between them, which are triggered by external events. State-oriented models are most suitable for control systems, such as real-time reactive systems, where the system's temporal behavior is the most important aspect of the design. An **activity-oriented model**, such as a dataflow graph, is one that describes a system as a set of activities related by data or execution dependencies. This model is most applicable to transformational systems, such as digital signal processing systems, where data passes through a set of transformations at a fixed rate. Using a **structure-oriented model**, such as a block diagram, we would describe a system's physical modules and interconnections between them. Unlike state-oriented and activity-oriented models which primarily reflect a system's functionalities, the structure-oriented models focus mainly on the system's physical composition. Alternatively, we can use a **data-oriented model**, such as an entity-relationship diagram, when we need to represent the system as a collection of data related by their attributes, class membership and interactions. This model is most suitable for information systems, such as databases, where the function of the system is less important than the data organization of the system. Finally, a designer could use a **heterogeneous model** – one that integrates many of the characteristics of the previous four models – whenever he needs to represent a variety of different views in a complex system.

In the rest of this section we will describe some frequently used models.

1.3 Finite-state machine

A **finite-state machine** (FSM) is an example of a state-oriented model. It is the most popular model for describing control systems, since the temporal behavior of such systems is most naturally represented in the form of states and transitions between states.

Basically, the FSM model consists of a set of **states**, a set of **transitions** between states, and a set of **actions** associated with these states or transitions.

The finite state machine can be defined abstractly as the quintuple

$$< S, I, O, f, h >$$

where S, I, and O represent a set of states, set of inputs and a set of outputs, respectively, and f and h represent the next-state and the output functions. The next state function f is defined abstractly as a mapping $S \times I \rightarrow S$. In other words, f assigns to every pair of state and input symbols another state symbol. The FSM model assumes that transitions from one state to another occur only when input symbols change. Therefore, the next-state function f defines what the state of the FSM will be after the input symbols change.

The output function h determines the output values in the present state. There are two different types of finite state machine which correspond to two different definitions of the output function h. One type is a **state-based** or **Moore-type**, for which h is defined as a mapping $S \rightarrow O$. In other words, an output symbol is assigned to each state of the FSM and outputed during the time the FSM is in that particular state. The other type is an **input-based** or **Mealy-type** FSM, for which h is defined as the mapping $S \times I \rightarrow O$. In this case, an output symbol in each state is defined by a pair of state and input symbols and it is outputed while the state and the corresponding input symbols persist.

According to our definition, each set S, I, and O may have any number of symbols. However, in reality we deal only with binary variables, operators and memory elements. Therefore, S, I, and O must be implemented as a cross-product of binary signals or memory elements, whereas functions f and h are defined by Boolean expressions that will be implemented with logic gates.

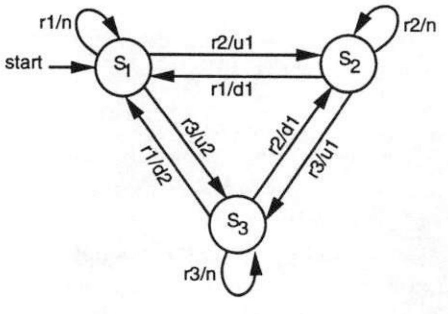


Figure 3: FSM model for the elevator controller. (†)

In Figure 3, we see an input-based FSM that models the elevator controller in a building with three floors, as described in Section 1.1. In this model, the set of inputs $I = \{r1, r2, r3\}$ represents the floor requested. For example, $r2$ means that floor 2 is requested. The set of outputs $O = \{d2, d1, n, u1, u2\}$ represents the direction and number of floors the elevator should go. For example, $d2$ means that the elevator should go down 2 floors, $u2$ means that the elevator should go up 2 floors, and n means that the elevator should stay idle. The set of states represents the floors. In Figure 3, we can see that if the current floor is 2 (i.e., the current state is S_2), and floor 1 is requested, then the output will be $d1$.

In Figure 4 we see the state-based model for the same elevator controller, in which the value of the output is indicated in each state. Each state has been split into three states representing each of the output signals that the state machine in Figure 3 will output when entering that particular state.

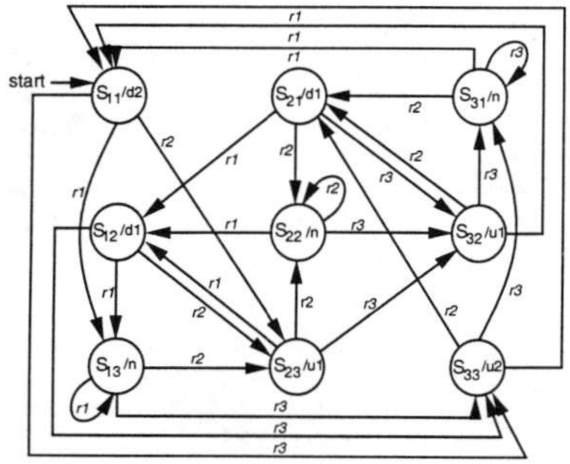


Figure 4: State-based FSM model for the elevator controller. (†)

In practical terms, the primary difference between these two models is that the state-based FSM may require quite a few more states than the input-based model. This is because in a input-based model, there may be multiple arcs pointing to a single state, each arc having a different output value; in the state-based model, however, each different output value would require its own state, as is the case in Figure 4.

1.4 Finite-state machine with datapath

In cases when a FSM must represent integer or floating-point numbers, we could encounter a state-explosion problem, since, if each possible value for a number requires its own state, then the FSM could require an enormous number of states. For example, a 16-bit integer can represent 2^{16} or 65536 different states. There is a fairly simple way to eliminate the state-explosion problem, however, as it is possible to extend a FSM with integer and floating-point variables, so that each variable replaces thousands of states. The introduction of a 16-bit variable, for example, would reduce the number of states in the FSM model by 65536.

In order to formally define a FSMD[Gaj97], we must extend the definition of a FSM introduced in the previous section by introducing sets of datapath variables, inputs and outputs that will complement the sets of FSM states, inputs and outputs.

As we mentioned in the previous section an FSM is a quintuple

$$< S, I, O, f, h >$$

Each state, input and output symbols are defined by a cross-product of Boolean variables. More precisely,

$$I = A_1 \times A_2 \times \ldots A_k$$
$$S = Q_1 \times Q_2 \times \ldots Q_m$$
$$O = Y_1 \times Y_2 \times \ldots Y_n$$

where $A_i, 1 \leq i \leq k$, is an input signal, $Q_i, 1 \leq i \leq m$ is the flip-flop output and $Y_i, 1 \leq i \leq n$ is an output signal.

In order to include a datapath, we must extend the above FSM definition by adding the set of datapath variables, inputs and outputs. More formally, we define a variables set

$$V = V_1 \times V_2 \times \ldots V_q$$

which defines the state of the datapath by defining the values of all variables in each state.

In the same fashion, we can separate the set of FSMD inputs into a set of FSM inputs I_C and a set of datapath inputs I_D. Thus,

$$I = I_C \times I_D$$

where $I_C = A_1 \times A_2 \times \ldots A_k$ as before and $I_D = B_1 \times B_2 \times \ldots B_p$.

Similarly, the output set consists of FSM outputs O_C and datapath outputs O_D. In other words,

$$O = O_C \times O_D$$

where $O_C = Y_1 \times Y_2 \ldots Y_n$ as before and $O_D = Z_1 \times Z_2 \times \ldots Z_r$. However, note that A_i, Q_j and Y_k represent Boolean variables while B_i, V_i and Z_i are Boolean vectors which in turn represent integers, floating-point numbers and characters. For example, in a 16-bit datapath, B_i, V_i and Z_i would be 16 bits wide and if they were positive integers, they would be able to assume values from 0 to 2^{16-1}.

Except for very trivial cases, the size of the datapath variables, and ports makes specification of functions f and h in a tabular form very difficult. In order to be able to specify variable values in an efficient and understandable way in the definition of an FSMD, we will specify variable values with arithmetic expressions.

We define the set of all possible expressions, *Expr*, over the set of variables V, to be the set of all constants K of the same type as variables in V, the set of variables V itself and all the expressions obtained by combining two expressions with arithmetic, logic, or rearrangement operators.

More formally,

$$Expr(V) = K \cup V \cup \{(e_i \Box e_j) \mid e_i, e_j \in Expr, \Box \textit{ is an acceptable operator}\}$$

Using $Expr(V)$ we can define the values of the status signals as well as transformations in the datapath. Let $STAT = \{stat_k = e_i \Delta e_j \mid e_i, e_j, \in Expr(V), \Delta \in \{\leq, <, =, \neq, >, \geq\}\}$ be the set of all status signals which are described as relations between variables or expressions of variables. Examples of status signals are $Data \neq 0, (a - b) > (x + y)$ and $(counter = 0) AND (x > 10)$. The relations defining status signals are either true, in which case the status signal has value 1 or false in which case it has value 0.

With formal definition of expressions and relations over a set of variables we can simplify function $f : (S \times V) \times I \rightarrow S \times V$ by separating it into two parts: f_C and f_D.. The function f_C defines the next state of the control unit

$$f_C : S \times I_C \times STAT \rightarrow S$$

while the function f_D defines the values of datapath variables in the next state

$$f_D : S \times V \times I_D \rightarrow V$$

In other words, for each state $s_i \in S$ we compute a new value for each variable $V_j \in V$ in the datapath by evaluating an expression $e_j \in Expr(V)$. Thus, the function f_D is represented by a set of simpler functions, in which each function in the set defines variable values for the state s_i

$$f_D := \{f_{Di} : V \times I_D \rightarrow V : \{V_j = e_j | V_j \in V, e_j \in Expr(V \times I_D)\}\}$$

In other words, function f_D is decomposed into a set of functions f_{Di}, where each f_{Di} assigns one expression e_k to each variable V_j in the datapath in state s_i. Therefore, new values for all variables in the datapath are computed by evaluating expressions e_j, for all j such that $1 \leq j \leq q$.

Similarly, we can decompose the output function $h : S \times V \times I \rightarrow O$ into two different functions, h_C and h_D where h_C defines the external control outputs O_C as in the definition of an FSM and h_D defines external datapath outputs.

Therefore,

$$h_C : S \times I_C \times STAT \rightarrow O_C$$

and

$$h_D : S \times V \times I_D \rightarrow O_D$$

Note, again that variables in O_C are Boolean variable and that variables in O_D are Boolean vectors.

Using this kind of FSMD, we could model the elevator controller example in Figure 3 with only one state, as shown in Figure 5. This reduction in the number of states is possible because we have designated a variable $cfloor$ to store the state value of the FSM in Figure 3 and $rfloor$ to store the values of $r1$, $r2$ and $r3$.

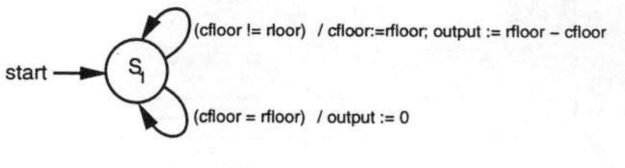

(cfloor != rloor) / cfloor:=rfloor; output := rfloor – cfloor
start
S1
(cfloor = rfloor) / output := 0

Figure 5: FSMD model for the elevator controller. (†)

In general, the FSM is suitable for modeling control-dominated systems, while the FSMD can be suitable for both control- and computation-dominated systems. However, it should be pointed out that neither the FSM nor the FSMD model is suitable for complex systems, since neither one explicitly supports concurrency and hierarchy. Without explicit support for concurrency, a complex system will precipitate an explosion in the number of states. Consider, for example, a system consisting of two concurrent subsystems, each with 100 possible states. If we try to represent this system as a single FSM or FSMD, we must represent all possible states of the system, of which there are $100 \times 100 = 10,000$. At the same time, the lack of hierarchy would cause an increase in the number of arcs. For example, if there are 100 states, each requiring its own arc to transition to a specific state for a particular input value, we would need 100 arcs, as opposed to the single arc required by a model that can hierarchically group those 100 states into one state. The problem with such models, of course, is that once they reach several hundred states or arcs, they become incomprehensible to humans.

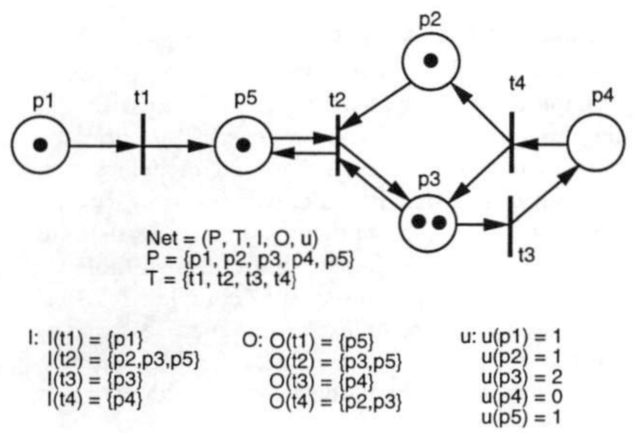
1.5 Petri net

The **Petri net** model [Pet81, Rei92] is another type of state-oriented model, specifically defined to model systems that comprise interacting concurrent tasks. The Petri net model consists of a set of **places**, a set of **transitions**, and a set of **tokens**. Tokens reside in places, and circulate through the Petri net by being consumed and produced whenever a transition fires.

More formally, a Petri net is a quintuple

$$< P, T, I, O, u > \qquad (1)$$

where $P = \{p_1, p_2, \ldots, p_m\}$ is a set of places, $T = \{t_1, t_2, \ldots, t_n\}$ is a set of transitions, and P and T are disjoint. Further, the input function, $I : T \rightarrow P^+$, defines all the places providing input to a transition, while the output function, $O : T \rightarrow P^+$, defines all the output places for each transition. In other words, the input and output functions specify the connectivity of places and transitions. Finally, the marking function $u : P \rightarrow N$ defines the number of tokens in each place, where N is the set of non-negative integers.



Net = (P, T, I, O, u)
P = {p1, p2, p3, p4, p5}
T = {t1, t2, t3, t4}

I: I(t1) = {p1}	O: O(t1) = {p5}	u: u(p1) = 1
I(t2) = {p2,p3,p5}	O(t2) = {p3,p5}	u(p2) = 1
I(t3) = {p3}	O(t3) = {p4}	u(p3) = 2
I(t4) = {p4}	O(t4) = {p2,p3}	u(p4) = 0
		u(p5) = 1

Figure 6: A Petri net example. (†)

In Figure 6, we see a graphic and a textual representation of a Petri net. Note that there are five places (graphically represented as circles) and four transitions (graphically represented as solid bars) in this Petri net. In this instance, the places $p2$, $p3$, and $p5$ provide inputs to transition $t2$, and $p3$ and $p5$ are the output places of $t2$. The marking function u assigns one token to $p1$, $p2$ and $p5$ and two tokens to $p3$, as denoted by $u(p1, p2, p3, p4, p5) = (1, 1, 2, 0, 1)$.

As mentioned above, a Petri net executes by means of firing transitions. A transition can **fire** only if it is enabled – that is, if each of its input places has at least one token. A transition is said to have fired when it has removed all of its enabling tokens from its input places, and then deposited one token into each output place. In Figure 6, for example, after transition t_2 fires, the marking u will change to $(1, 0, 2, 0, 1)$.

Petri nets are useful because they can effectively model a variety of system characteristics. Figure 7(a), for example, shows the modeling of *sequencing*, in which transition $t1$ fires after transition $t2$. In Figure 7(b), we see the modeling of *non-deterministic branching*, in which two transitions are enabled but only one of them can fire. In Figure 7(c), we see the modeling of *synchronization*, in which a transition can fire only after both input places have tokens. Figure 7(d) shows how one would model *resource contention*, in which two transitions compete for the same token which resides in the place in the center. In Figure 7(e), we see how we could model *concurrency*, in which two transitions, $t2$ and $t3$, can fire simultaneously. More precisely, Figure 7(e) models two concurrent processes, a producer and a consumer; the token located in the place at the center is produced by $t2$ and consumed by $t3$.

Petri net models can be used to check and validate certain useful system properties such as safeness and liveness. **Safeness**, for example, is the property of Petri nets that guarantees that the number of tokens in the net will not grow indefinitely. In fact, we cannot construct a Petri net in which the number of tokens is unbounded. **Liveness**, on the other hand, is the property of Petri nets that guarantees a dead-lock free operation, by ensuring that there is always at least one transition that can fire.

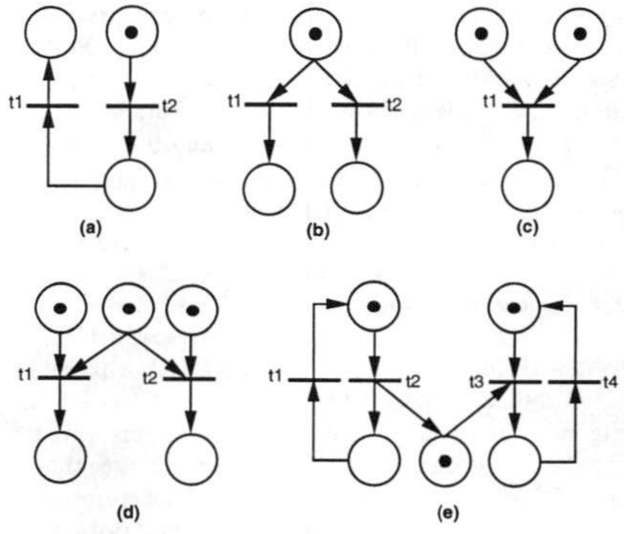


Figure 7: Petri net representing: (a) sequencing, (b) branching, (c) synchronization, (d) contention, (e) concurrency. (†)

Although a Petri net does have many advantages in modeling and analyzing concurrent systems, it also has limitations that are similar to those of an FSM:

it can quickly become incomprehensible with any increase in system complexity.

1.6 Hierarchical concurrent finite-state machine

The **hierarchical concurrent finite-state machine** (HCFSM) is essentially an extension of the FSM model, which adds support for **hierarchy** and **concurrency**, thus eliminating the potential for state and arc explosion that occurred when describing hierarchical and concurrent systems with FSM models.

Like the FSM, the HCFSM model consists of a set of **states** and a set of **transitions**. Unlike the FSM, however, in the HCFSM each state can be further decomposed into a set of **substates**, thus modeling hierarchy. Furthermore, each state can also be decomposed into **concurrent substates**, which execute in parallel and communicate through global variables. The transitions in this model can be either structured or unstructured, with structured transitions allowed only between two states on the same level of hierarchy, while unstructured transitions may occur between any two states regardless of their hierarchical relationship.

One language that is particularly well-adapted to the HCFSM model is Statecharts [Har87], since it can easily support the notions of hierarchy, concurrency and communication between concurrent states. Statecharts uses unstructured transitions and a broadcast communication mechanism, in which events emitted by any given state can be detected by all other states.

The Statecharts language is a graphic language. Specifically, we use rounded rectangles to denote states at any level, and encapsulation to express a hierarchical relation between these states. Dashed lines between states represent concurrency, and arrows denote the transitions between states, each arrow being labeled with an event and, optionally, with a parenthesized condition and/or action.

