

# Composition of State Machines

<b>5.1</b>	<b>Concurrent Composition</b>	<b>111</b>
	<i>Sidebar: About Synchrony</i>	112
5.1.1	Side-by-Side Synchronous Composition	113
5.1.2	Side-by-Side Asynchronous Composition	116
	<i>Sidebar: Scheduling Semantics for Asynchronous Composition</i>	118
5.1.3	Shared Variables	119
5.1.4	Cascade Composition	122
5.1.5	General Composition	125
<b>5.2</b>	<b>Hierarchical State Machines</b>	<b>126</b>
<b>5.3</b>	<b>Summary</b>	<b>130</b>
	<b>Exercises</b>	<b>132</b>

State machines provide a convenient way to model behaviors of systems. One disadvantage that they have is that for most interesting systems, the number of states is very large, often even infinite. Automated tools can handle large state spaces, but humans have more difficulty with any direct representation of a large state space.

A time-honored principle in engineering is that complicated systems should be described as compositions of simpler systems. This chapter gives a number of ways to do this with

---

state machines. The reader should be aware, however, that there are many subtly different ways to compose state machines. Compositions that look similar on the surface may mean different things to different people. The rules of notation of a model are called its **syntax**, and the meaning of the notation is called its **semantics**.

**Example 5.1:** In the standard syntax of arithmetic, a plus sign  $+$  has a number or expression before it, and a number or expression after it. Hence,  $1 + 2$ , a sequence of three symbols, is a valid arithmetic expression, but  $1+$  is not. The semantics of the expression  $1 + 2$  is the addition of two numbers. This expression means “the number three, obtained by adding 1 and 2.” The expression  $2 + 1$  is syntactically different, but semantically identical (because addition is commutative).

The models in this book predominantly use a visual syntax, where the elements are boxes, circles, arrows, etc., rather than characters in a character set, and where the positioning of the elements is not constrained to be a sequence. Such syntaxes are less standardized than, for example, the syntax of arithmetic. We will see that the same syntax can have many different semantics, which can cause no end of confusion.

**Example 5.2:** A now popular notation for concurrent composition of state machines called Statecharts was introduced by Harel (1987). Although they are all based on the same original paper, many variants of Statecharts have evolved (von der Beeck, 1994). These variants often assign different semantics to the same syntax.

In this chapter, we assume an actor model for extended state machines using the syntax summarized in Figure 5.1. The semantics of a single such state machine is described in Chapter 3. This chapter will discuss the semantics that can be assigned to compositions of multiple such machines.

The first composition technique we consider is concurrent composition. Two or more state machines react either simultaneously or independently. If the reactions are simultaneous, we call it **synchronous composition**. If they are independent, then we call it

**asynchronous composition.** But even within these classes of composition, many subtle variations in the semantics are possible. These variations mostly revolve around whether and how the state machines communicate and share variables.

The second composition technique we will consider is hierarchy. Hierarchical state machines can also enable complicated systems to be described as compositions of simpler systems. Again, we will see that subtle differences in semantics are possible.

## 5.1 Concurrent Composition

To study concurrent composition of state machines, we will proceed through a sequence of patterns of composition. These patterns can be combined to build arbitrarily complicated systems. We begin with the simplest case, side-by-side composition, where the state machines being composed do not communicate. We then consider allowing communication through shared variables, showing that this creates significant subtleties that can complicate modeling. We then consider communication through ports, first looking at serial composition, then expanding to arbitrary interconnections. We consider both synchronous and asynchronous composition for each type of composition.

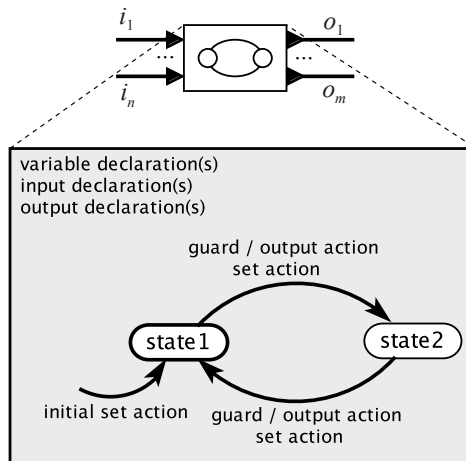


Figure 5.1: Summary of notation for state machines used in this chapter.