Figure 8 shows an example of a system represented by means of Statecharts. In this figure, we can see that state Y is decomposed into two concurrent states, A and D; the former consisting of two further substates, B and C, while the latter comprises substates E, F, and G. The bold dots in the figure indicate the starting points of states. According to the Statecharts language, when event b occurs while in state C, A will transfer to state B. If, on the other hand, event a occurs while in state B, A will transfer to state C, but only if condition P holds at the instant of occurrence. During the transfer from B to C, the action c associated with the transition will be performed.

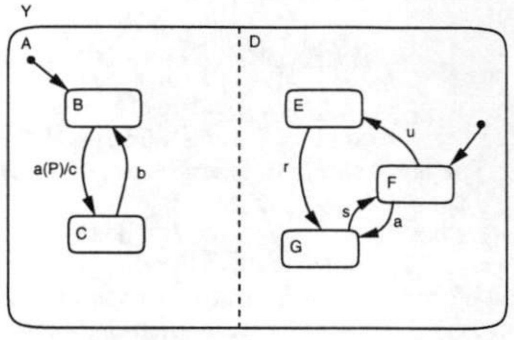


Figure 8: Statecharts: hierarchical concurrent states. (†)

Because of its hierarchy and concurrency constructs, the HCFSM model is well-suited to representing complex control systems. The problem with this model, however, is that, like any other state-oriented model, it concentrates exclusively on modeling control, which means that it can only associate very simple actions, such as assignments, with its transitions or states. As a result, the HCFSM is not suitable for modeling certain characteristics of complex systems, which may require complex data structures or may perform in each state an arbitrarily complex activity. For such systems, this model alone would probably not suffice.

1.7 Programming languages

Programming languages provide a heterogeneous model that can support data, activity and control modeling. Unlike the structure chart, programming languages are presented in a textual, rather than a graphic, form.

There are two major types of programming languages: imperative and declarative. The **imperative** class includes languages like C and Pascal, which use a control-driven model of execution, in which statements are executed in the order written in the program. LISP and PROLOG, by contrast, are examples of **declarative** languages, since they model execution through demand-driven or pattern-driven computation. The key difference here is that declarative languages specify no explicit order of execution, focusing instead on defining the target of the computation through a set of functions or logic rules.

In the aspect of data modeling, imperative programming languages provide a variety of data structures. These data structures include, for example, **basic** data types, such as integers and reals, as well as

composite types, like arrays and records. A programming language would model small activities by means of **statements**, and large activities by means of **functions** or **procedures**, which can also serve as a mechanism for supporting hierarchy within the system. These programming languages can also model control flow, by using control constructs that specify the order in which activities are to be performed. These control constructs can include **sequential** composition (often denoted by a semicolon), **branching** (*if* and *case* statements), **looping** (*while*, *for*, and *repeat*), as well as **subroutine calls**.

The advantage to using an imperative programming language is that this paradigm is well-suited to modeling computation-dominated behavior, in which some problem is solved by means of an algorithm, as, for example, in a case when we need to sort a set of numbers stored in an array.

The main problem with programming languages is that, although they are well-suited for modeling the data, activity, and control mechanism of a system, they do not explicitly model the system's states, which is a disadvantage in modeling embedded systems.

1.8 Program-state machine

A **program-state machine** (PSM) [GVN94] is an instance of a heterogeneous model that integrates an HCFSM with a programming language paradigm. This model basically consists of a hierarchy of **program-states**, in which each program-state represents a distinct mode of computation. At any given time, only a subset of program-states will be active, i.e., actively carrying out their computations.

Within its hierarchy, the model would consist of both composite and leaf program-states. A **composite** program-state is one that can be further decomposed into either **concurrent** or **sequential** program-substates. If they are concurrent, all the program-substates will be active whenever the program-state is active, whereas if they are sequential, the program-substates are only active one at a time when the program-state is active. A sequentially decomposed program-state will contain a set of transition arcs, which represent the sequencing between the program-substates. There are two types of transition arcs. The first, a **transition-on-completion arc (TOC)**, will be traversed only when the source program-substate has completed its computation and the associated arc condition evaluates to true. The second, a **transition-immediately arc (TI)**, will be traversed immediately whenever the arc condition becomes true, regardless of whether the source program-substate has completed its computation. Finally, at the bottom of the hierarchy, we have the **leaf** program-states whose computations are described through programming language statements.

When we are using the program-state machine as our model, the system as an entity can be graphically represented by a rectangular box, while the program-states within the entity will be represented by boxes with curved corners. A concurrent relation between program-substates is denoted by the dotted line between them. Transitions are represented with directed arrows. The starting state is indicated by a triangle, and the completion of individual program-states is indicated by a transition arc that points to the *completion point*, represented as a small square within the state. TOC arcs are those that originate from a square inside the source substate, while TI arcs originate from the perimeter of the source substate.

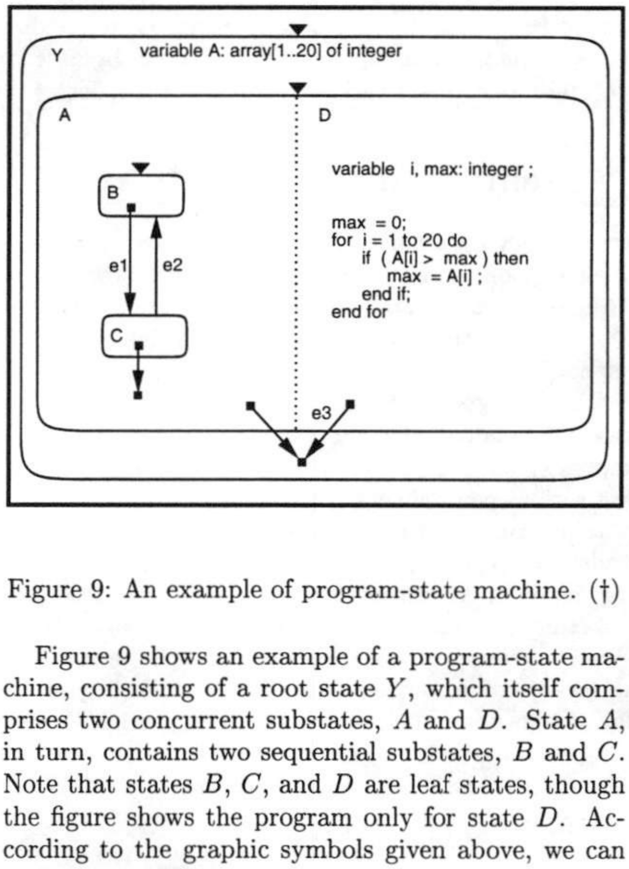


Figure 9: An example of program-state machine. (†)

Figure 9 shows an example of a program-state machine, consisting of a root state Y, which itself comprises two concurrent substates, A and D. State A, in turn, contains two sequential substates, B and C. Note that states B, C, and D are leaf states, though the figure shows the program only for state D. According to the graphic symbols given above, we can see that the arcs labeled $e1$ and $e3$ are TOC arcs, while the arc labeled $e2$ is a TI arc. The configuration of arcs would mean that when state B finishes and condition $e1$ is true, control will transfer to state C. If, however, condition $e2$ is true while in state C, control will transfer to state B regardless of whether

C finishes or not.

Since PSMs can represent a system's states, data, and activities in a single model, they are more suitable than HCFSMs for modeling systems which have complex data and activities associated with each state. A PSM can also overcome the primary limitation of programming languages, since it can model states explicitly. It allows a modeler to specify a system using hierarchical state-decomposition until he/she feels comfortable using program constructs. The programming language model and HCFSM model are just two extremes of the PSM model. A program can be viewed as a PSM with only one leaf state containing language constructs. A HCFSM can be viewed as a PSM with all its leaf states containing no language constructs.

In this section we presented the main models to capture systems. Obviously, there are more models used in codesign, mostly targeted at specific applications. For example, the codesign finite state machine (CFSM) model [CGH+93], which is based on communicating FSMs using event broadcasting, is targeted at small, reactive real-time systems and can be used to formally define and verify a system's properties.

2 Architectures

To this point, we have demonstrated how various models can be used to describe a system's functionality, data, control and structure. An architecture is intended to supplement these descriptive models by specifying how the system will actually be implemented. The goal of an architecture, then, is to describe the number of components, the type of each component, and the type of each connection among these various components in a system.

Architectures can range from simple controllers to parallel heterogeneous processors. Despite this variety, however, architectures nonetheless fall into a few distinct classes, namely, (1) **application-specific architectures**, such as DSP systems, (2) **general-purpose processors**, such as RISCs, and (3) **parallel processors**, such as VLIW, SIMD and MIMD machines.

2.1 Controller architecture

The simplest of the application-specific architectures is the **controller** variety, which is a straight-forward implementation of the finite-state machine model presented in Section 1.3 and defined by the quintuple $< S, I, O, f, h >$. A controller consists of a register and two combinational blocks, as shown in Figure 10. The register, usually called the *State register*, is designed to store the states in S, while the two combinational blocks, referred to as the *Next-state logic* and the *Output logic*, implement functions f and h. *Inputs* and *Outputs* are representations of Boolean signals that are defined by sets I and O.

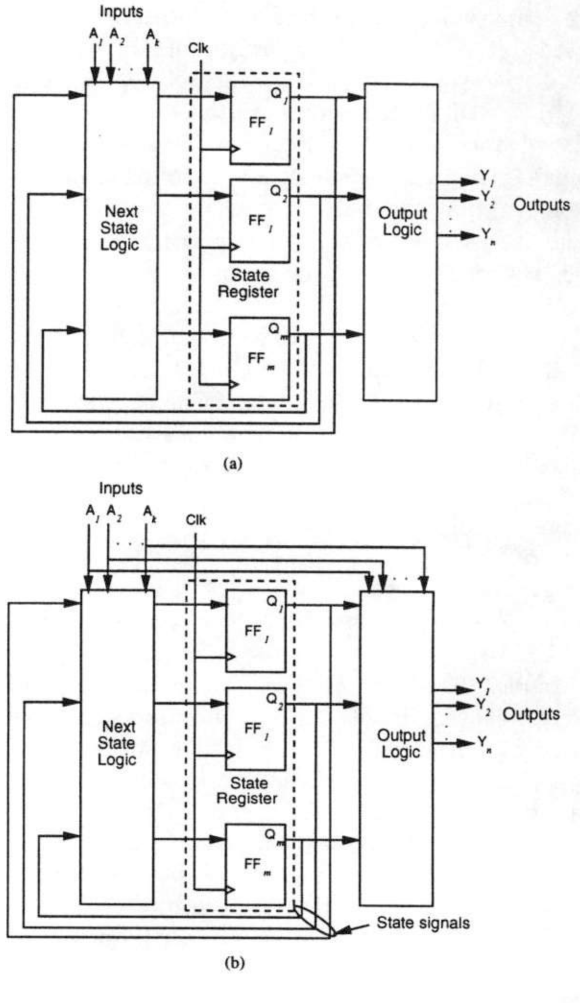


Figure 10: A generic controller design: (a) state-based, (b) input-based. (‡)

As mentioned in Section 1.3, there are two distinct types of controllers, those that are input-based and those that are state-based. These types of controllers differ in how they define the output function, h. For input-based controllers, h is defined as a mapping $S \times I \rightarrow O$, which means that the *Output logic* is dependent on two parameters, namely, *State register* and *Inputs*. For state-based controllers, on the other hand, h is defined as the mapping $S \rightarrow O$, which means the *Output logic* depends on only one parameter, the *State register*. Since the inputs and outputs are Boolean signals, in either case, this architecture is well-suited to implementing controllers that do not require complex data manipulation.

The controller synthesis consists of state minimization and encoding, Boolean minimization and technology mapping for the *Next-state* and *Output logic*.

2.2 Custom Datapath architecture

In a custom datapath we compute arbitrary expressions. In a datapath we use a different number of counters, registers, register-files and memories with a varied number of ports that are connected with several buses. Note that these same buses can be used to supply operands to functional units as well as to supply results back to storage units. It is also possible for the functional units to obtain operands from several buses, though this would require the use of a selector in front of each input. It is also possible for each unit to have input and output latches which are used to temporarily store the input operands or results. Such latching can significantly shorten the amount of time that the buses will be used for operand and result transfer, and thus can increase the traffic over these buses.

On the other hand, input and output latching requires a more complicated control unit since each operation requires more than one clock cycle. Namely, at least one clock cycle is required to fetch operands from registers, register files or memories and store them into input latches, at least one clock cycle to perform the operation and store a result into an output latch, and at least one clock cycle to store the result from an output latch back to a register or memory.

An example of such a custom datapath is shown in Figure 11. Note that it has a counter, a register, a 3-port register-file and a 2-port memory. It also has four buses and three functional units: two ALUs and a multiplier. As you can see, ALU1 does not have any latches, while ALU2 has latches at both the inputs and the outputs and the single multiplier has only the inputs latched. With this arrangement, ALU1 can receive its left operand from buses 2 and 4, while the multiplier can receive its right operand from buses 1 and 4. Similarly, the storage units can also receive data from several buses. Such custom datapaths are frequently used in application specific design to obtain the best performance-cost ratio.

Datapaths are also used in all standard processor implementations to perform numerical computation or data manipulations. A processor datapath consists of some temporary storage, in addition to arithmetic, logic and shift units. Lets consider, for example, how

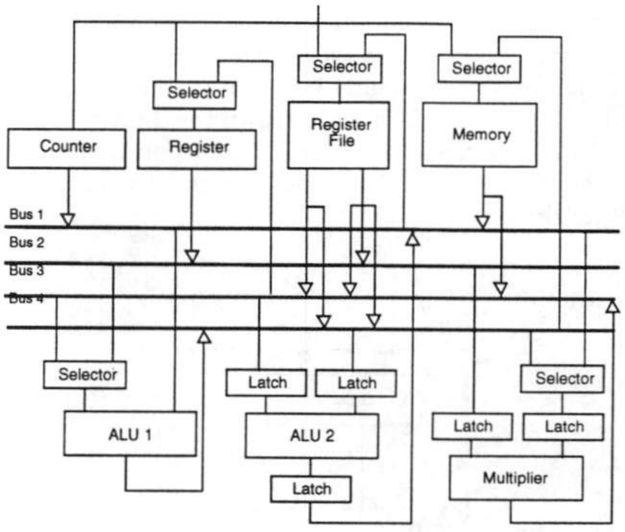


Figure 11: An example of a custom datapath. (‡)

we might perform the summation of a hundred numbers by declaring the sum to be a temporary variable, initially set to zero, and executing the following loop statement:

```
        $sum = 0$
loop:
        for $i = 1$ to 100
            $sum = sum + x_i$
        end loop
```

The above loop body could be executed on a datapath consisting of one register called an **Accumulator** and an ALU. The variable *sum* would be stored in the *Accumulator*, and in each clock cycle, the new x_i would be added to the *sum* in the ALU so that the new value of *sum* could again be stored in the *Accumulator*.

Generally speaking, the majority of digital designs work in the same manner. The variable values and constant are stored in registers or memories, they are fetched from storage components after the rising edge of the clock signal, they are transformed in combinatorial components during the time between two rising edges of the clock and the results are stored back into the storage components at the next rising edge of the clock signal.

In Figure 12, we have shown a simple datapath that could perform the above summation. This datapath contains a selector, which selects either 0 or some outside data as the left operand for the ALU. The right operand will always be the content of the *Accumulator*, which could also be output through a tri-state driver. The *Accumulator* is a shift register with a parallel load. This datapath's schematic is shown in Figure 12(a), and in Figure 12(b), we have shown the 9-bit control word that specifies the values of the control signals for the *Selector*, the *ALU*, the *Accumulator* and the output drivers.

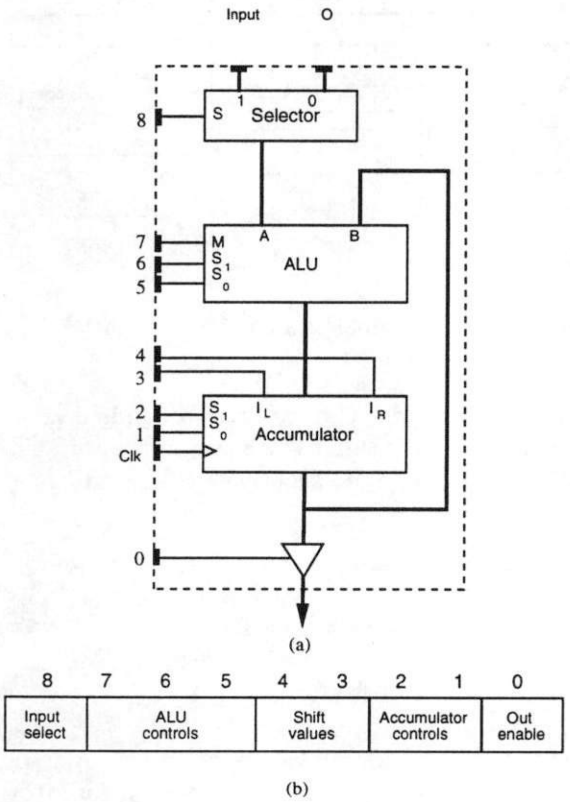


Figure 12: Simple datapath with one accumulator: (a) datapath schematic, (b) control word. (‡)

On each clock cycle, a specific control word would define the operation of the datapath. In order to compute the sum of 100 numbers, we would need 102 clock cycles. In this case, the control words would be the same for all the clock cycles, except the first and the last. In the first clock cycle, we would have to clear the *Accumulator*; in the next 100 clock cycles we would add the new data to the accumulated sum, finally, in the last clock cycle, we would output the accumulated sum.

Datapaths are also used in many applications where a fixed computation must be performed repeatedly on different sets of data, as is the case in the digital signal processing (DSP) systems used for digital filtering, image processing, and multimedia. A datapath architecture often consists of high-speed arithmetic units, connected in parallel, and heavily pipelined in order to achieve a high throughput.

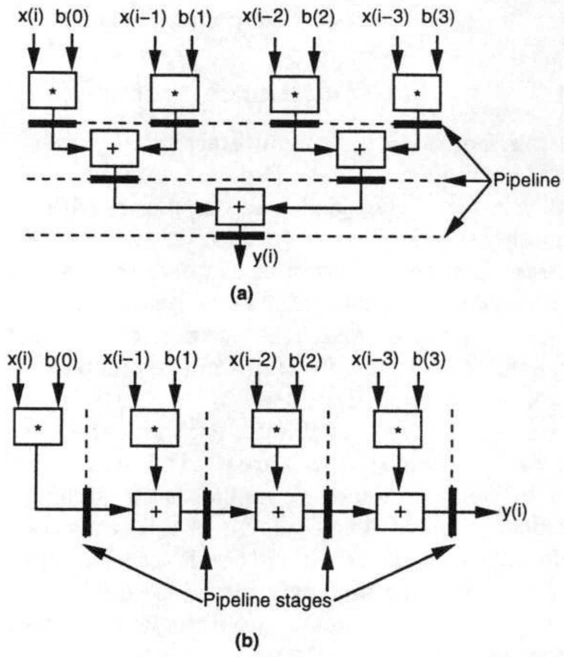


Figure 13: Two different datapaths for FIR filter: (a) with three pipeline stages, (b) with four pipeline stages. (†)

In Figure 13, we can see two different datapaths, both of which are designed to implement a finite-impulse-response (FIR) filter, which is defined by the expression

$$y(i) = \sum_{k=0}^{N-1} x(i-k)b(k)$$

where N is 4. Note that the datapath in Figure 13(a) performs all its multiplications concurrently, and adds the products in parallel by means of a summation tree. The datapath in Figure 13(b) also performs its multiplications concurrently, but it will then add the products serially. Further, note that the datapath in Figure 13(a) has three pipeline stages, each indicated by a dashed line, whereas the datapath in Figure 13(b) has four similarly indicated pipeline stages. Although both datapaths use four multipliers and three adders, the datapath in Figure 13(b) is regular and easier to implement in ASIC technologies.

In this kind of architecture, as long as each operation in an algorithm is implemented by its own unit, as in Figure 13, we do not need a control for the system, since data simply flows from one unit to the next, and the clock is used to load pipeline registers. Sometimes, however, it may be necessary to use fewer units to save silicon area, in which case we would need a simple controller to steer the data among the units and registers, and to select the appropriate arithmetic function for those units that can perform different functions at different times. Another situation would be to implement more than one algorithm with the same datapath, with each algorithm executing at a different time. In this case, since each algorithm requires a unique flow of data through the datapath, we would need a controller to regulate the flow. Such controllers are usually simple and without conditional branches.

2.3 FSMD architecture

A FSMD architecture implements the FSMD model by combining a controller with a datapath. As shown in Figure 14(a), the datapath has two types of I/O ports. One type of I/O ports are data ports which are used by the outside environment to send and receive data to and from the ASIC. The data could be of type integer, floating-point, or characters and it is usually packed into one or more words. The data ports are usually 8, 16, 32 or 64 bits wide. The other type of I/O ports are control ports which are used by the control unit to control the operations performed by the datapath and receive information about the status of selected registers in the datapath.

As shown in Figure 14(b), the datapath takes the operands from storage units, performs the computation in the combinatorial units and returns the results to storage units during each state, which is usually equal to one clock cycle.

As mentioned in the previous section the selection of operands, operations and the destination for the result is controlled by the control unit by setting proper values of datapath control signals. The datapath also indicates through status signals when a particular value is stored in a particular storage unit or when a particular relation between two data values stored in the datapath is satisfied.

Similar to the datapath, a control unit has a set of input and a set of output signals. Each signal is a Boolean variable that can take a value of 0 or 1. There are two types of input signals: external signals and status signals. External signals represent the conditions in the external environment on which the FSMD architecture must respond. On the other hand, the status signals represent the state of the datapath.

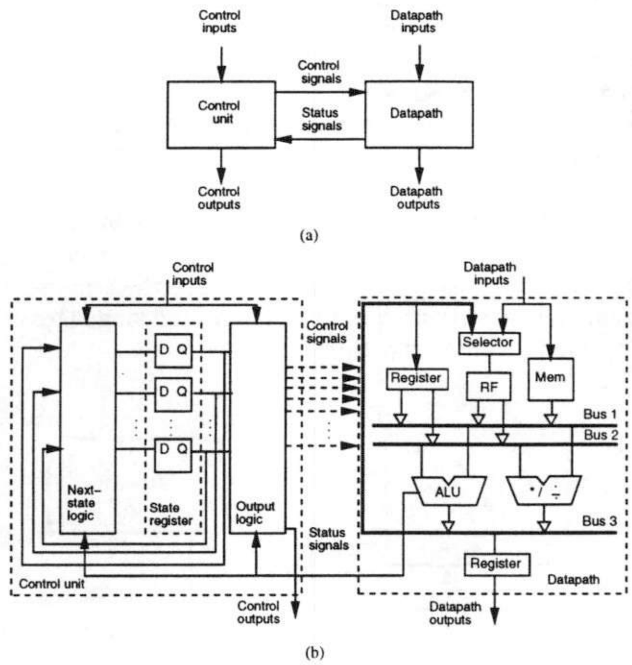


Figure 14: Design model: (a) high-level block diagram, (b) register-transfer-level block diagram. (‡)

Their value is obtained by comparing values of selected variables stored in the datapath. There are also two types of output signals: external signals and datapath control signals. External signals identify to the environment that a FSMD architecture has reached certain state or finished a particular computation. The datapath controls, as mentioned before, select the operation for each component in the datapath.

FSMD architectures are used for various ASIC designs. Each ASIC design consists of one or more FSMD architectures, although two implementations may differ in the number of control units and datapaths, the number of components and connections in the datapath, the number of states in the control unit and the number of I/O ports. The FSM controller and DSP datapath mentioned above are two special cases of this kind of architecture. In addition, the FSMD is also the basic architecture for general-purpose processors, since each processor includes both a control unit and a datapath.

2.4 CISC architecture

The primary motivation for developing an architecture of **complex-instruction-set computers** (CISC) was to reduce the number of instructions in compiled code, which would in turn minimize the number of memory accesses required for fetching instructions. The motivation was valid in the past, since memories were expensive and much slower than processors. The secondary motivation for CISC development was to simplify compiler construction, by including in the processor instruction set complex instructions that mimic programming language constructs. These complex instructions would reduce the semantic gap between programming and machine languages and simplify compiler construction.

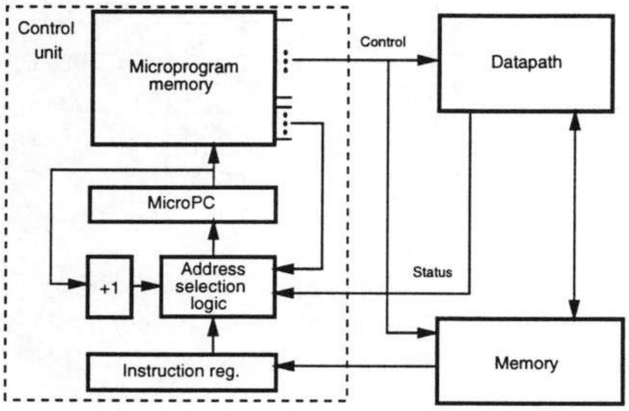


Figure 15: CISC with microprogrammed control. (†)

In order to support a complex instruction set, a CISC usually has a complex datapath, as well as a controller that is microprogrammed, shown in Figure 15, which consists of a *Microprogram memory*, a *Microprogram counter (MicroPC)*, and the *Address selection logic.* Each word in the microprogram memory represents one control word, such as the one shown in Figure 12, that contains the values of all the datapath control signals for one clock cycle. This means that each bit in the control word represents the value of one datapath control line, used for loading a register or selecting an operation in the ALU, for example. Furthermore, each processor instruction consists of a sequence of control words. When such an instruction is fetched from the *Memory*, it is stored first in the *Instruction register*, and then used by the *Address selection logic* to determine the starting address of the corresponding control-word sequence in the *Microprogram memory.* After this starting address has been loaded into the *MicroPC*, the corresponding control word will be fetched from the *Microprogram memory*, and used to transfer the data in the datapath from one register to another. Since the *MicroPC* is concurrently incremented to point to the next control word, this procedure will be repeated for each control word in the sequence. Finally, when the last control word is being executed, a new instruction will be fetched from the *Memory*, and the entire process will be repeated.

From this description, we can see that the number of control words, and thus the number of clock cycles can vary for each instruction. As a result, instruction pipelining can be difficult to implement in CISCs. In addition, relatively slow microprogram memory requires a clock cycle to be longer than necessary. Since instruction pipelines and short clock cycles are necessary for fast program execution, CISC architectures may not be well-suited for high-performance processors.

Although a variety of complex instructions could be executed by a CISC architectures, program-execution statistics have shown that the instructions used most frequently tend to be simple, with only a few addressing modes and data types. Statistics have also shown that the most complex instructions were seldom or never used. This low usage of complex instructions can be attributed to the slight semantic differences between programming language constructs and available complex instructions, as well as the difficulty in mapping language constructs into such complex instructions. Because of this difficulty, complex instructions are seldom used in optimizing compilers for CISC processors, thus reducing the usefulness of CISC architectures.

The steadily declining prices of memories and their increasing speeds have made compactly-coded programs and complex instruction sets unnecessary for high-performance computing. In addition, complex instruction sets have made construction of optimizing compilers for CISC architecture too costly. For these two reasons, the CISC architecture was displaced in favor of the RISC architecture.

2.5 RISC architecture

In contrast to the CISC architecture, the architecture of a **reduced-instruction-set computer** (RISC) is optimized to achieve short clock cycles, small numbers of cycles per instruction, and efficient pipelining of instruction streams. As shown in Figure 16, the datapath of an RISC processor generally consists of a large register file and an ALU. A large register file is necessary since it contains all the operands and the results for program computation. The data is brought to the register file by load instructions and returned to the memory by store instructions. The larger the register file is, the smaller the number of load and store instructions in the code. When the RISC executes an instruction, the instruction pipe begins by fetching an instruction into the *Instruction register.* In the second pipeline stage the instruction is then decoded and the appropriate operands are fetched from the *Register file.* In the third stage, one of two things occurs: the RISC either executes the required operation in the *ALU*, or, alternatively, computes the address for the *Data cache.* In the fourth stage the data is stored in either the *Data cache* or in the *Register file.* Note that the execution of each instruction takes only four clock cycles, approximately, which means that the instruction pipeline is short and efficient, losing very few cycles in the case of data or branch dependencies.

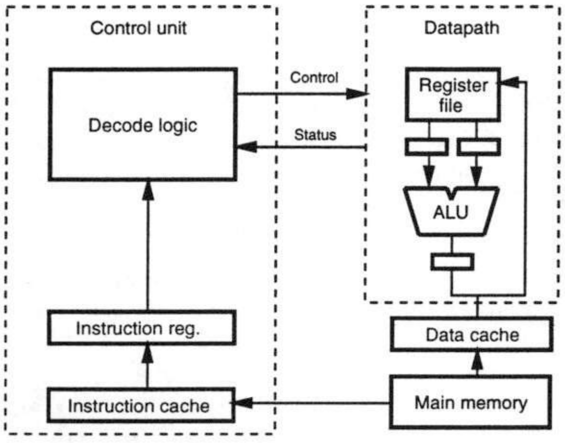


Figure 16: RISC with hardwired control. (†)

We should also note that, since all the operands are contained in the register file, and only simple addressing modes are used, we can simplify the design of the datapath as well. In addition, since each operation can be executed in one clock cycle and each instruction in four, the control unit remains simple and can be implemented with random logic, instead of microprogrammed control. Overall, this simplification of the control and datapath in the RISC results in a short clock cycle, and, ultimately, higher performance.

It should also be pointed out, however, that the greater simplicity of RISC architectures require a more sophisticated compiler. For example, a RISC design does not stop the instruction pipeline whenever instruction dependencies occur, which means that the compiler is responsible for generating a dependency-free code, either by delaying the issue of instructions or by reordering instructions. Furthermore, due to the fact that the number of instructions is reduced, the RISC compiler will need to use a sequence of RISC instructions in order to implement complex operations. At the same time, of course, although these features require more sophistication in the compiler, they also give the compiler a great deal of flexibility in performing aggressive optimization.

Finally, we should note that RISC programs tend to require 20% to 30% more program memory, due to the lack of complex instructions. However, since simpler instruction sets can make compiler design and running time much shorter, the efficiency of the compiled code is ultimately much higher. In addition, because of these simpler instruction sets, RISC processors tend to require less silicon area and a shorter design cycle than their CISC counterparts.

2.6 VLIW architecture

A **very-long-instruction-word computer** (VLIW) exploits parallelism by using multiple functional units in its datapath, all of which execute in a lock step manner under one centralized control. A VLIW instruction contains one field for each functional unit, and each field of a VLIW instruction specifies the addresses of the source and destination operands, as well as the operation to be performed by the functional unit. As a result, a VLIW instruction is usually very wide, since it must contain approximately one standard instruction for each functional unit.

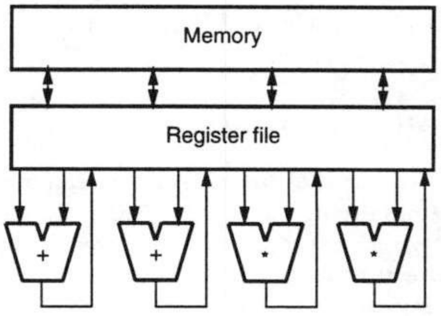


Figure 17: An example of VLIW datapath. (†)

In Figure 17, we see an example of a VLIW datapath, consisting of four functional units: namely, two ALUs and two multipliers, a register file and a memory. In order to utilize all the four functional units, the register file in this example has 16 ports: eight output ports, which supply operands to the functional units, four input ports, which store the results obtained from functional units, and four input/output ports, designed to allow communication with the memory.

What is interesting to note here is that, ideally, the VLIW in Figure 17 would provide four times the performance we could get from a processor with a single functional unit, under the assumption that the code executing on the VLIW had four-way parallelism, which enables the VLIW to execute four independent instructions in each clock cycle. In reality, however, most code has a large amount of parallelism interleaved with code that is fundamentally serial. As a result, a VLIW with a large number of functional units might not be fully utilized. The ideal conditions would also require us to assume that all the operands were in the register file, with 8 operands being fetched and four results stored back on every clock cycle, in addition to four new operands being brought from the memory to be available for use in the next clock cycle. It must be noted, however, that this computation profile is not easy to achieve, since some results must be stored back to memory and some results may not be needed in the next clock cycle. Under these conditions, the efficiency of a VLIW datapath might be less than ideal.

Finally, we should point out that there are two technological limitation that can affect the implementation of a VLIW architecture. First, while register files with 8–16 ports can be built, the efficiency and performance of such register files tend to degrade quickly when we go beyond that number. Second, since VLIW program and data memories require a high communication bandwidth, these systems tend to require expensive high-pin packaging technology as well. Overall, these are the reasons why VLIW architectures are not as popular as RISC architectures.

2.7 Parallel architecture

In the design of **parallel processors**, we can take advantage of spatial parallelism by using multiple processing elements (PEs) that work concurrently. In this type of architecture, each PE may contain its own datapath with registers and a local memory. Two typical types of parallel processors are the **SIMD** (single instruction multiple data) and the **MIMD** (multiple instruction multiple data) processors.

In SIMD processors, usually called **array processors**, all of the PEs execute the same instruction in a lock step manner. To broadcast the instructions to all the PEs and to control their execution, we generally use a single global controller. Usually, an array processor is attached to a host processor, which means that it can be thought of as a kind of hardware accelerator for tasks that are computationally intensive. In such cases, the host processor would load the data into each PE, and then collect the results after the computations are finished. When it is necessary, PEs can also communicate directly with their nearest neighbors.

The primary advantage of array processors is that they are very convenient for computations that can be naturally mapped on a rectangular grid, as in the case of image processing, where an image is decomposed into pixels on a rectangular grid, or in the case of weather forecasting, where the surface of the globe is decomposed into n-by-n-mile squares. Programming one grid point in the rectangular array processor is quite easy, since all the PEs execute the same instruction stream. However, programming any data routing through the array is very difficult, since the programmer would have to be aware of all the positions of each data for every clock cycle. For this reason, problems, like matrix triangulations or inversions, are difficult to program on an array processor.

Array processors, then, are easy to build and easy to program, but only when the natural structure of the problem matches the topology of the array processor. As a result, they can not be considered general purpose machines, because users have difficulty writing programs for general classes of problems.

An MIMD processor, usually called a **multiprocessor system**, differs from an SIMD in that each PE executes its own instruction stream. In this kind of architecture, the program can be loaded by a host processor, or each processor can load its own program from a shared memory. Each processor can communicate with every other processor within the multiprocessor system, using one of the two communication mechanisms. In a **shared-memory** multiprocessor, all the processors are connected to a shared memory through an interconnection network, which means that each processor can access any data in the shared memory. In a **message-passing** multiprocessor, on the other hand, each processor tends to have a large local memory, and sends data to other processors in the form of messages through an interconnection network. The interconnection network for a shared memory must be fast, since it is very frequently used to communicate small amounts of data, like a single word. In contrast, the interconnection network used for message passing tends to be much slower, since it is used less frequently and communicates long messages, including many words of data. Finally, it should be noted that multiprocessors are much easier to program, since they are task-oriented instead of instruction-oriented. Each task runs independently and can be synchronized after completion, if necessary. Thus, multiprocessors make program and data partitioning, code parallelization and compilation much simpler than array processors.

Such a multiprocessor, in which the interconnection network consists of several buses, is shown in Figure 18. Each **processing element** (PE) consists of a processor or ASIC and a local memory connected by the local bus. The shared or global memory may be either single port, dual port, or special purpose memory such as FIFO. The PEs and global memories are connected by one or more **system buses** via corresponding **interfaces**. The system bus is associated with a well-defined **protocol** to which the components on the bus have to respect. The protocol may be standard, such as VME, or custom. An **interface** bridges the gap between a local bus of a PE/memory and system buses.

The heterogeneous architecture is a superset of all previous architectures and it can be customized for a particular application to achieve the best cost-performance trade-off. Figure 19 shows some typical configurations.

Figure 19(a) shows a simple embedded processor system with an IO device. The IO device directly communicate with the processor on the processor bus via an interface. Figure 19(b) shows a shared memory system where two PEs are connected to a global memory via a system bus. The PEs can be either processor system or ASIC system, each of which may contain its own local bus and memory subsystem. Figure 19(c) and (d) show two types of message passing system where two PEs communicate via a channel. The former can perform asynchronous communication given the dedicated devices such as a FIFO. The latter can perform synchronous communication if the proper handshaking between the two PEs are performed via the system bus.

3 Languages

3.1 Introduction

A system can be described at any one of several distinct levels of abstraction, each of which serves a particular purpose. By describing a system at the logic level, for example, designers can verify detailed timing as well as functionality. Alternatively, at the architectural level, the complex interaction among system components such as processors, memories, and ASICs can be verified. Finally, at the conceptual level, it is possible to describe the system's functionality without any notion of its components. Descriptions at such level can serve as the specification of the system for

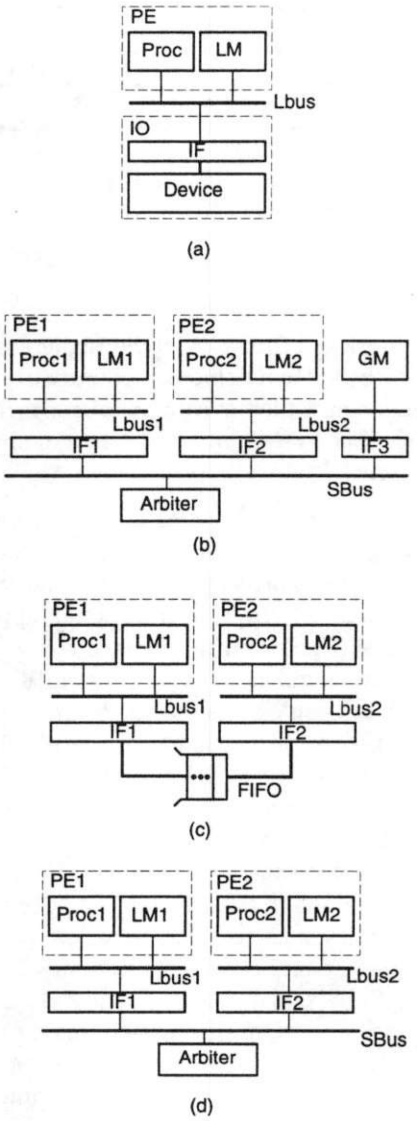


Figure 19: Some typical configurations: (a) standard processor, (b) shared memory, (c) non-blocking message passing, (c) blocking message passing.