### About Synchrony

The term **synchronous** means (1) occurring or existing at the same time or (2) moving or operating at the same rate. In engineering and computer science, the term has a number of meanings that are mostly consistent with these definitions, but oddly inconsistent with one another. In referring to concurrent software constructed using **threads** or **processes**, synchronous communication refers to a **rendezvous** style of communication, where the sender of a message must wait for the receiver to be ready to receive, and the receiver must wait for the sender. Conceptually, the two threads see the communication occurring at the same time, consistent with definition (1). In Java, the keyword `synchronized` defines blocks of code that are not permitted to execute simultaneously. Oddly, two code blocks that are synchronized *cannot* “occur” (execute) at the same time, which is inconsistent with both definitions.

In the world of software, there is yet a third meaning of the word synchronous, and it is this third meaning that we use in this chapter. This third meaning underlies the **synchronous languages** (see box on page 148). Two key ideas govern these languages. First, the outputs of components in a program are (conceptually) simultaneous with their inputs (this is called the **synchrony hypothesis**). Second, components in a program execute (conceptually) **simultaneously and instantaneously**. Real executions do not literally occur simultaneously nor instantaneously, and outputs are not really simultaneous with the inputs, but a correct execution must behave as if they were. This use of the word synchronous is consistent with *both* definitions above; executions of components occur at the same time and operate at the same rate.

In circuit design, the word synchronous refers to a style of design where a clock that is distributed throughout a circuit drives latches that record their inputs on edges of the clock. The time between clock edges needs to be sufficient for circuits between latches to settle. Conceptually, this model is similar to the model in synchronous languages. Assuming that the circuits between latches have zero delay is equivalent to the synchrony hypothesis, and global clock distribution gives simultaneous and instantaneous execution.

In power systems engineering, synchronous means that electrical waveforms have the same frequency and phase. In signal processing, synchronous means that signals have the same sample rate, or that their sample rates are fixed multiples of one another. The term **synchronous dataflow**, described in Section 6.3.2, is based on this latter meaning of the word synchronous. This usage is consistent with definition (2).



### 5.1.1 Side-by-Side Synchronous Composition

The first pattern of composition that we consider is **side-by-side composition**, illustrated for two actors in Figure 5.2. In this pattern, we assume that the inputs and outputs of the two actors are disjoint, i.e., that the state machines do not communicate. In the figure, actor  $A$  has input  $i_1$  and output  $o_1$ , and actor  $B$  has input  $i_2$  and output  $o_2$ . The composition of the two actors is itself an actor  $C$  with inputs  $i_1$  and  $i_2$  and outputs  $o_1$  and  $o_2$ .<sup>1</sup>

In the simplest scenario, if the two actors are **extended state machines** with variables, then those variables are also disjoint. We will later consider what happens when the two state machines share variables. Under **synchronous composition**, a reaction of  $C$  is a simultaneous reaction of  $A$  and  $B$ .

**Example 5.3:** Consider FSMs  $A$  and  $B$  in Figure 5.3.  $A$  has a single pure output  $a$ , and  $B$  has a single pure output  $b$ . The side-by-side composition  $C$  has two pure outputs,  $a$  and  $b$ . If the composition is synchronous, then on the first reaction,  $a$  will be *absent* and  $b$  will be *present*. On the second reaction, it will be the reverse. On subsequent reactions,  $a$  and  $b$  will continue to alternate being present.

<sup>1</sup>The composition actor  $C$  may rename these input and output **ports**, but here we assume it uses the same names as the component actors.

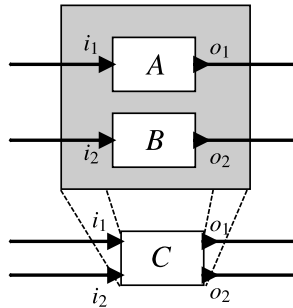


Figure 5.2: Side-by-side composition of two actors.

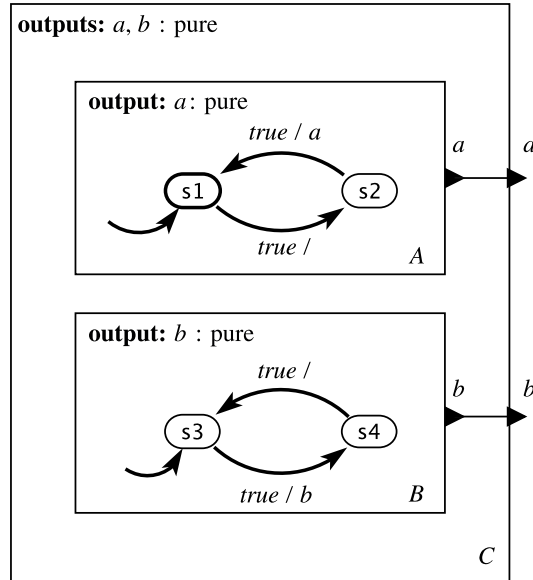


Figure 5.3: Example of side-by-side composition of two actors.

Synchronous side-by-side composition is simple for several reasons. First, recall from Section 3.3.2 that the environment determines when a state machine reacts. In synchronous side-by-side composition, the environment need not be aware that  $C$  is a composition of two state machines. Such compositions are **modular** in the sense that the composition itself becomes a component that can be further composed as if it were itself an atomic component.

Moreover, if the two state machines  $A$  and  $B$  are **deterministic**, then the synchronous side-by-side composition is also deterministic. We say that a property is **compositional** if a property held by the components is also a property of the composition. For synchronous side-by-side composition, determinism is a compositional property.

In addition, a synchronous side-by-side composition of finite state machines is itself an FSM. A rigorous way to give the semantics of the composition is to define a single state machine for the composition. Suppose that as in Section 3.3.3, state machines  $A$  and  $B$

are given by the five tuples,

$$A = (States_A, Inputs_A, Outputs_A, update_A, initialState_A)$$

$$B = (States_B, Inputs_B, Outputs_B, update_B, initialState_B) .$$

Then the synchronous side-by-side composition  $C$  is given by

$$States_C = States_A \times States_B \quad (5.1)$$

$$Inputs_C = Inputs_A \times Inputs_B \quad (5.2)$$

$$Outputs_C = Outputs_A \times Outputs_B \quad (5.3)$$

$$initialState_C = (initialState_A, initialState_B) \quad (5.4)$$

and the update function is defined by

$$update_C((s_A, s_B), (i_A, i_B)) = ((s'_A, s'_B), (o_A, o_B)),$$

where

$$(s'_A, o_A) = update_A(s_A, i_A),$$

and

$$(s'_B, o_B) = update_B(s_B, i_B),$$

for all  $s_A \in States_A$ ,  $s_B \in States_B$ ,  $i_A \in Inputs_A$ , and  $i_B \in Inputs_B$ .

Recall that  $Inputs_A$  and  $Inputs_B$  are sets of **valuations**. Each valuation in the set is an assignment of values to ports. What we mean by

$$Inputs_C = Inputs_A \times Inputs_B$$

is that a valuation of the inputs of  $C$  must include *both* valuations for the inputs of  $A$  and the inputs of  $B$ .

As usual, the single FSM  $C$  can be given pictorially rather than symbolically, as illustrated in the next example.

**Example 5.4:** The synchronous side-by-side composition  $C$  in Figure 5.3 is given as a single FSM in Figure 5.4. Notice that this machine behaves exactly as described in Example 5.3. The outputs  $a$  and  $b$  alternate being present. Notice further that  $(s1, s4)$  and  $(s2, s3)$  are not **reachable states**.

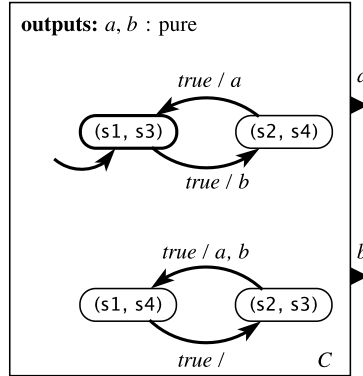


Figure 5.4: Single state machine giving the semantics of synchronous side-by-side composition of the state machines in Figure 5.3.

### 5.1.2 Side-by-Side Asynchronous Composition

In an **asynchronous composition** of state machines, the component machines react independently. This statement is rather vague, and in fact, it has several different interpretations. Each interpretation gives a **semantics** to the composition. The key to each semantics is how to define a **reaction** of the composition  $C$  in Figure 5.2. Two possibilities are:

- **Semantics 1.** A reaction of  $C$  is a reaction of one of  $A$  or  $B$ , where the choice is **nondeterministic**.
- **Semantics 2.** A reaction of  $C$  is a reaction of  $A$ ,  $B$ , or both  $A$  and  $B$ , where the choice is **nondeterministic**. A variant of this possibility might allow *neither* to react.