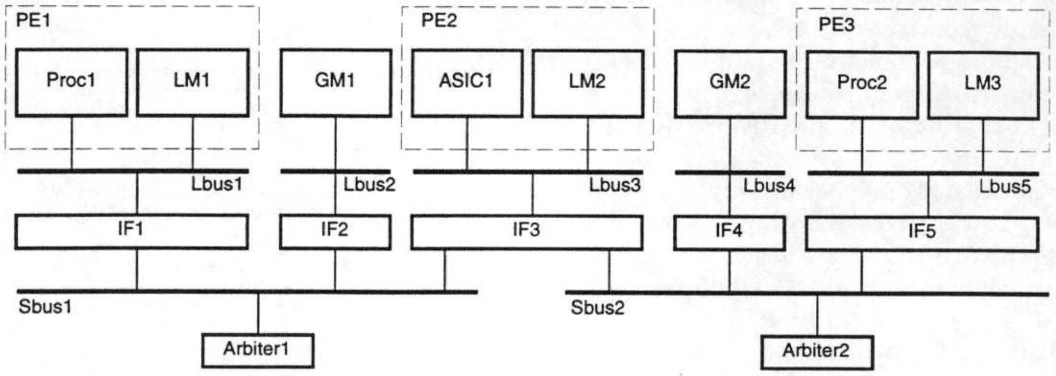


Figure 18: A heterogeneous multiprocessor

designers to work on. Increasingly, designers need to conceptualize the system using an **executable specification** language, which is capable of capturing the functionality of the system in a machine-readable and simulatable form.

Such an approach has several advantages. First, simulating an executable specification allows the designer to verify the correctness of the system's intended functionality. In the traditional approach, which started with a natural-language specification, such verification would not be possible until enough of the design had been completed to obtain a simulatable system description (usually gate-level schematics). The second advantage of this approach is that the specification can serve as an input to codesign tools, which, in turn, can be used to obtain an implementation of the system, ultimately reducing design times by a significant amount. Third, such a specification can serve as comprehensive documentation, providing an unambiguous description of the system's intended functionality. Finally, it also serves as a good medium for the exchange of design information among various users and tools. As a result, some of the problems associated with system integration can be minimized, since this approach would emphasize well-defined system components that could be designed independently by different designers.

The increasing design complexity associated with systems-on-a-chip also makes an **executable modeling** language extremely desirable where an intermediate implementation can be represented and validated before proceeding to the next synthesis step. For the same reason, we need such a modeling language to be able to describe design artifacts from previous designs and intellectual properties (IP) provided by other sources.

Since different conceptual models possess different characteristics, any given specification language can be well or poorly suited for that model, depending on whether it supports all or just a few of the model's characteristics. To find the language that can capture a given conceptual model directly, we would need to establish a one-to-one correlation between the characteristics of the model and the constructs in the language.

3.2 Characteristics of system models

In this section, we will present some of the characteristics most commonly found in modeling systems. In presenting these characteristics, part of our goal will be to assess how useful each characteristic is in capturing one or more types of system behavior.

3.3 Concurrency

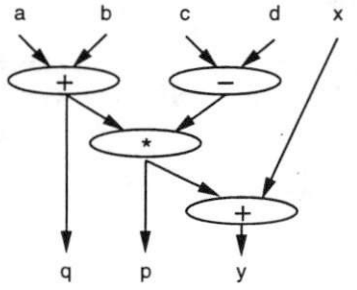
Any system can be decomposed into chunks of functionality called **behaviors**, each of which can be described in several ways, using the concepts of processes, procedures or state machines. In many cases, the functionality of a system is most easily conceptualized as a set of concurrent behaviors, simply because representing such systems using only sequential constructs would result in complex descriptions that can be difficult to comprehend. If we can find a way to capture concurrency, however, we can usually obtain a more natural representation of such systems. For example, consider a system with only two concurrent behaviors that can be individually represented by the finite-state machines F_1 and F_2. A standard representation of the system would be a cross product of the two finite-state machines, $F_1 \times F_2$, potentially resulting in a large number of states. A more elegant solution, then, would be to use a conceptual model that has two or more concurrent finite-state machines, as do the Statecharts [Har87] and many other concurrent languages.

Concurrency representations can be classified into two groups, data-driven or control-driven, depending on how explicitly the concurrency is indicated. Furthermore, a special class of data-driven concurrency called pipelined concurrency is of particular importance to signal processing applications.

Data-driven concurrency: Some behaviors can be clearly described as sets of operations or statements without specifying any explicit ordering for their execution. In a case like this, the order of execution would be determined only by data dependencies between them. In other words, each operation will perform a computation on input data, and then output new data, which will, in turn, be input to other operations. Operation executions in such **dataflow** descriptions depend only upon the availability of data, rather than upon the physical location of the operation or statement in the specification. Dataflow representations can be easily described from programming languages using the **single assignment rule**, which means that each variable can appear exactly once on the left hand side of an assignment statement.

```
1: q = a + b
2: y = p + x
3: p = (c – d) * q
```

(a)



(b)

Figure 20: Data-driven concurrency: (a) dataflow statements, (b) dataflow graph generated from (a). (†)

Consider, for example, the single assignment statements in Figure 20(a). As in any other data-driven execution, it is of little consequence that the assignment to p follows the statement that uses the value of p to compute the value of y. Regardless of the sequence of the statements, the operations will be executed solely as determined by availability of data, as shown in the dataflow graph of Figure 20(b). Following this principle, we can see that, since a, b, c and d are primary inputs, the add and subtract operations in statements 1 and 3 will be carried out first. The results of these two computations will provide the data required for the multiplication in statement 3. Finally, the addition in statement 2 will be performed to compute y.

Pipelined concurrency: Dataflow description in the previous section can be viewed as a set of operations which consume data from their inputs and produce data on their outputs. Since the execution of each operation is determined by the availability of its input data, the degree of concurrency that can be exploited is limited by data dependencies. However, when the same dataflow operations are applied to a stream of data samples, we can use **pipelined** concurrency to improve the throughput, that is, the rate at which the system is able to process the data stream. Such throughput improvement is achieved by dividing operations into groups, called pipeline **stages**, which operate on different data sets in the stream. By operating on different data sets, pipeline stages can run concurrently. Note that each stage will take the same amount of time, called a **cycle**, to compute its results.

For example, Figure 21(a) shows a dataflow graph operating on the data set $a(n), b(n), c(n), d(n)$ and x(n), while producing the data set $q(n), p(n)$ and $y(n)$, where the index n indicates the nth data in the stream, called **data sample** n. Figure 21(a) can be converted into a pipeline by partitioning the graph into three stages, as shown in Figure 21(b).

In order for the pipeline stages to execute concurrently, storage elements such as registers or FIFO queues have to be inserted between the stages (indicated by thick lines in Figure 21(b)). In this way, while the second stage is processing the results produced by the first stage at the previous cycle, the first stage can simultaneously process the next data sample in the stream. Figure 21(c) illustrates the pipelined execution of Figure 21(b), where each row represents a stage, each column represents a cycle. In the third column, for example, while the first stage is adding $a(n+2)$ and $b(n+2)$, and subtracting $c(n+2)$ and $d(n+2)$, the second stage is multiplying $(a(n+1)+b(n+1))$ and $c(n+1)-d(n+1)$, and the third stage is finishing the computation of the nth sample by adding $((a(n)+b(n)) * (c(n)-d(n))$ to $x(n)$.

Control-driven concurrency: The key concept in control-driven concurrency is the control thread, which can be defined as a set of operations in the system that must be executed sequentially. As mentioned above, in data-driven concurrency, it is the dependen-

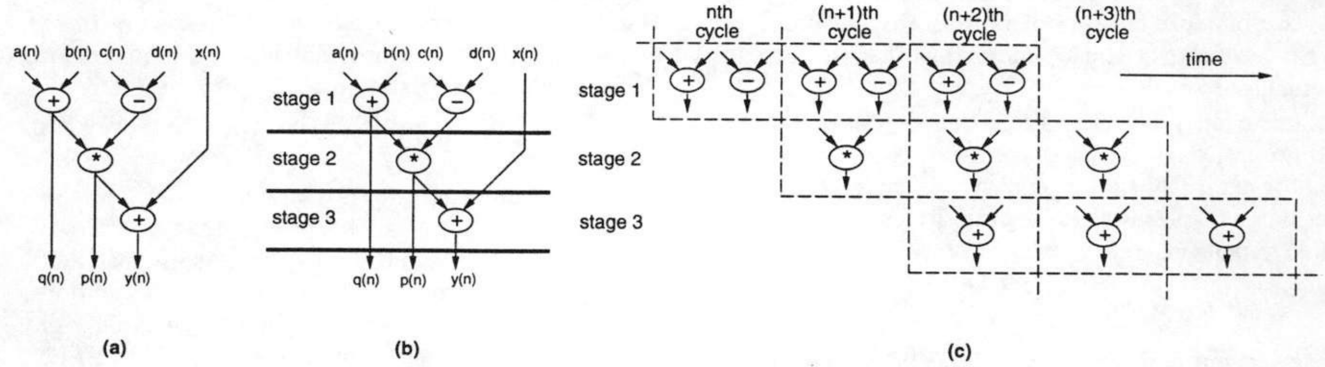


Figure 21: Pipelined concurrency: (a) original dataflow, (b) pipelined dataflow, (c) pipelined execution.

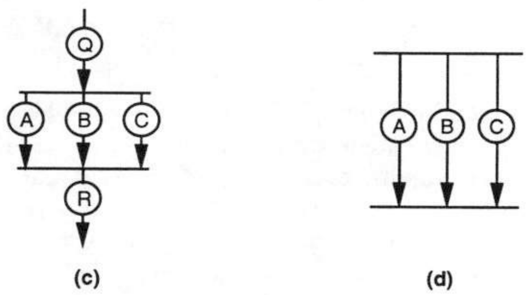
cies between operations that determine the execution order. In control-driven concurrency, by contrast, it is the control thread or threads that determine the order of execution. In other words, control-driven concurrency is characterized by the use of explicit constructs that specify multiple threads of control, all of which execute in parallel.

```
sequential behavior X
begin
    Q();
    fork  A(); B(); C(); join;
    R();
end behavior X;
```

(a)

```
concurrent behavior X
begin
    process A();
    process B();
    process C();
end behavior X;
```

(b)



(c) (d)

Figure 22: Control-driven concurrency: (a) fork-join statement, (b) process statement, (c) control threads for fork-join statements, (d) control threads for process statement. (†)

Control-driven concurrency can be specified at the task level, where constructs such as fork-joins and processes can be used to specify concurrent execution of operations. Specifically, a **fork** statement creates a set of concurrent control threads, while a **join** statement waits for the previously forked control threads to terminate. The *fork* statement in Figure 22(a), for example, spawns three control threads *A, B* and *C*, all of which execute concurrently. The corresponding *join* statement must wait until all three threads have terminated, after which the statements in *R* can be executed. In Figure 22(b), we can see how *process* statements are used to specify concurrency. Note that, while a fork-join statement starts from a single control thread and splits it into several concurrent threads as shown in Figure 22(c), a *process* statement represents the behavior as a set of concurrent threads, as shown in Figure 22(d). For example, the **process** statements of Figure 22(b) create three processes *A, B* and *C*, each representing a different control thread. Both *fork-join* and *process* statements may be nested, and both approaches are equivalent to each other in the sense that a *fork-join* can be implemented using nested processes and vice versa.

3.4 State transitions

Systems are often best conceptualized as having various **modes**, or states, of behavior, as in the case of controllers and telecommunication systems. For example, a traffic-light controller [DH89] might incorporate different modes for day and night operation, for manual and automatic functioning, and for the status of the traffic light itself.

In systems with various modes, the transitions between these modes sometimes occur in an unstructured manner, as opposed to a linear sequencing through the modes. Such arbitrary transitions are akin to the use of *goto* statements in programming languages. For example, Figure 23 depicts a system that transitions between modes *P, Q, R, S* and *T*, the sequencing determined solely by certain conditions. Given a state machine with N states, there can be $N \times N$ possible transitions among them.

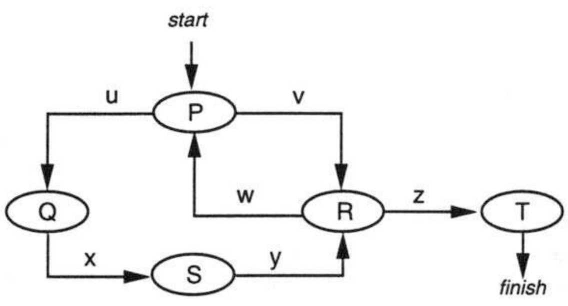


Figure 23: State transitions between arbitrarily complex behaviors. (†)

In systems like this, transitions between modes can be triggered by the detection of certain events or certain conditions. For example, in Figure 23, the transition from state P to state Q will occur whenever event u happens while in P. In some systems, actions can be associated with each transition, and a particular mode or state can have an arbitrarily complex behavior or computation associated with it. In the case of the traffic-light controller, for example, in one state it may simply be sequencing between the red, yellow and green lights, while in another state it may be executing an algorithm to determine which lane of traffic has a higher priority based on the time of the day and the traffic density. In simple (Section 1.3) and hierarchical (Section 1.6) finite-state machine models, simple assignment statements, such as $x = y + 1$, can be associated with a state. In the PSM model (Section 1.8), any arbitrary program with iteration and branching constructs can be associated with a state.

3.5 Hierarchy

One of the problems we encounter with large systems is that they can be too complex to be considered in their entirety. In such cases, we can see the advantage of hierarchical models. First, since hierarchical models allow a system to be conceptualized as a set of smaller subsystems, the system modeler is able to focus on one subsystem at a time. This kind of modular decomposition of the system greatly simplifies the development of a conceptual view of the system. Furthermore, once we arrive at an adequate conceptual view, the hierarchical model greatly facilitates our comprehension of the system's functionality. Finally, a hierarchical model provides a mechanism for scoping objects, such as declaration types, variables and subprogram names. Since a lack of hierarchy would make all such objects global, it would be difficult to relate them to their particular use in the model, and could hinder our efforts to reuse these names in different portions of the same model.

There are two distinct types of hierarchy – structural hierarchy and behavioral hierarchy – both of which are commonly found in conceptual views of systems.

Structural hierarchy: A structural hierarchy is one in which a system specification is represented as a set of interconnected components. Each of these components, in turn, can have its own internal structure, which is specified with a set of lower-level interconnected components, and so on. Each instance of an interconnection between components represents a set of communication channels connecting the components. The advantage of a model that can represent a structural hierarchy is that it can help the designer to conceptualize new components from a set of existing components.

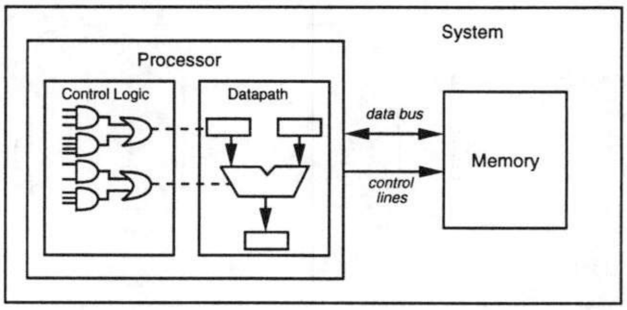


Figure 24: Structural hierarchy. (†)

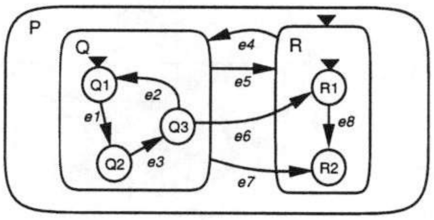
This kind of structural hierarchy in systems can be specified at several different levels of abstraction. For example, a system can be decomposed into a set of processors and ASICs communicating over buses in a parallel architecture. Each of these chips may consist of several blocks, each representing a FSMD architecture. Finally, each RT component in the FSMD architecture can be further decomposed into a set of gates while each gate can be decomposed into a set of transistors. In addition, we should note that different portions of the system can be conceptualized at different levels of abstraction, as in Figure 24, where the processor has been structurally decomposed into a datapath represented as a set of RT components, and into its corresponding control logic represented as a set of gates.

Behavioral hierarchy: The specification of a **behavioral hierarchy** is defined as the process of decomposing a behavior into distinct subbehaviors, which can be either sequential or concurrent.

The **sequential decomposition** of a behavior may be represented as either a set of procedures or a state machine. In the first case, a **procedural sequential decomposition** of a behavior is defined as the process of representing the behavior as a sequence of procedure calls. Even in the case of a behavior that consists of a single set of sequential statements, we can still think of that behavior as comprising a procedure which encapsulates those statements. A procedural sequential decomposition of behavior P is shown in Figure 25(a), where behavior P consists of a sequential execution of the subbehaviors represented by procedures Q and R. Behavioral hierarchy would be represented here by nested procedure calls. Recursion in procedures allows us to specify a dynamic behavioral hierarchy, which means that the depth of the hierarchy will be determined only at run time.

```
behavior P
   variable x, y;
begin
   Q(x) ;
   R(y) ;
end P;
```

(a)



(b)

Figure 25: Sequential behavioral decomposition: (a) procedures, (b) state-machines. (†)

Figure 25(b) shows a **state-machine sequential decomposition** of behavior P. In this diagram, P is decomposed into two sequential subbehaviors Q and R, each of which is represented as a state in a state-machine. This state-machine representation conveys hierarchy by allowing a subbehavior to be represented as another state-machine itself. Thus, Q and R are state-machines, so they are decomposed further into sequential subbehaviors. The behaviors at the bottom level of the hierarchy, including *Q1, ... R2*, are called **leaf behaviors**.

In a sequentially decomposed behavior, the subbehaviors can be related through several types of transitions: simple transitions, group transitions and hierarchical transitions. A **simple transition** is similar to that which connects states in an FSM model in that it causes control to be transferred between two states that both occupy the same level of the behavioral hierarchy. In Figure 25(b), for example, the transition triggered by event *e1* transfers control from behavior *Q1* to *Q2*. **Group transitions** are those which can be specified for a group of states, as is the case when event *e5* causes a transition from *any* of the subbehaviors of Q to the behavior R. **Hierarchical transitions** are those (simple or group) transitions which span several levels of the behavioral hierarchy. For example, the transition labeled *e6* transfers control from behavior *Q3* to behavior *R1*, which means that it must span two hierarchical levels. Similarly, the transition labeled *e7* transfers control from Q to state *R2*, which is at a lower hierarchical level.

For a sequentially decomposed behavior, we must explicitly specify the **initial** subbehavior that will be activated whenever the behavior is activated. In Figure 25(b), for example, R is the first subbehavior that is active whenever its parent behavior P is activated, since a solid triangle points to this first subbehavior. Similarly, *Q1* and *R1* would be the initial subbehaviors of behaviors Q and R, respectively.

The **concurrent decomposition** of behaviors allows subbehaviors to run in parallel or in pipelined fashion.

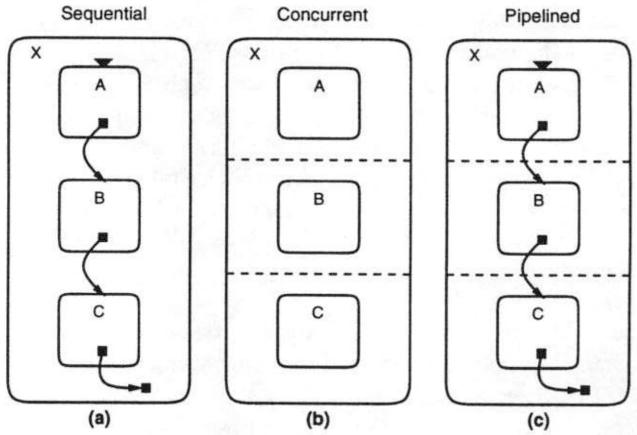


Figure 26: Behavioral decomposition types: (a) sequential, (b) parallel, (c) pipelined.

Figure 26 shows a behavior X consisting of three subbehaviors A, B and C. In Figure 26(a) the subbehaviors are running sequentially, one at a time, in the order indicated by the arrows. In Figure 26(b),

A, B and C run in parallel, which means that they will start when X starts, and when all of them finish, X will finish, just like the fork-join construct discussed in Section 3.3. In Figure 26(c), A, B and C run in pipelined mode, which means that they represent pipeline stages which run concurrently where A supplies data to B and B to C as discussed in Section 3.3.

3.6 Programming constructs

Many behaviors can best be described as sequential algorithms. Consider, for example, the case of a system intended to sort a set of numbers stored in an array, or one designed to generate a set of random numbers. In such cases, if the system designer manages to decompose the behavior hierarchically into smaller and smaller subbehaviors, he will eventually reach a stage where the functionality of a subbehavior can be most directly specified by means of an algorithm.

The advantage of using such programming constructs to specify a behavior is that they allow the system modeler to specify an explicit sequencing for the computations in the system. Several notations exist for describing algorithms, but programming language constructs are most commonly used. These constructs include assignment statements, branching statements, iteration statements and procedures. In addition, data types such as records, arrays and linked lists are usually helpful in modeling complex data structures.

```
int    buf[10], i, j;

for( i = 0; i < 10; i ++ )
    for( j = 0; j < i; j ++ )
        if( buf[i] > buf[j] )
            swap( &buf[i], &buf[j] );
```

Figure 27: Code segment for sorting.

Figure 27 shows how we would use programming constructs to specify a behavior that sorts a set of ten integers into descending order. Note that the procedure *swap* exchanges the values of its two parameters.

3.7 Behavioral completion

Behavioral completion refers to a behavior's ability to indicate that it has completed, as well as to the ability of other behaviors to detect this completion. A behavior is said to have completed when all the computations in the behavior have been performed, and all the variables that have to be updated have had their new values written into them.

In the finite-state machine model, we usually designate an explicitly defined set of states as **final states**. This means that, for a state machine, completion will have occurred when control flows to one of these final states, as shown in Figure 28(a).

In cases where we use programming language constructs, a behavior will be considered complete when the last statement in the program has been executed. For example, whenever control flows to a return statement, or when the last statement in the procedure is executed, a procedure is said to be complete.

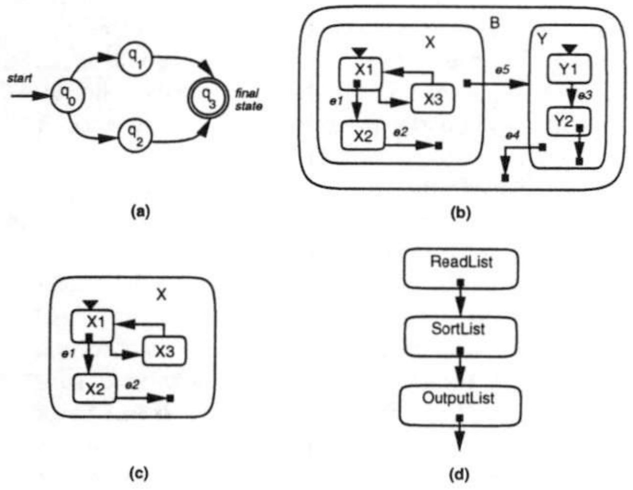


Figure 28: Behavioral completion: (a) finite-state machine, (b) program-state machine, (c) a single level view of the program-state X, (d) decomposition into sequential subbehaviors. (†)

The PSM model denotes completion using a special predefined **completion point**. When control flows to this completion point, the program-state enclosing it is said to have completed, at which point the transition-on-completion (TOC) arc, which can be traversed only when the source program-state has completed, could now be traversed.

For example, consider the program-state machine in Figure 28(b). In this diagram, the behavior of leaf program-states such as *X1* have been described with programming constructs, which means that their completion will be defined in terms of their execution of the last statement. The completion point of the program-state machine for X has been represented as a bold square. When control flows to it from program-state *X2* (i.e., when the arc labeled by event *e2* is traversed), the program-state X will be said to have completed. Only then can event *e5* cause a TOC transition to program-state Y. Similarly, program-state B will be said to have completed whenever control flows along the TOC arc labeled *e4* from program-state Y to the completion point for B.

The specification of behavioral completion has two advantages. First, in hierarchical specifications, completion helps designers to conceptualize each hierarchical level, and to view it as an independent module, free from interference from inter-level transitions. Figure 28(c), for example, shows how the program-state X in Figure 28(b) would look by itself in isolation from the larger system. Having decomposed the functionality of X into the program-substates *X1*, *X2* and *X3*, the system modeler does not have to be concerned with the effects of the completion transition labeled by event *e5*. From this perspective, the designer can develop the program-state machine for X independently, with its own completion point (transition labeled *e2* from *X2*). The second advantage of specifying behavioral completion is that the concept allows the natural decomposition of a behavior into subbehaviors which are then sequenced by the "completion" transition arcs. For example, Figure 28(d) shows how we can split an application which sorts a list of numbers into three distinct, yet meaningful subbehaviors: *ReadList, SortList* and *OutputList.* Since TOC arcs sequence these behaviors, the system requires no additional events to trigger the transitions between them.

3.8 Exception handling

Often, the occurrence of a certain event can require that a behavior or mode be interrupted immediately, thus prohibiting the behavior from updating values further. Since the computations associated with any behavior can be complex, taking an indefinite amount of time, it is crucial that the occurrence of the event, or **exception**, should terminate the current behavior immediately rather than having to wait for the computation to complete. When such exceptions arise, the next behavior to which control will be transferred is indicated explicitly.

Depending on the direction of transferred control the exceptions can be further divided into two groups: (a) **abortion**, when the behavior is terminated, and (b) **interrupt**, where control is temporarily transferred to other behaviors. An example of an abortion is shown in Figure 29(a) where behavior X is terminated after the occurence of events *e1* or *e2*. An example of interrupt is shown in Figure 29(b) where control from behavior X is transferred to Y or Z after the occurrence of *e1* or *e2* and is returned after their

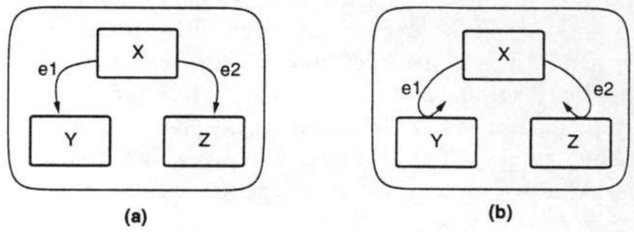


Figure 29: Exception types: (a) abortion, (b) interrupt.

completion.

Examples of such exceptions include resets and interrupts in many computer systems.

3.9 Timing

On many occasions in system specifications, there may be need to specify detailed timing relations, when a component receives or generates events in specific time ranges, which are measured in real time units such as nanoseconds.

In general, a timing relation can be described by a 4-tuple $T = (e1, e2, min, max)$, where event e1 preceeds e2 by at least *min* time units and at most *max* time units. When such a timing relation is used with real components it is called **timing delay**, when it is used with component specifications it is called **timing constraint**.

Such timing information is especially important for describing parts of the system which interact extensively with the environment according to a predefined **protocol**. The protocol defines the set of timing relations between signals, which both communicating parties have to respect.

A protocol is usually visualized by a **timing diagram**, such as the one shown in Figure 30 for the read cycle of a static RAM. Each row of the timing diagram shows a waveform of a signal, such as *Address, Read, Write* and *Data* in Figure 30. Each dashed vertical line designates an occurrence of an event, such as *t1, t2* through *t7.* There may be timing delays or timing constraints associated with pairs of events, indicated by an arrow annotated by x/y, where x stands for the *min* time, y stands for the *max* time. For example, the arrow between *t1* and *t3* designates a timing delay, which says that *Data* will be valid at least 10, but no more than 20 nanoseconds after *Address* is valid.

The timing information is very important for the subset of embedded systems known as real time sys-

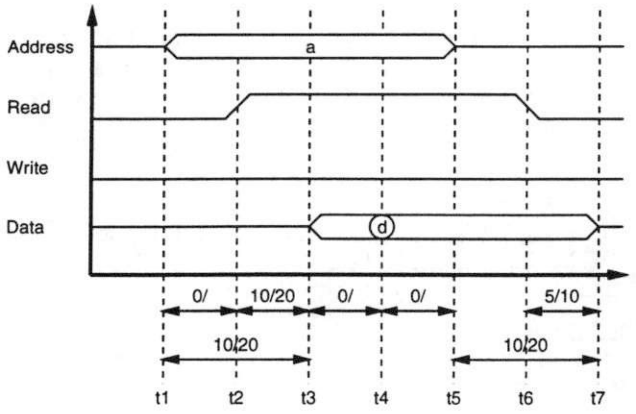


Figure 30: Timing diagram

tems, whose performance is measured in terms of how well the implementation respects the timing constraints. A favorite example of such systems would be an aircraft controller, where failure to respond to an abnormal event in a predefined timing limit will lead to disaster.

3.10 Communication

In general, systems consist of several interacting behaviors which need to communicate with each other to be cooperative. Thus a general communication model is necessary for system specification.

In traditional programming languages standard forms of communication between functions are shared variable access and parameter passing in procedure calls. These mechanisms provide communication in an abstract form. The way the communication is performed is predefined and hidden to the programmer. For example, functions communicate through global variables, which share a common memory space, or via parameter passing. In case of local procedure calls, parameter passing is implemented by exchanging information on the stack or through processor registers. In the case of remote procedure calls, parameters are passed via the complex protocol of marshaling/unmarshaling and sending/receiving data through a network.

While these mechanisms are sufficient for standard programming languages, they poorly address the needs for systems-on-a-chip descriptions, where the way the communication is performed is often custom and impossible to predefine. For example, in telecommunication applications the major task of modeling the system is actually describing custom communication procedures. Hence, it is very important for a system description language to provide the ability to redefine or extend the standard form of communication.

As an analogy, the need for a general mechanism to specify communication is the same as the need in computation to generalize operators like + and * into functions, which provide a general mechanism to define custom computation. In the absence of such a mechanism designers tend to mix the behavior responsible for communication with the behavior for computation, which results in the loss of modularity and reusability.

It follows that we need

(a) a mechanism to separate the specification of computation and communication;

(b) a mechanism to declare abstract communication functions in order to describe what they are and how they can be used;

(c) a mechanism to define a custom communication implementation which describes how the communication is actually performed.

In order to find a general communication model the **structure** of a system must be defined. A system's structure consists of a set of **blocks** which are interconnected through a set of communication **channels**. While the behavior in the blocks specifies how the computation is performed and when the communication is started, the channels encapsulate the communication implementation. In this way blocks and channels effectively separate the specification of comunication and computation.

Each block in a system contains a behavior and a set of **ports** through which the behavior can communicate. Each channel contains a set of communication functions and a set of **interfaces**. An interface declares a subset of the functions of the channel, which can be used by the connected behaviors. So while the declaration of the communication functions is given in the interfaces, the implementation of these functions is specified in the channel.

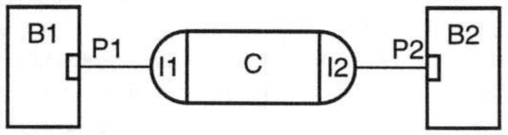


Figure 31: Communication model.

For example, the system shown in Figure 31 contains two blocks $B1$ and $B2$, and a channel C. Block $B1$ communicates with the left interface $I1$ of channel C via its port $P1$. Similarly block $B2$ accesses the right interface $I2$ of channel C through its port $P2$. Note that blocks $B1$ and $B2$ can be easily replaced by other blocks as long as the port types stay the same. Similarly channel C can be exchanged with any other channel that provides compatible interfaces.

More specifically, a channel serves as an encapsulator of a set of communication **media** in the form of variables, and a set of **methods** in the form of functions that operate on these variables. The methods specify how data is transferred over the channel. All accesses to the channel are restricted to these methods.

For example, Figure 32 shows three communication examples. Figure 32(a) shows two behaviors communicating via a shared variable M. Figure 32(b) shows a similar situation using the channel model. In fact, communication through shared memory is just a special case of the general channel model. The channel C from Figure 32(b) can be implemented as a shared memory channel as shown in Figure 33. Here the methods *receive* and *send* provide the restricted accesses to the variables *M* and *valid*.

```
channel C {
     bool  valid;
     int   M;

     int   receive( void ) {
              while( valid == 0 );
              return M;
              }
     void  send( int a ) {
              M = a;
              valid = 1;
              }
     };
```

Figure 33: Integer channel.

A channel can also be hierachical, as shown in Figure 32(c). In the example channel $C1$ implements a high level communication protocol which breaks a stream of data packets into a byte stream at the sender side, or assembles the byte stream into a packet stream at the receiver side. $C1$ in turn uses a lower level channel $C2$, for example a synchronous bus which transfers the byte stream produced by $C1$.

The adoption of mechanisms discussed above achieves information hiding, since the media and the way the communication is implemented are hidden. Also the modeling complexity is reduced, since the user only needs to make function calls to the methods. This model also encourages the separation of computation and communication, since the functionality responsible for communication can be confined in the channel specification and will not be mixed with the description used for computation.

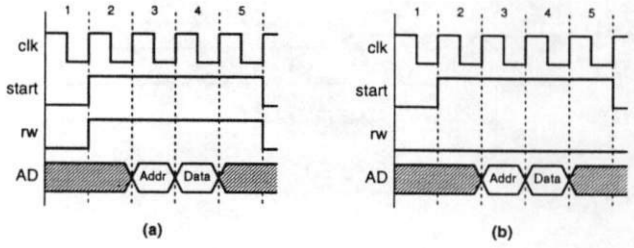


Figure 34: A simple synchronous bus protocol: (a) read cycle, (b) write cycle.

Note that the ability to describe timing is very important for channel specification. Consider, for example, the synchronous bus specification shown in Figure 34. A component using this bus can initiate a communication by asserting the *start* and *rw* signals in the first cycle, supplying the address in the second cycle, and then supplying data in the following cycles. The communication will terminate when the *start* signal is deasserted. The description of the protocol is shown in Figure 35. The description encapsulates the communication media, in this case the signals *clk, start, rw* and *AD*, and a set of methods, in this case *read_cycle* and *write_cycle*, which implement the communication protocol for reading and writing the data as described in the diagram. The call *clk.wait()* sychronizes the signal assignments with the bus clock *clk*.

3.11 Process synchronization

In a system that is conceptualized as several concurrent processes, the processes are rarely completely independent of each other. Each process may generate data and events that need to be recognized by other processes. In cases like these, when the processes exchange data or when certain actions must be performed by different processes at the same time, we need to synchronize the processes in such a way that one process is suspended until the other reaches a certain point in its execution. Common synchronization methods fall into two classifications, namely control-dependent and data-dependent schemes.

Control-dependent synchronization: In control-dependent synchronization techniques, it is the

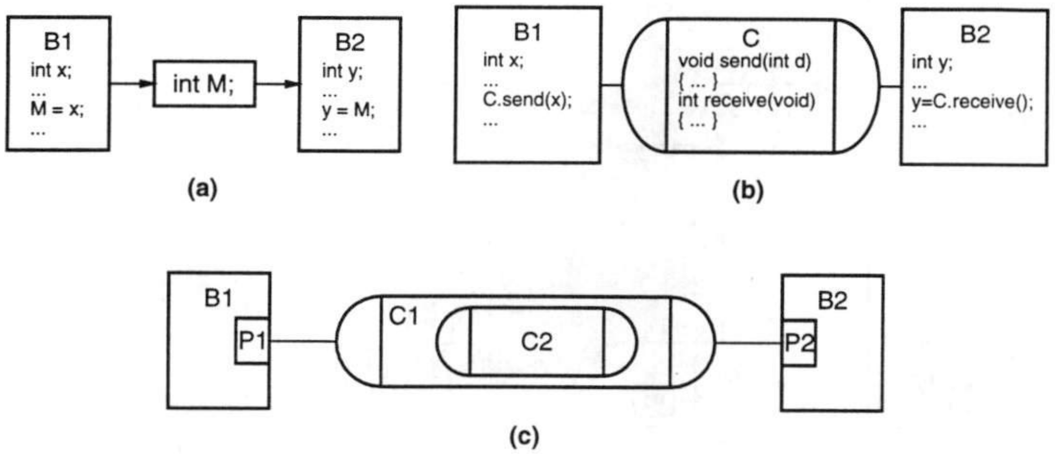


Figure 32: Examples of communication: (a) shared memory, (b) channel, (c) hierarchical channel.

```
channel  bus() {
        clock           clk;
        clocked bit     start;
        clocked bit     rw;
        clocked bit     AD;

       word    read_cycle( word addr ) {
                word     d;
                start = 1, rw = 1,        clk.wait();
                AD = addr,                clk.wait();
                d = AD,                   clk.wait();
                start = 0, rw = 0,        clk.wait();
                return d;
                }
        void   write_cycle( word a, word d ) {
                start = 1, rw = 0,        clk.wait();
                AD = addr,                clk.wait();
                AD = d,                   clk.wait();
                start = 0,                clk.wait();
                }
        }
```

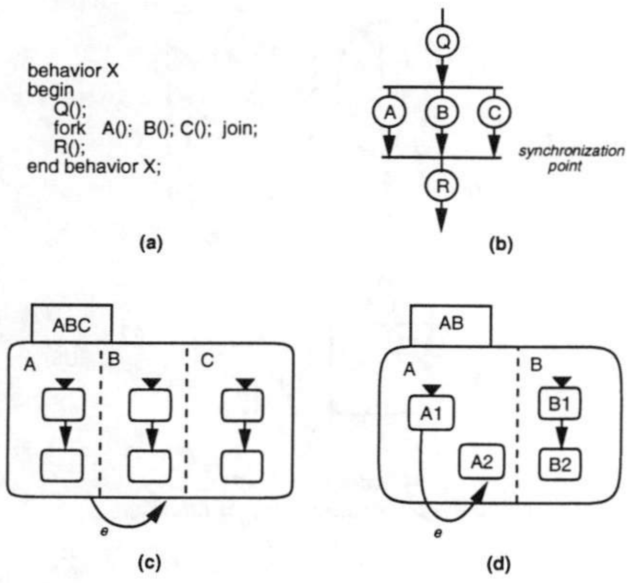
Figure 35: Protocol description of the synchronous bus protocol.

control structure of the behavior that is responsible for synchronizing two processes in the system. For example, the *fork-join* statement introduced in Section 3.5 is an instance of such a control construct. Figure 36(a) shows a behavior *X* which forks into three concurrent subprocesses, *A, B* and *C*. In Figure 36(b) we see how these distinct execution streams for the behavior *X* are synchronized by a *join* statement, which ensures that the three processes spawned by the fork statement are all complete before *R* is executed. Another example of control-dependent synchronization is the technique of **initialization**, in which processes are synchronized to their initial states either the first time the system is initialized, as is the case with most HDLs, or during the execution of the processes. In the Statecharts [DH89] of Figure 36(c), we can see how the event *e*, associated with a transition arc that reenters the boundary of *ABC*, is designed to synchronize all the orthogonal states *A, B* and *C* into their default substates. Similarly, in Figure 36(d), event *e* causes *B* to initialize to its default substate *B1* (since *AB* is exited and then reentered), at the same time transitioning *A* from *A1* to *A2*.