Semantics 1 is referred to as an **interleaving semantics**, meaning that  $A$  or  $B$  never react simultaneously. Their reactions are interleaved in some order.

A significant subtlety is that under these semantics machines  $A$  and  $B$  may completely miss input events. That is, an input to  $C$  destined for machine  $A$  may be present in a reaction where the nondeterministic choice results in  $B$  reacting rather than  $A$ . If this is not desirable, then some control over scheduling (see sidebar on page 118) or **synchronous composition** becomes a better choice.

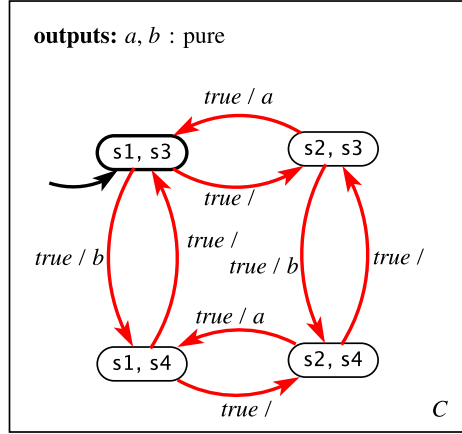


Figure 5.5: State machine giving the semantics of asynchronous side-by-side composition of the state machines in Figure 5.3.

**Example 5.5:** For the example in Figure 5.3, semantics 1 results in the composition state machine shown in Figure 5.5. This machine is nondeterministic. From state  $(s1, s3)$ , when  $C$  reacts, it can move to  $(s2, s3)$  and emit no output, or it can move to  $(s1, s4)$  and emit  $b$ . Note that if we had chosen semantics 2, then it would also be able to move to  $(s2, s4)$ .

For asynchronous composition under semantics 1, the symbolic definition of  $C$  has the same definitions of  $States_C$ ,  $Inputs_C$ ,  $Outputs_C$ , and  $initialState_C$  as for synchronous composition, given in (5.1) through (5.4). But the update function differs, becoming

$$update_C((s_A, s_B), (i_A, i_B)) = ((s'_A, s'_B), (o'_A, o'_B)),$$

where either

$$(s'_A, o'_A) = update_A(s_A, i_A) \text{ and } s'_B = s_B \text{ and } o'_B = absent$$

or

$$(s'_B, o'_B) = update_B(s_B, i_B) \text{ and } s'_A = s_A \text{ and } o'_A = absent$$

for all  $s_A \in States_A$ ,  $s_B \in States_B$ ,  $i_A \in Inputs_A$ , and  $i_B \in Inputs_B$ . What we mean by  $o'_B = \text{absent}$  is that all outputs of  $B$  are absent. Semantics 2 can be similarly defined (see Exercise 2).

### Scheduling Semantics for Asynchronous Composition

In the case of semantics 1 and 2 given in Section 5.1.2, the choice of which component machine reacts is nondeterministic. The model does not express any particular constraints. It is often more useful to introduce some scheduling policies, where the environment is able to influence or control the nondeterministic choice. This leads to two additional possible semantics for asynchronous composition:

- **Semantics 3.** A reaction of  $C$  is a reaction of one of  $A$  or  $B$ , where the environment chooses which of  $A$  or  $B$  reacts.
- **Semantics 4.** A reaction of  $C$  is a reaction of  $A$ ,  $B$ , or both  $A$  and  $B$ , where the choice is made by the environment.

Like semantics 1, semantics 3 is an [interleaving semantics](#).

In one sense, semantics 1 and 2 are more [compositional](#) than semantics 3 and 4. To implement semantics 3 and 4, a composition has to provide some mechanism for the environment to choose which component machine should react (for scheduling the component machines). This means that the hierarchy suggested in Figure 5.2 does not quite work. Actor  $C$  has to expose more of its internal structure than just the ports and the ability to react.

In another sense, semantics 1 and 2 are less compositional than semantics 3 and 4 because determinism is not preserved by composition. A composition of deterministic state machines is not a deterministic state machine.

Notice further that semantics 1 is an [abstraction](#) of semantics 3 in the sense that every behavior under semantics 3 is also a behavior under semantics 1. This notion of abstraction is studied in detail in Chapter 14.