Data-dependent synchronization: In addition to these techniques of control-dependent synchronization, processes may also be synchronized by means of one of the methods for interprocess communication: shared memory or message passing as mentioned in Section 3.10.

Shared-memory based synchronization works by making one of the processes suspend until the other process has updated the shared memory with an appropriate value. In such cases, the variable in the shared memory might represent an event, a data value or the status of another process in the system, as is illustrated in Figure 37 using the Statecharts language.

Synchronization by common event requires one process to wait for the occurrence of a specific event, which can be generated externally or by another process. In Figure 37(a), we can see how event *e* is used for synchronizing states *A* and *B* into substates *A2* and *B2*, respectively. Another method is that of **synchronization by common variable**, which requires one of the processes to update the variable with a suitable value. In Figure 37(b), *B* is synchronized



```
behavior X
begin
    Q();
    fork  A();  B();  C();  join;
    R();
end behavior X;
```

Figure 36: Control synchronization: (a) behavior X with a fork-join, (b) synchronization of execution streams by join statement, (c) and (d) synchronization by initialization in Statecharts. (†)

into state *B2* when we assign the value "1" to variable *x* in state *A2*.

Still another method is **synchronization by status detection**, in which a process checks the status of other processes before resuming execution. In a case like this, the transition from *A1* to *A2* precipitated by event *e*, would cause *B* to transition from *B1* to *B2*, as shown in Figure 37(c).

3.12 SpecC+ Language description

In this section, we will present an example of a specification language called SpecC+, which was specifically developed to capture directly a conceptual model possessing all the above discussed characteristics.

The SpecC+ view of the world is a hierarchical network of **actors**. Each actor possesses

(a) a set of ports through which the actor communicates with the environment;

(b) a set of state variables;

(c) a set of communication channels;

(d) a behavior which defines how the actor will change its state and perform communication through its ports when it is invoked.

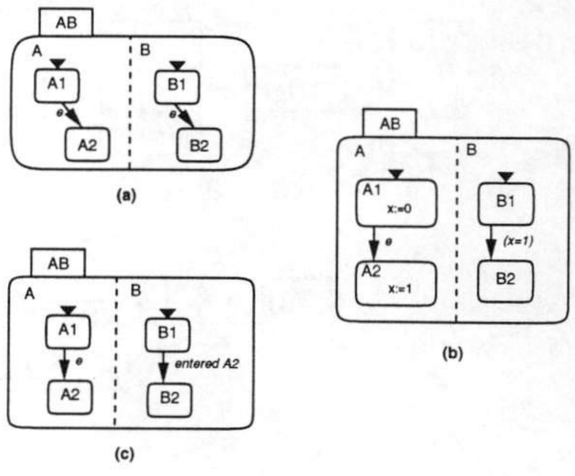
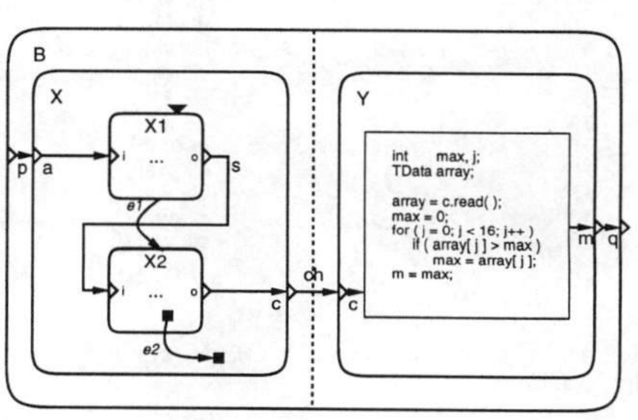


Figure 37: Data-dependent synchronization in Statecharts: (a) synchronization by common event, (b) synchronization by common data, (c) synchronization by status detection. (†)

Figure 39 shows the textual representation of the example in Figure 38, where an actor is represented by a rectangular box with curved corners.



```
int    max, j;
TData array;

array = c.read( );
max = 0;
for ( j = 0; j < 16; j++ )
    if ( array[ j ] > max )
        max = array[ j ];
m = max;
```

Figure 38: A graphical SpecC+ specification example.

There is an *actor* construct which capture all the information for an actor. An *actor* construct looks like a C++ class which exports an *main* method. The ports are declared in the parameter list. The state variable, channels and child actor instances are declared as typed variables, and the behavior is specified by the methods, or functions start from *main*. Actor construct can be used as a type to instantiate actor instances.

SpecC+ supports both **behavioral hierarchy** and

structural hierarchy in the sense that it captures a system as a hierarchy of actors. Each actor is either a composite actor or a leaf actor.

Composite actors are decomposed hierarchically into a set of child actors. For structural hierarchy, the child actors are interconnected via the communication channels by child actor instantiation statements, similar to component instantiation in VHDL. For example, actor *X* is instantiated in line 57 of Figure 39 by mapping its port *a* and *c* to the ports (*p*) and communication channels (*ch*) defined in its parent actor *B*. For behavioral hierarchy, the child actors can either be concurrent, in which case all child actors are active whenever the parent actor is active, or can be sequential, in which case the child actors are only active one at a time. In Figure 38, actors *B* and *X* are composite actors. Note that while *B* consists of concurrent child actors *X* and *Y*, *X* consists of sequential child actors *X1* and *X2*.

Leaf actors are those that exist at the bottom of the hierarchy whose functionality is specified with imperative programming constructs. In Figure 38, for example, *Y* is a leaf actor.

SpecC+ also supports **state transitions**, in the sense that we can represent the sequencing between child actors by means of a set of transition arcs. In this language, an arc is represented as a 3-tuple $< T,\ C,\ N >$, where *T* represents the type of transition, *C* represents the condition triggering the transition, and *N* represents the next actor to which control is transferred by the transition. If no condition is associated with the transition, it is assumed to be "true" by default.

SpecC+ supports two types of transition arcs. A **transition-on-completion arc** (TOC) is traversed whenever the source actor has completed its computation and the associated condition evaluates as true. A leaf actor is said to have completed when its last statement has been executed. A sequentially decomposed actor is said to be complete only when it makes a transition to a special predefined completion point, indicated by the name *complete* in the next-actor field of a transition arc. In Figure 38, for example, we can see that actor *X* completes only when child actor *X2* completes and control flows from *X2* to the *complete* point when *cond2* is true (as specified by the arc $< TOC,\ cond2,\ complete >$ in line 36 of Figure 39). Finally, a concurrently decomposed actor is said to be completed when all of its child actors have completed. In Figure 38, for example, actor *B* completes when all the concurrent child actors *X* and *Y* have completed.

Unlike the TOC arc, a **transition-immediate-**

```
1 typedef int       TData[16];
2
3 interface IData( void ) {
4       TData  read( void );
5       void   write( TData d );
6       };
7
8 channel CData( void ) implements IData {
9       bool        valid;
10      event       s;
11      TData       storage;
12
13      TData  read( void ) {
14             if( valid ) s.wait();
15             return storage;
16             }
17      void   write( TData d ) {
18             storage=d; valid = 1; s.notify();
19             }
20      };
21
22 actor X1( in TData i, out TData o ) { ... };
23 actor X2( in TData i, IData o ) {
24      void main( void ) {
25             ...
26             o.write(...);
27             }
28      };
29
30 actor    X( in int a, IData c ) {
31      TData  s;
32      X1     x1( a, s );
33      X2     x2( s, c );
34
35      psm     main( void ) {
36             x1 : ( TI, cond1, x2 );
37             x2 : ( TOC, cond2, complete );
38             }
39      };
40
41 actor Y ( IData c, out int m ) {
42      void    main( void ) {
43             int     max, j;
44             TData          array;
45
46             array = c.read();
47             max = 0;
48             for( j = 0; j < 16; j ++ )
49                    if( array[j] > max )
50                           max = array[j];
51             m = max;
52             }
53      };
54
55 actor B( in TData p, out int q ) {
56      CData  ch;
57      X      x( p, ch );
58      Y      y( ch, q );
59
60      csp     main( void ) {
61             par { x.main(); y.main(); }
62             }
63      };
```

Figure 39: A textual SpecC+ specification example.

ly arc (TI) is traversed instantaneously whenever the associated condition becomes true, regardless of whether the source actor has or has not completed its computation. For example, in Figure 38, the arc $< TI,\ cond1,\ x2 >$ terminates *X1* whenever *cond1* is true and transfers control to actor *X2*. In other words, a TI arc effectively terminates all lower level child actors of the source actor.

Transitions are represented in Figure 38 with directed arrows. In the case of a sequentially-decomposed actor, an inverted bold triangle points to the first child actor. An example of such an initial child actor is *X1* of actor *X*. The completion of sequentially decomposed actors is indicated by a transition arc pointing to the completion point, represented as a bold square within the actor. Such a completion point is found in actor *X* (transition from *X2* labeled *e2*). TOC arcs originate from a bold square inside the source child actor, as does the arc labeled *e2*. TI arcs, in contrast, originate from the perimeter of the source child actor, as does the arc labeled *e1*.

SpecC+ supports both **data-dependent synchronization** and **control-dependent synchronization**. In the first method, actors can synchronize using common event. For example, in Figure 38, actor *Y* is the consumer of the data produced by actor *X* via channel *c*, which is of type *CData* Figure 39. In the implementation of *CData* at line 8 of Figure 39, an event *s* is used to make sure *Y* can get valid data from *X*: the *wait* function over *s* will suspend Y if the data is not ready. In the second method, we could use a TI arc from actor *B* back to itself in order to synchronize all the concurrent child actors of *B* to their initial states. Furthermore, the fact that *X* and *Y* are concurrent actors enclosed in *B* automatically implements a barrier, since by semantics, *B* finishes when both the execution of *X* and *Y* are finished.

Communication in SpecC+ is achieved through the use of communication channels. Channels can be primitive channels such as variables and signals (like variable *s* of actor *X* in Figure 38), or complex channels such as object channels (like variable *ch* in Figure 38), which directly supports the hierarchical communication model discussed in Section 3.10.

The specification of an object channel is separated in the *interface* declaration and the *channel* definition, each of which can be used as data types for channel variables. The interface defines a set of function prototype declarations without the actual function body. For example, the interface *IData* in Figure 38 defines the function prototypes *read* and *write*. The channel encapsulates the communication media and provides a set of function implementations. For example, the channel *CData* encapsulates media *s* and *storage* and an implementation of methods *read* and *write*. The interface and the channel are related by the *implements* keyword. A channel related to an interface in this way is said to implement this interface, meaning the channel is obligated to implement the set of functions prescribed by the interface. For example, *CData* has to implement *read* and *write* since they appear in *IData*. It is possible that several channels can implement the same interface, which implies that they can provide different implementations of the same set of functions.

Interfaces are usually used as port data types in port declarations of an actor (as port *c* of actor Y at line 41 of Figure 39). A port of one interface type will be bound to a particular channel which implements such an interface during actor instantiation. For example, port *c* of actor *Y* is mapped to channel *c* of actor *B*, when actor Y is instantiated.

The fact that a port of interface type can be bound to a real channel until actor instantiation is called **late binding**. Such a late binding mechanism helps to improve the reusability of an actor description, since it is possible to plug in any channel as long as they implement the same interface.

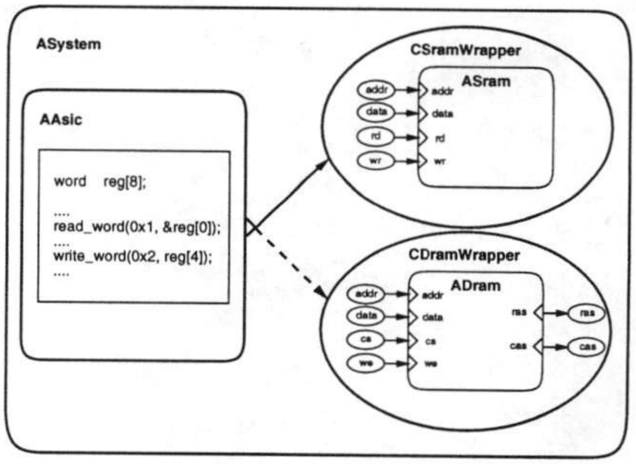


Figure 40: Component wrapper specification.

Consider, the example in Figure 40 as specified in Figure 41.

The system described in this example contains an ASIC (actor *AAsic*) talking to a memory. The interface *IRam* specifies the possible transactions to access memories: read a word via *read_word* and write a word via *write_word*. The description of *AAsic* can

```
1  interface IRam( void ) {
2        void      read_word( word a, word *d );
3        void      write_word( word a, word d );
4        };
5
6  actor AAsic( IRam ram ) {
7        word    reg[8];
8
9        void      main( void ) {
10                 ...
11                 ram.read_word( 0x0001, &reg[0] );
12                 ...
13                 ram.write_word( 0x0002, reg[4] );
14                 }
15       };
16
17 actor ASram( in signal<word> addr,
18       inout signal<word> data,
19       in signal<bit> rd, in signal<bit> wr ) {
20       ...
21       };
22
23 actor ADram( in signal<word> addr,
24       inout signal<word> data,
25       in signal<bit> cs, in signal<bit> we,
26       out signal<bit> ras, out signal<bit> cas ) {
27       ...
28       };
29
30 channel CSramWrapper( void ) implements IRAM {
31       signal<word>  addr, data;     // address, data
32       signal<bit>   rd, wr;         // read/write select
33       ASram         sram( addr, data, rd, wr );
34
35       void     read_word( word a, word *d ) { ... }
36       void     write_word( word a, word d ) { ... }
37       ...
38       };
39
40 channel CDramWrapper( void ) implements IRam {
41       signal<word>     addr, data; // address, data
42       signal<bit>      cs, we;  // chip select, write enable
43       signal<bit>      ras, cas; // row, col address strobe
44       ADram            sram( addr, data, cs, we, ras, cas );
45
46       void     read_word( word a, word *d ) { ... }
47       void     write_word( word a, word d ) { ... }
48       ...
49       };
50
51 actor ASystem( void ) {
52       CSramWrapper   ram;         // can be replaced by
53       // CDramWrapper ram;        // this declaration
54       AAsic          asic( ram );
55
56       void     main( void ) { ... }
57       };
```

Figure 41: Source code of the component wrapper specification.

use *IRam* as its port so that its behavior can make function calls to methods *read_word* and *write_word* without knowing how these methods are exactly implemented. There are two types of memories available in the library, represented by actors *ASram* and *ADram* respectively, the descriptions of which provide their behavioral models. Obviously, the static RAM *ASram* and dynamic RAM *ADram* have different pins and timing protocols to access them, which can be encapsulated with the component actors themselves in channels called wrappers, as *CSramWrapper* and *CDramWrapper* in Figure 40. When the actor *AAsic* is instantiated in actor *ASystem* (lines 52 and 53 in Figure 41), the port *IRam* will be resolved to either *CSramWrapper* or *CDramWrapper*.

The improvement of reusability of this style of specification is two fold: first, the encapsulation of communication protocols into the channel specification make these channels highly reusable since they can be stored in the library and instantiated at will. If these channel descriptions are provided by component vendors, the error-prone effort spent on understanding the data sheets and interfacing the components can be greatly relieved. Secondly, actor descriptions such as *AAsic* can be stored in the library and easily reused without any change subject to the change of other components with which it interfaces.

It should be noted that while methods in an actor represent the behavior of itself, the methods of a channel represent the behavior of their callers. In other words, when the described system is implemented, the methods of the channels will be **inlined** into the connected actors. When a channel is inlined, the encapsulated media get exposed and its methods are moved to the caller. In the case of a wrapper, the encapsulated actors also get exposed.

Figure 42 shows some typical configurations. In Figure 42(a), two synthesizable components A and B (eg. actors to be implemented on an ASIC) are interconnected via a channel C, for example, a standard bus. Figure 42(b) shows the situation after inlining. The methods of the channel C are inserted into the actors and the bus wires are exposed. In Figure 42(c) a synthesizable component A communicates with a fixed component B (eg. an off-the-shelf component) through a wrapper W. When W is inlined, as shown in Figure 42(d), the fixed component B and the signals get exposed. In Figure 42(e) again a synthesizable component A communicates with a fixed component B using a predefined protocol, that is encapsulated in the channel C. However, B has its own built-in protocol, which is encapsulated in the wrapper W. A

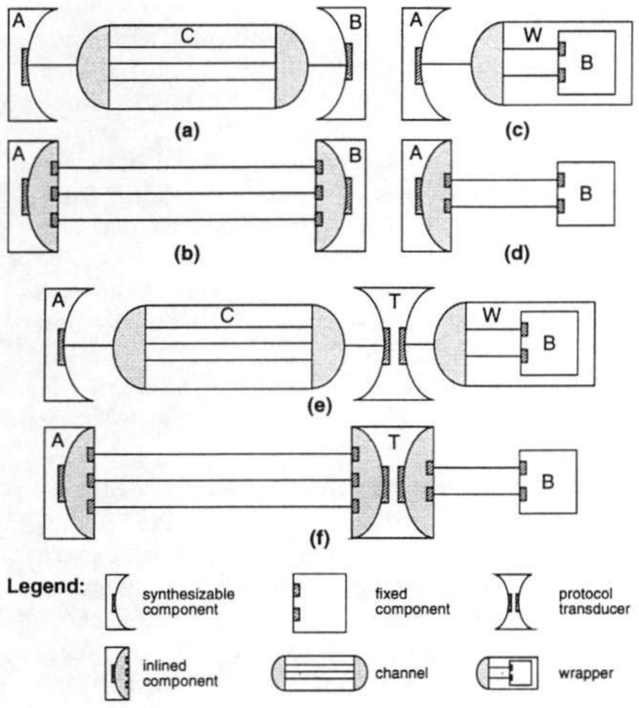


Figure 42: Common configurations before and after channel inlining: (a)/(b) two synthesizable actors connected by a channel, (c)/(d) synthesizable actor connected to a fixed component, (e)/(f) protocol transducer.

protocol transducer T has to be inserted between the channel C and the wrapper W in order to translate all transactions between the two protocols. Figure 42(f) shows the final situation, when both channels C and W are inlined.