The subtle differences between these choices make asynchronous composition rather treacherous. Considerable care is required to ensure that it is clear which semantics is used.

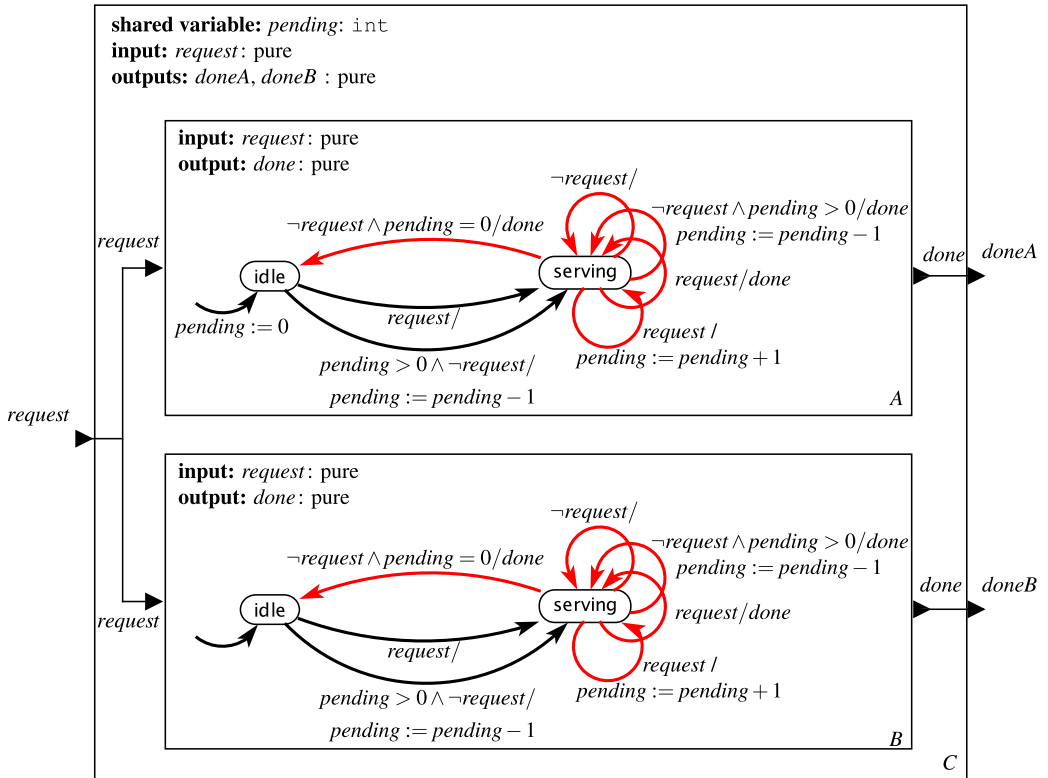


Figure 5.6: Model of two servers with a shared task queue, assuming asynchronous composition under semantics 1.

### 5.1.3 Shared Variables

An **extended state machine** has local variables that can be read and written as part of taking a transition. Sometimes it is useful when composing state machines to allow these variables to be shared among a group of machines. In particular, such shared variables can be useful for modeling **interrupts**, studied in Chapter 10, and **threads**, studied in Chapter 11. However, considerable care is required to ensure that the semantics of the model conforms with that of the program containing interrupts or threads. Many complications arise, including the **memory consistency** model and the notion of **atomic operations**.

**Example 5.6:** Consider two servers that can receive requests from a network. Each request requires an unknown amount of time to service, so the servers share a queue of requests. If one server is busy, the other server can respond to a request, even if the request arrives at the network interface of the first server.

This scenario fits a pattern similar to that in Figure 5.2, where  $A$  and  $B$  are the servers. We can model the servers as state machines as shown in Figure 5.6. In this model, a shared variable *pending* counts the number of pending job requests. When a request arrives at the composite machine  $C$ , one of the two servers is nondeterministically chosen to react, assuming asynchronous composition under semantics 1. If that server is idle, then it proceeds to serve the request. If the server is serving another request, then one of two things can happen: it can coincidentally finish serving the request it is currently serving, issuing the output *done*, and proceed to serve the new one, or it can increment the count of pending requests and continue to serve the current request. The choice between these is nondeterministic, to model the fact that the time it takes to service a request is unknown.

If  $C$  reacts when there is no request, then again either server  $A$  or  $B$  will be selected nondeterministically to react. If the server that reacts is idle and there are one or more pending requests, then the server transitions to serving and decrements the variable *pending*. If the server that reacts is not idle, then one of three things can happen. It may continue serving the current request, in which case it simply transitions on the [self transition](#) back to serving. Or it may finish serving the request, in which case it will transition to idle if there are no pending requests, or transition back to serving and decrement *pending* if there are pending requests.

The model in the previous example exhibits many subtleties of concurrent systems. First, because of the [interleaving semantics](#), accesses to the shared variable are [atomic operations](#), something that is quite challenging to guarantee in practice, as discussed in Chapters 10 and 11. Second, the choice of semantics 1 is reasonable in this case because the input goes to both of the component machines, so regardless of which component machine reacts, no input event will be missed. However, this semantics would not work if the two machines had independent inputs, because then requests could be missed. Semantics 2 can help prevent that, but what strategy should be used by the environment to determine



which machine reacts? What if the two independent inputs both have requests present at the same reaction of  $C$ ? If we choose semantics 4 in the sidebar on page 118 to allow both machines to react simultaneously, then what is the meaning when both machines update the shared variable? The updates are no longer atomic, as they are with an interleaving semantics.

Note further that choosing asynchronous composition under semantics 1 allows behaviors that do not make good use of idle machines. In particular, suppose that machine  $A$  is serving, machine  $B$  is idle, and a *request* arrives. If the nondeterministic choice results in machine  $A$  reacting, then it will simply increment *pending*. Not until the nondeterministic choice results in  $B$  reacting will the idle machine be put to use. In fact, semantics 1 allows behaviors that never use one of the machines.

Shared variables may be used in [synchronous compositions](#) as well, but sophisticated subtleties again emerge. In particular, what should happen if in the same reaction one machine reads a shared variable to evaluate a guard and another machine writes to the shared variable? Do we require the write before the read? What if the transition doing the write to the shared variable also reads the same variable in its guard expression? One possibility is to choose a **synchronous interleaving semantics**, where the component machines react in arbitrary order, chosen nondeterministically. This strategy has the disadvantage that a composition of two deterministic machines may be nondeterministic. An alternative version of the synchronous interleaving semantics has the component machines react in a fixed order determined by the environment or by some additional mechanism such as [priority](#).

The difficulties of shared variables, particularly with asynchronous composition, reflect the inherent complexity of concurrency models with shared variables. Clean solutions require a more sophisticated semantics, to be discussed in Chapter 6. In that chapter, we will explain the [synchronous-reactive](#) model of computation, which gives a synchronous composition semantics that is reasonably compositional.

So far, we have considered composition of machines that do not directly communicate. We next consider what happens when the outputs of one machine are the inputs of another.

### 5.1.4 Cascade Composition

Consider two state machines  $A$  and  $B$  that are composed as shown in Figure 5.7. The output of machine  $A$  feeds the input of  $B$ . This style of composition is called **cascade composition** or **serial composition**.

In the figure, output port  $o_1$  from  $A$  feeds events to input port  $i_2$  of  $B$ . Assume the data type of  $o_1$  is  $V_1$  (meaning that  $o_1$  can take values from  $V_1$  or be *absent*), and the data type of  $i_2$  is  $V_2$ . Then a requirement for this composition to be valid is that

$$V_1 \subseteq V_2 .$$

This asserts that any output produced by  $A$  on port  $o_1$  is an acceptable input to  $B$  on port  $i_2$ . The composition **type checks**.

For cascade composition, if we wish the composition to be asynchronous, then we need to introduce some machinery for buffering the data that is sent from  $A$  to  $B$ . We defer discussion of such asynchronous composition to Chapter 6, where **dataflow** and **process network** models of computation will provide such asynchronous composition. In this chapter, we will only consider synchronous composition for cascade systems.

In synchronous composition of the cascade structure of Figure 5.7, a reaction of  $C$  consists of a reaction of both  $A$  and  $B$ , where  $A$  reacts first, produces its output (if any), and then  $B$  reacts. Logically, we view this as occurring in zero time, so the two reactions are in a sense **simultaneous and instantaneous**. But they are causally related in that the outputs of  $A$  can affect the behavior of  $B$ .