SpecC+ supports the specification of **timing** explicitly and distinguishes two types of timing specifications, namely **timing constraints** and **timing delays**, as discussed in Section 3.9. At the specification level timing constraints are used to specify time limits that have to be satisfied. At the implementation level computational delays have to be noted.

Consider, for example, the timing diagram of the read protocol for a SRAM, as shown earlier in Figure 30. The protocol visualized by the timing diagram can be used to define the *read_word* method of the SRAM channel above (line 35 in Figure 41). The code segment in Figure 43 shows the specification of the read access to the SRAM.

The *do-timing* statement effectively describes all information contained in the timing diagram. The first part lists all the events of the diagram. Events are specified as a label and its associated piece of code, which describes the change on signal values. The second part is a list of *range* statements, which specify the timing constraints or timing delays using the 4-tuples as described in Section 3.9.

```
void read_word( word a, word *d ) {
    do {
        t1: { addr = a; }
        t2: { rd = 1; }
        t3: { }
        t4: { *d = data; }
        t5: { addr.disconnect(); }
        t6: { rd = 0; }
        t7: { break}; }
        }
    timing {
        range( t1; t2; 0; );
        range( t1; t3; 10; 20 );
        range( t2; t3; 10; 20 );
        range( t3; t4; 0; );
        range( t4; t5; 0; );
        range( t5; t7; 10; 20 );
        range( t6; t7; 5; 10 );
        }
};
```

Figure 43: Timing specification of the SRAM read protocol.

This style of timing description is used at the specification level. In order to get an executable model of the protocol, scheduling has to be performed for each *do-timing* statement. Figure 44 shows the implementation of the *read_word* method which follows an ASAP scheduling, where all timing constraints are replaced by delays, which are specified using the *waitfor* function.

```
void read_word( word a, word *d ) {
    addr = a;
    rd = 1;
    waitfor( 10 );
    *d = data;
    addr.disconnect();
    rd = 0;
    waitfor( 10 );
};
```

Figure 44: Timing implementation of the SRAM read protocol.

In this section we presented the characteristics most commonly found in modeling systems and discussed their usefulness in capturing system behavior. Also, we presented SpecC+, an example of a specification language, which was specifically developed to capture directly these characteristics. The next section presents a generic codesign methodology based on the

SpecC+ language.

4 Generic codesign methodology

As shown in Figure 45, codesign usually starts from a formal specification which specifies the functionality as well as the performance, power, cost, and other constraints of the intended design. During the codesign process, the designer will go through a series of well-defined design steps which will eventually map the functionality of the specification to the target architecture. These design steps include allocation, partitioning, scheduling and communication synthesis, which form the synthesis flow of the methodology.

The result of the synthesis flow will then be fed into the backend tools, shown in the lower part of Figure 45. Here, a compiler is used to implement the functionality mapped to processors, a high level synthesizer is used to map the functionality mapped to ASICs, and a interface synthesizer is used to implement the functionality of interfaces.

During each design step, the design model will be statically analyzed to estimate certain quality metrics and how they satisfy the constraints. This design model will also be used to generate a simulation model, which is used to validate the functional correctness of the design. In case the validation fails, a debugger can be used to locate and fix the errors. Simulation is also used to collect profiling information which in turn will improve the accuracy of the quality metrics estimation. This set of tasks forms the analysis and validation flow of the methodology (see Figure 45).

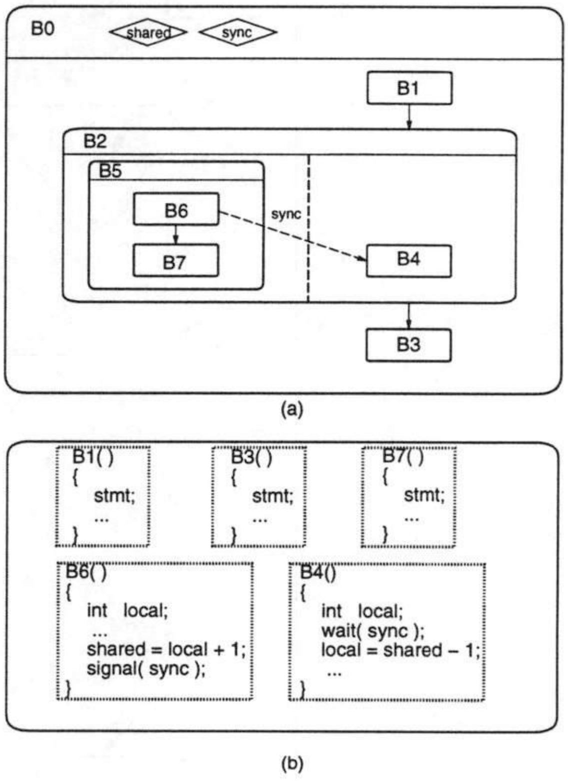
The following sections describe the tasks of the generic methodology in more detail.

4.1 System specification

We have described the characteristics needed for specifying systems in Section 3.1. The system specification should describe the functionality of the system without premature engagement in the implementation. It should be made logically as close as possible to the conceptual model of the system so that it is easy to be maintained and modified. It should also be executable so that the specified functionality is verifiable.

The behavior model in Section 3.12 makes it a good candidate since it is a simple model which meets these requirements.

In the example shown in Figure 46, the system itself is specified as the top behavior B0, which contains



```
B1()
{
  stmt;
  ...
}

B3()
{
  stmt;
  ...
}

B7()
{
  stmt;
  ...
}

B6()
{
  int local;
  ...
  shared = local + 1;
  signal( sync );
}

B4()
{
  int local;
  wait( sync );
  local = shared - 1;
  ...
}
```

Figure 46: Conceptual model of specification: (a) control-flow view, (b) atomic behaviors.

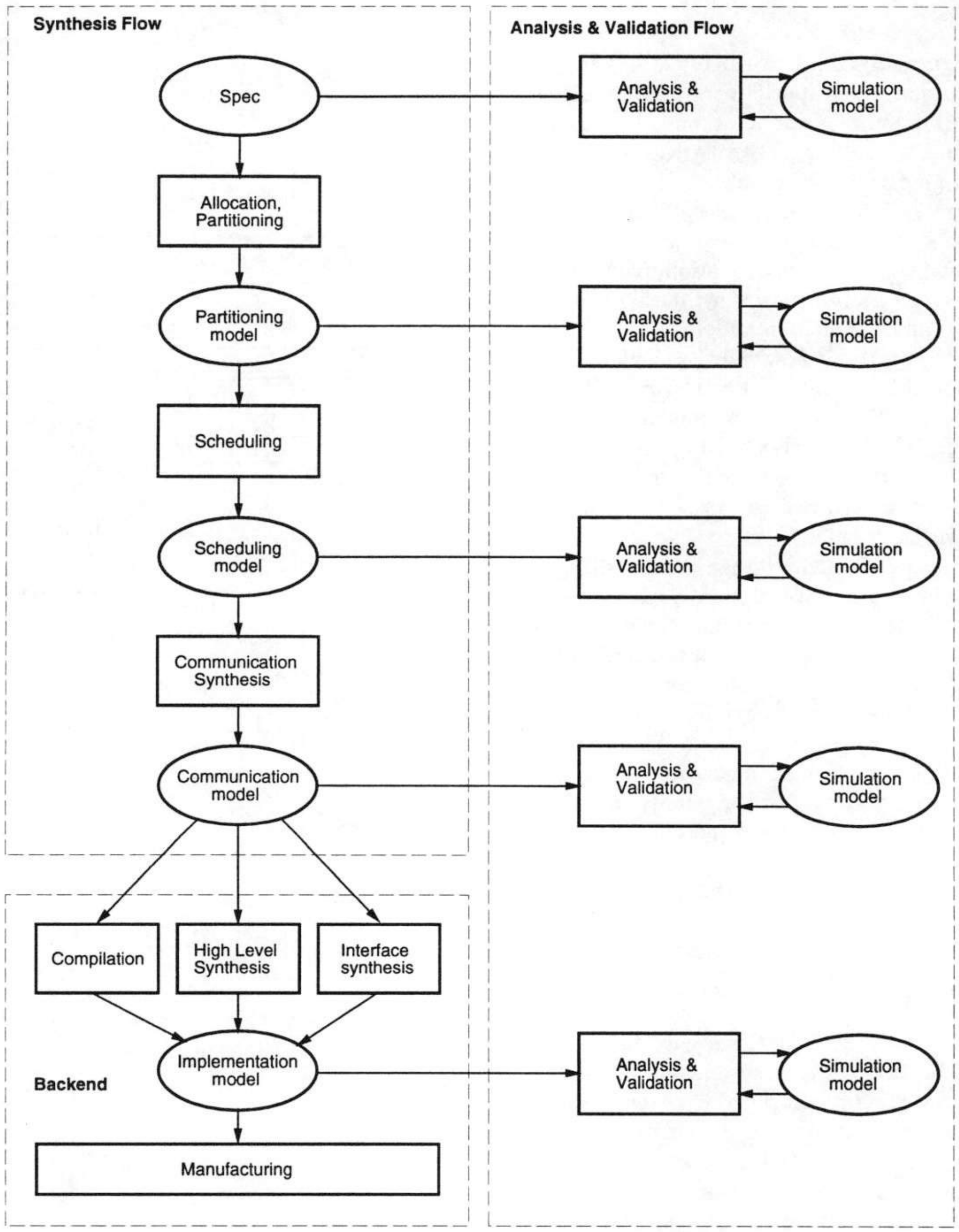


Figure 45: Generic methodology.

an integer variable *shared* and a boolean variable *sync*. There are three child behaviors, B1, B2, B3, with sequential ordering, in behavior B0. While B1 and B3 are atomic behaviors specified by a sequence of imperative statements, B2 is a composite behavior consisting of two concurrent behaviors B4 and B5. B5 in turn consists of B6 and B7 in sequential order. While most of the actual behavior of an atomic behavior is omitted in the figure for space reasons, we do show a producer-consumer example relevant for later discussion: B6 computes a value for variable *shared*, and B4 consumes this value by reading *shared*. Since B6 and B4 are executed in parallel, they synchronize with the variable *sync* using signal/wait primitives to make sure that B4 accesses *shared* only after B6 has produced the value for it.

4.2 Allocation

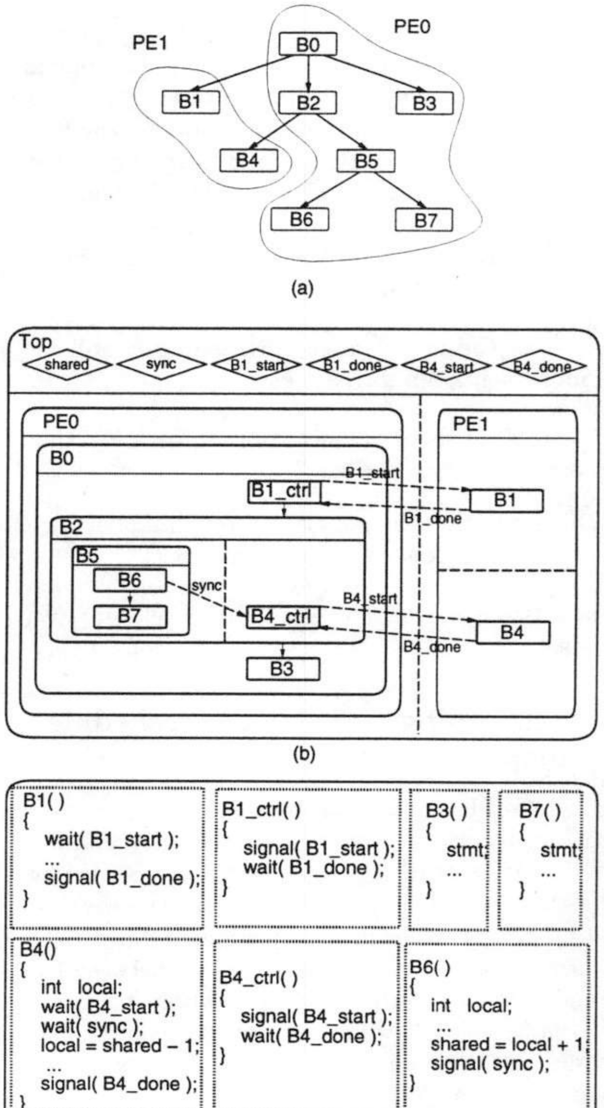
Given a library of system components such as processors, memories and custom IP modules, the task of allocation is defined as the selection of the type and number of these components, as well as the determination of their interconnection, in such a way that the functionality of the system can be implemented, the constraints satisfied, and the objective cost function minimized. The result of the allocation task can be a customization of the generic architecture discussed in Section 2.2. Allocation is usually carried out manually by designers and is the starting point of the **design exploration** process.

4.3 Partitioning and the model after partitioning

The task of partitioning defines the mapping between the set of behaviors in the specification and the set of allocated components in the selected architecture. The quality of such a mapping is determined by how well the result can meet the design constraints and minimize the design cost.

The system model after partitioning must reflect the partitioning decision and must be complete in order for us to perform validation. More specifically, the partitioned model refines the specification in the following way:

(a) An additional level of hierarchy is inserted which describes the selected architecture. Figure 47 shows the partitioned model of the example in Figure 46. Here, the added level of hierarchy includes two concurrent behaviors, PE0 and PE1.



```
B1( )
{
   wait( B1_start );
   ...
   signal( B1_done );
}
```

```
B1_ctrl( )
{
   signal( B1_start );
   wait( B1_done );
}
```

```
B3( )
{
   stmt;
   ...
}
```

```
B7( )
{
   stmt;
   ...
}
```

```
B4()
{
   int  local;
   wait( B4_start );
   wait( sync );
   local = shared – 1;
   ...
   signal( B4_done );
}
```

```
B4_ctrl( )
{
   signal( B4_start );
   wait( B4_done );
}
```

```
B6( )
{
   int  local;
   ...
   shared = local + 1;
   signal( sync );
}
```

Figure 47: Conceptual model after partitioning: (a) partitioning decision, (b) conceptual model, (c) atomic behaviors.

(b) In general, controlling behaviors are needed and must be added for child behaviors assigned to different PEs than their parents. For example, in Figure 47, behavior B1_ctrl and B4_ctrl are inserted in order to control the execution of B1 and B4, respectively.

(c) In order to maintain the functional equivalence between the partitioned model and the original specification, synchronization between PEs is inserted. In Figure 47 synchronization variables , *B1_start, B1_done, B4_start, B4_done* are added so that the execution of B1 and B4, which are assigned to PE1, can be controlled by their controlling behaviors B1_ctrl and B4_ctrl through inter-PE synchronization.

However, the model after partitioning is still far from implementation for two reasons:

(a) There are concurrent behaviors in each PE that have to be serialized;

(b) Different PEs communicate through global variables which have to be localized.

These issues will be addressed in the following two sections.

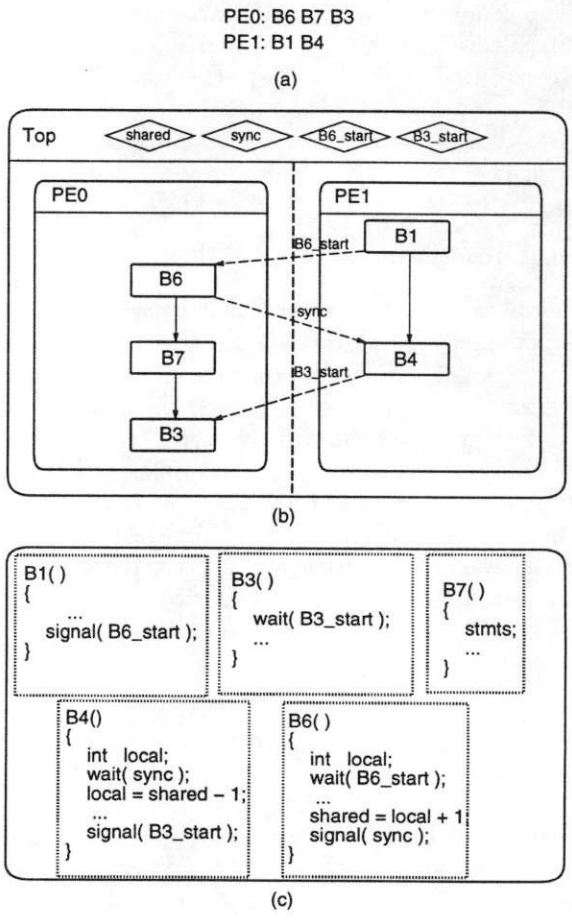
4.4 Scheduling and the scheduled model

Given a set of behaviors and possibly a set of performance constraints, the scheduling task determines a total order in invocation time of the behaviors running on the same PE, while respecting the partial order imposed by dependencies in the functionality as well as minimizing the synchronization overhead between the PEs and context switching overhead within the PEs.

Depending upon how much information on the partial order of the behaviors is available at compile time, there are different strategies for scheduling.

In one extreme, where ordering information is unknown until runtime, the system implementation often relies on the dynamic scheduler of an underlying runtime system. In this case, the model after scheduling is not much different from the model after partitioning, except that a runtime system is added to carry out the scheduling. This strategy suffers from context switching overhead when a running task is blocked and a new task is scheduled.

On the other extreme, if the partial order is completely known at compile time, a static scheduling strategy can be taken, provided a good estimation on



```
B1( )
{
   ...
   signal( B6_start );
}

B3( )
{
   wait( B3_start );
   ...
}

B7( )
{
   stmts;
   ...
}

B4()
{
   int  local;
   wait( sync );
   local = shared - 1;
   ...
   signal( B3_start );
}

B6( )
{
   int  local;
   wait( B6_start );
   ...
   shared = local + 1;
   signal( sync );
}
```

Figure 48: Conceptual model after scheduling: (a) scheduling decision, (b) conceptual model, (c) atomic behaviors.

the execution time of each behavior can be obtained. This strategy eliminates the context switching overhead completely, but may suffer from inter-PE synchronization especially in the case of inaccurate performance estimation. On the other hand, the strategy based on dynamic scheduling does not have this problem because whenever a behavior is blocked for inter-PE synchronization, the scheduler will select another to execute. Therefore the selection of the scheduling strategy should be based on the trade-off between context switching overhead and CPU utilization.

The model generated after static scheduling will remove the concurrency among behaviors inside the same PE. As shown in Figure 48, all child behaviors in PE0 are now sequentially ordered. In order to maintain the partial order across the PEs, synchronization between them must be inserted. For example, B6 is synchronized by *B6_start*, which will be asserted by B1 when it finishes.

Note that B1_ctrl and B4_ctrl in the model after partitioning are eliminated by the optimization carried out by static scheduling. It should also be mentioned that in this section we define the tasks, rather than the algorithms of codesign. Good algorithms are free to combine several tasks together. For example, an algorithm can perform the partitioning and static scheduling at the same time, in which case intermediate results, such as B1_ctrl and B4_ctrl, are not generated at all.

4.5 Communication synthesis and the communication model

Up to this stage, the communication and synchronization between concurrent behaviors are accomplished through shared variable accesses. The task of this stage is to resolve the shared variable accesses into an appropriate inter-PE communication scheme at implementation level. Several communication schemes exist:

(a) The designer can choose to assign a shared variable to a shared memory. In this case, the communication synthesizer will determine the location of the variables assigned to the shared memory. Given the location of the shared variables, the synthesizer then has to change all accesses to the shared variables in the model into statements that read or write to the corresponding addresses. The synthesizer also has to insert interfaces for the PEs and shared memories to adapt to different protocols on the buses.

(b) The designer may also choose to assign a shared variable to the local memory of one particular PE. In this case, accesses to this shared variable in models of other PEs have to be changed into function calls to message passing primitives such as send and receive. Again, interfaces have to be inserted to make the message-passing possible.

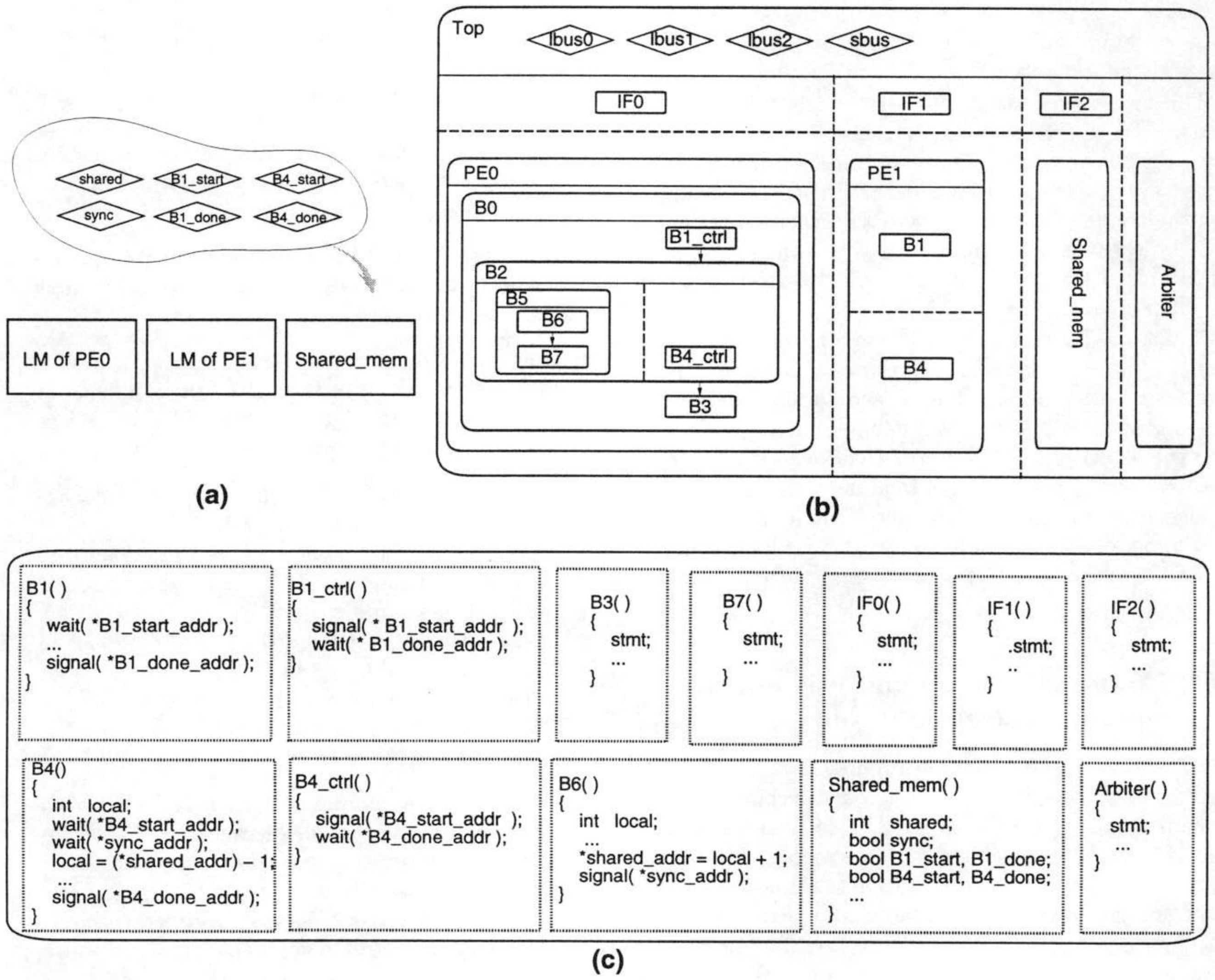
(c) Another option is to maintain a copy of the shared variable in all the PEs that access it. In this case, all the statements that perform a write on this variable have to be modified to implement a broadcasting scheme so that all the copies of the shared variable remain consistent. Necessary interfaces also need to be inserted to implement the broadcasting scheme.

The generated model after communication synthesis, as shown in Figure 49, is different from previous models in the following way:

(a) New behaviors for interfaces, shared memories and arbiters are inserted at the highest level of the hierarchy. In Figure 49 the added behaviors are *IF0, IF1, IF2, Shared_mem, Arbiter.*

(b) The shared variables from the previous model are all resolved. They either exist in shared memory or in local memory of one or more PEs. The communication channels of different PEs now become the local buses and system buses. In Figure 49, we have chosen to put all the global variables in *Shared_mem*, and hence all the global declarations in the top behavior are moved to the behavior *Shared_mem.* New global variables in the top behavior are the buses *lbus0, lbus1, lbus2, sbus.*

(c) If necessary, a communication layer is inserted into the runtime system of each PE. The communication layer is composed of a set of inter-PE communication primitives in the form of driver routines or interrupt service routines, each of which contain a stream of I/O instructions, which in turn talk to the corresponding interfaces. The accesses to the shared variables in the previous model are transformed into function calls to these communication primitives. For the simple case of Figure 49, the communication synthesizer will determine the addresses for all global variables, for example, *shared_addr* for variable *shared*, and all accesses to the variables are appropriately transformed. The accesses to the variables are exchanged with reading and writing to the corresponding addresses. For example, *shared = local + 1* becomes **shared_addr = local+1.*



```
B1( )
{
  wait( *B1_start_addr );
  ...
  signal( *B1_done_addr );
}
```

```
B1_ctrl( )
{
  signal( * B1_start_addr );
  wait( * B1_done_addr );
}
```

```
B3( )
{
   stmt;
   ...
}
```

```
B7( )
{
   stmt;
   ...
}
```

```
IF0( )
{
   stmt;
   ...
}
```

```
IF1( )
{
   .stmt;
   ..
}
```

```
IF2( )
{
   stmt;
   ...
}
```

```
B4()
{
   int  local;
   wait( *B4_start_addr );
   wait( *sync_addr );
   local = (*shared_addr) - 1;
   ...
   signal( *B4_done_addr );
}
```

```
B4_ctrl( )
{
   signal( *B4_start_addr );
   wait( *B4_done_addr );
}
```

```
B6( )
{
   int  local;
   ...
   *shared_addr = local + 1;
   signal( *sync_addr );
}
```

```
Shared_mem( )
{
   int  shared;
   bool sync;
   bool B1_start, B1_done;
   bool B4_start, B4_done;
   ...
}
```

```
Arbiter( )
{
   stmt;
   ...
}
```

Figure 49: Conceptual model after communication synthesis: (a) communication synthesis decision, (b) conceptual model, (c) atomic behaviors.

4.6 Analysis and validation flow

Before each design step, which takes an input design model and generates a more detailed design model, the input design model has to be functionally verified. It also needs to be analyzed, either statically, or dynamically with the help of the simulator or estimator, in order to obtain an estimation of the quality metrics, which will be evaluated by the synthesizer to make good design decisions. This motivates the set of tools to be used in the analysis and validation flow of the methodology. An example of such a tool set consists of

(a) a static analyzer,

(b) a simulator,

(c) a debugger,

(d) a profiler, and

(e) a visualizer.

The **static analyzer** associates each behavior with quality metrics such as program size and program performance in case it is to be implemented as software, or metrics of hardware area and hardware performance if it is to be implemented as an ASIC. To achieve a fast estimation with satisfactory accuracy, the analyzer relies on probabilistic techniques and the knowledge of backend tools such as compiler and high level synthesizer.

The **simulator** serves the dual purpose of functional validation and dynamic analysis. Simulation is achieved by generating an executable simulation model from the design model. The simulation model runs on a simulation engine, which in the form of runtime library, provides an implementation for the simulation tasks such as simulation time advance and synchronization among concurrent behaviors.

Simulation can be performed at different accuracy levels. Common accuracy models are functional, cycle based, and discrete event simulation. A functionally accurate simulation compiles and executes the design model directly on a host machine without paying special attention to simulation time. A clock cycle accurate simulation executes the design model in a clock by clock fashion. A discrete event simulation incorporates a even more sophisticated timing model of the components, such as gate delay. Obviously there is a trade-off between simulation accuracy and simulator execution time.

It should be noted that, while most design methodologies adopt a fixed accuracy simulation at each design stage, applying a mixed accuracy model is also possible. For example, consider a behavior representing a piece of software that performs some computation and then sends the result to an ASIC. While the part of the software which communicates with the ASIC needs to be simulated at cycle level so that tricky timing problems become visible, it is not necessary to simulate the computation part with the same accuracy.

The **debugger** renders the simulation with break point and single step ability. This makes it possible to examine the state of a behavior dynamically. A **visualizer** can graphically display the hierarchy tree of the design model as well as make dynamic data visible in different views and keep them synchronized at all times. All these efforts are invaluable in quickly locating and fixing the design errors.

The **profiler** is a good complement of a static analyzer for obtaining dynamic information such as branching probability. Traditionally, it is achieved by instrumenting the design description, for example, by inserting a counter at every conditional branch to keep track of the number of branch executions.

4.7 Backend

At the stage of the backend, as shown in the lower part of Figure 45, the leaf behaviors of the design model will be fed into different tools in order to obtain their implementations. If the behavior is assigned to a standard processor, it will be fed into a compiler for this processor. If the behavior is to be mapped on an ASIC, it will be synthesized by a high level synthesis tool. If the behavior is an interface, it will be fed into an interface synthesis tool.

A **compiler** translates the design description into machine code for the target processor. A crucial component of a compiler is its code generator, which emits machine code from the intermediate representation generated by the parser part of the compiler. A **retargetable compiler** is a compiler whose code generator can emit code for a variety of target processors. An **optimizing compiler** is a compiler whose code generator fully exploits the architecture of the target processor, in addition to the standard optimization techniques such as constant propagation. Modern RISC processors, DSP processors, and VLIW processors depend heavily on optimizing compilers to take advantage of their specific architectures.

The **high level synthesizer** translates the design model into a netlist of register transfer level (RTL) components, as defined in Section 2.3 as a FSMD architecture. The tasks involved in high level synthesis include allocation, scheduling and binding. Allocation selects the number and type of the RTL components from the library. Scheduling assigns time steps to the operations in the behavioral description. Binding maps variables in the description to storage elements, operators to functional units, and data transfers to interconnect units. All these tasks try to optimize appropriate quality metrics subject to design constraints.

We define an interface as a special type of ASIC which links the PE that it is associated (via its native bus) with other components of the system (via the system bus). Such a interface implements the behavior of a communication task, which is generated by a communication synthesis tool to implement the shared variable accesses. Note that a transducer, which translates a transaction on one bus into one or a series of transactions on another bus, is just a special case of the above interface definition. An example of such a transducer translates a read cycle on a processor bus into a read cycle on the system bus. The communication tasks between different PEs are implemented jointly by the driver routines and interrupt service routines implemented in software and the interface circuitry implemented in hardware. While the partitioning of the communication task into software and hardware, and model generation for the two parts is the job of communication synthesis, the task of generating an RTL design from the interface model is the job of **interface synthesis**. Thus interface synthesis is a special case of high level synthesis. The characteristics that distinguish an interface circuitry from a normal ASIC is that its ports have to conform to some predefined protocols. These protocols are often specified in the form of timing diagrams in vendors' data sheets. This poses new challenges to the interface synthesizer for two reasons:

(a) the protocols impose a set of timing constraints on the minimum and maximum skews between events that the interface produces and other possibly external events, which the interface has to satisfy;

(b) the protocols provide a set of timing delays on the minimum and maximum skews between external events and other events, of which the interface may take advantage.

5 Conclusion and Future Work

Codesign represents the methodology for specification and design of systems that include hardware and software components. A codesign methodology consists of design tasks for refining the design and the models representing the refinements.

In this chapter we presented essential issues in codesign. System codesign starts by specifying the system in one of the specification languages based on some conceptual model. Conceptual models were defined in Section 1, implementation architectures in Section 2, while the features needed in executable specifications were given in Section 3. After a specification is obtained the designer must select an architecture, allocate components, and perform partitioning, scheduling and communication synthesis to generate the architectural behavioral description. After each of the above tasks the designer may validate her decisions by generating appropriate simulation models and validating the quality metrics as explained in Section 4.

Presently, very little research has been done in the codesign field. The current CAD tools are mostly simulator backplanes. Future work must include definition of specification languages, automatic refinement of different system descriptions and models, and development of tools for architectural exploration, algorithms for partitioning, scheduling, and synthesis, and backend tools for custom software and hardware synthesis, including IP creation and reuse.

Acknowledgements

We would like to acknowledge the support provided by UCI grant #TC20881 from Toshiba Inc. and grant #95-D5-146 and #96-D5-146 from Semiconductor Research Corporation.

We would also like to acknowledge Prentice-Hall Inc., Upper Saddle River, NJ 07458, for the permission to reprint figures from [GVNG94] (annotated by †), figures from [Gaj97] (annotated by ‡), and the partial use of text appearing in Chapter 2 and Chapter 3 in [GVNG94] and Chapter 6 and Chapter 8 in [Gaj97].

We would also like to thank Jie Gong, Sanjiv Narayan and Frank Vahid for valuable insights and early discussions about models and languages. Furthermore, we want to acknowledge Jon Kleinsmith, En-shou Chang, Tatsuya Umezaki for contributions in language requirements and model development.

References

[Ag90] G. Agha. "The Structure and Semantics of Actor Languages". *Lecture Notes in Computer Science, Foundation* *of Object-Oriented Languages.* Springer-Verlag, 1990.

[AG96] K. Arnold, J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[BCJ+97] F. Balarin, M. Chiodo, A. Jurecska, H. Hsieh, A. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, B. Tabbara *Hardware-Software Co-Design of Embedded Systems: A Polis Approach.* Kluwer Academic Publishers, 1997.

[COB95] P. Chou, R. Ortega, G. Boriello. "Interface Co-synthesis Techniques for Embedded Systems". In *Proceedings of the International Conference on Computer-Aided Design*, 1995.

[CGH+93] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, A. Sangiovanni-Vincentelli. "A formal specification model for hardware/software codesign". Technical Report UCB/ERL M93/48, U.C. Berkeley, June 1993.

[DH89] D. Drusinsky, D. Harel. "Using Statecharts for hardware description and synthesis". In *IEEE Transactions on Computer Aided Design*, 1989.

[EHB93] R. Ernst, J. Henkel, T. Benner. "Hardware-software cosynthesis for microcontrollers". In *IEEE Design and Test*, Vol. 12, 1993.

[FLLO95] R. French, M. Lam, J. Levitt, K. Olukotun. "A General Method for Compiling Event-Driven Simulation". In *Proceedings of 32th Design Automation Conference*, 6, 1995.

[FH92] C. W. Fraser, D. R. Hanson, T. A. Proebsting. "Engineering a Simple, Efficient Code Generator Generator". In *ACM Letters on Programming Languages and Systems*, 1, 3 (Sept. 1992).

[Gaj97] D. D. Gajski. *Principles of Digital Design*, Prentice Hall, 1997.

[GCM92] R. K. Gupta, C. N. Coelho Jr., G. De Micheli. "Synthesis and simulation of digital systems containing interacting hardware and software components". In *Proceedings of the 29th ACM, IEEE Design Automation Conference*, 1992.

[GDWL91] D. D. Gajski, N. D. Dutt, C. H. Wu, Y. L. Lin. *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, Boston, Massachusetts, 1991.

[GVN94] D. D. Gajski, F. Vahid, S. Narayan. "A system-design methodology: Executable-specification refinement". In *Proceedings of the European Conference on Design Automation*, 1994.

[GVNG94] D. Gajski, F. Vahid, S. Narayan, J. Gong. *Specification and Design of Embedded Systems.* New Jersey, Prentice Hall, 1994.

[Har87] D. Harel. "Statecharts: A visual formalism for complex systems". *Science of Computer Programming 8*, 1987.

[HHE94] D. Henkel, J. Herrmann, R. Ernst. "An approach to the adaption of estimated cost parameters in the cosyma system". *Third International Workshop on Hardware/Software Codesign*, Grenoble, 1994.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, New Jersey, 1985.

[HP96] J. L. Hennessy, D. A. Patterson. *Computer Architecture – A Quantitative Approach*, 2nd edition, Morgan-Kaufmann, 1996.

[KL95] A. Kalavade, E. A. Lee. "The extended partitioning problem: Hardware/software mapping and implementation-bin selection". In *Proceedings of the 6th International Workshop on Rapid Systems Prototyping*, 1995.

[Lie97] C. Liem. *Retargetable Compilers for Embedded Core Processors: Methods and Experiences in Industrial Applications.* Kluwer Academic Publishers, 1997.

[LM87] E. A. Lee, D. G. Messerschmidt. "Static Scheduling of Synchronous Data Flow Graphs for Digital Signal Processors". In *IEEE Transactions on Computer-Aided Design*, 87, pp.24-35.

[LMD94] B. Landwehr, P. Marwedel, R. Dömer. "OSCAR: Optimum Simultaneous Scheduling, Allocation and Resource Binding Based on Integer Programming". In *Proceedings of the European Design Automation Conference*, 1994.

[LS96] E. A. Lee, A. Sangiovanni-Vincentelli. "Comparing Models of Computation". In *Proceedings of the International Conference on Computer Design*, San Jose, CA, Nov. 10-14, 1996.

[MG95] P. Marwedel, G. Goosens. *Code Generation for Embedded Processors.* Kluwer Academic Publishers, 1995.

[NM97] R. Niemann, P. Marwedel. "An Algorithm for Hardware/Software Partitioning Using Mixed Integer Linear Programming". In *Design Automation for Embedded Systems, 2*, Kluwer Academic Publishers, 1997.

[Pet81] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[PK93] Z. Peng, K. Kuchcinski. "An Algorithm for partitioning of application specific systems". In *Proceedings of the European Conference on Design Automation*, 1993.

[Rei92] W. Reisig. *A Primer in Petri Net Design.* Springer-Verlag, New York, 1992.

[Stau94] J. Staunstrup. *A Formal Approach to Hardware Design.* Kluwer Academic Publishers, 1994.

[Str87] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, 1987.

[TM91] D. E. Thomas, P. R. Moorby. *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[YW97] T. Y. Yen, W. Wolf. *Hardware-software Co-synthesis of Distributed Embedded Systems.* Kluwer Academic Publishers, 1997.

Index