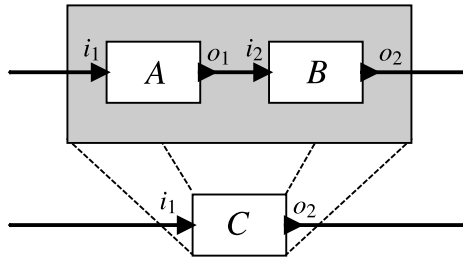


Figure 5.7: Cascade composition of two actors.

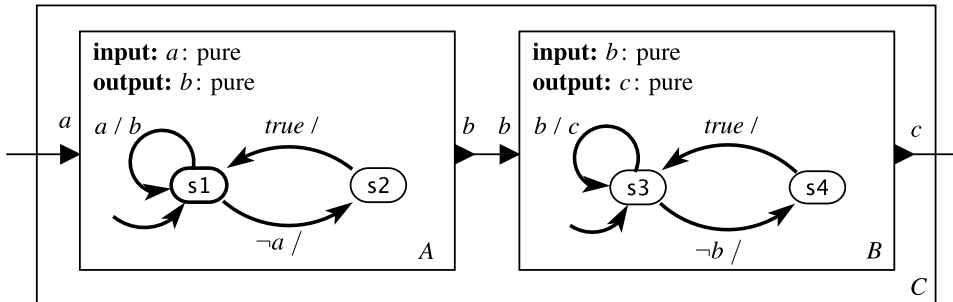


Figure 5.8: Example of a cascade composition of two FSMs.

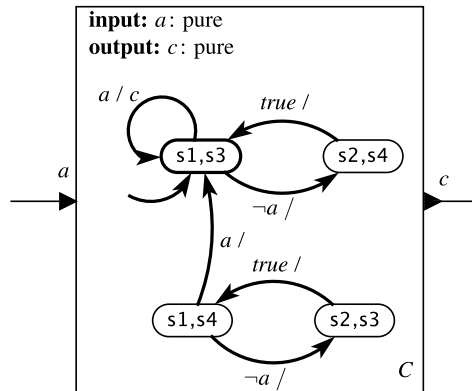


Figure 5.9: Semantics of the cascade composition of Figure 5.8, assuming synchronous composition.

**Example 5.7:** Consider the cascade composition of the two FSMs in Figure 5.8. Assuming synchronous semantics, the meaning of a reaction of  $C$  is given in Figure 5.9. That figure makes it clear that the reactions of the two machines are simultaneous and instantaneous. When moving from the initial state (s1, s3) to (s2, s4) (which occurs when the input  $a$  is absent), the composition machine  $C$

does not pass through  $(s2, s3)$ ! In fact,  $(s2, s3)$  is not a **reachable state**! In this way, a *single* reaction of  $C$  encompasses a reaction of both  $A$  and  $B$ .

To construct the composition machine as in Figure 5.9, first form the state space as the cross product of the state spaces of the component machines, and then determine which transitions are taken under what conditions. It is important to remember that the transitions are simultaneous, even when one logically causes the other.

**Example 5.8:** Recall the traffic light model of Figure 3.10. Suppose that we wish to compose this with a model of a pedestrian crossing light, like that shown in Figure 5.10. The output  $sigR$  of the traffic light can provide the input  $sigR$  of the pedestrian light. Under synchronous cascade composition, the meaning of the composite is given in Figure 5.11. Note that unsafe states, such as (green, green), which is the state when both cars and pedestrians have a green light, are not **reachable states**, and hence are not shown.

In its simplest form, cascade composition implies an ordering of the reactions of the

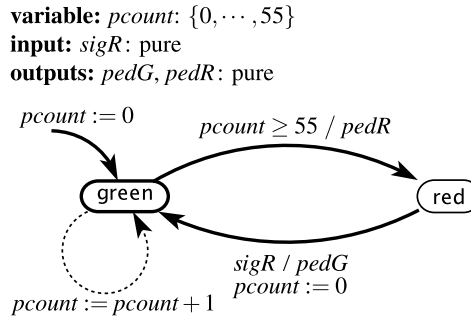


Figure 5.10: A model of a pedestrian crossing light, to be composed in a synchronous cascade composition with the traffic light model of Figure 3.10.

**variables:**  $count: \{0, \dots, 60\}, pcount: \{0, \dots, 55\}$   
**input:**  $pedestrian$ : pure  
**outputs:**  $sigR, sigG, sigY, pedG, pedR$ : pure

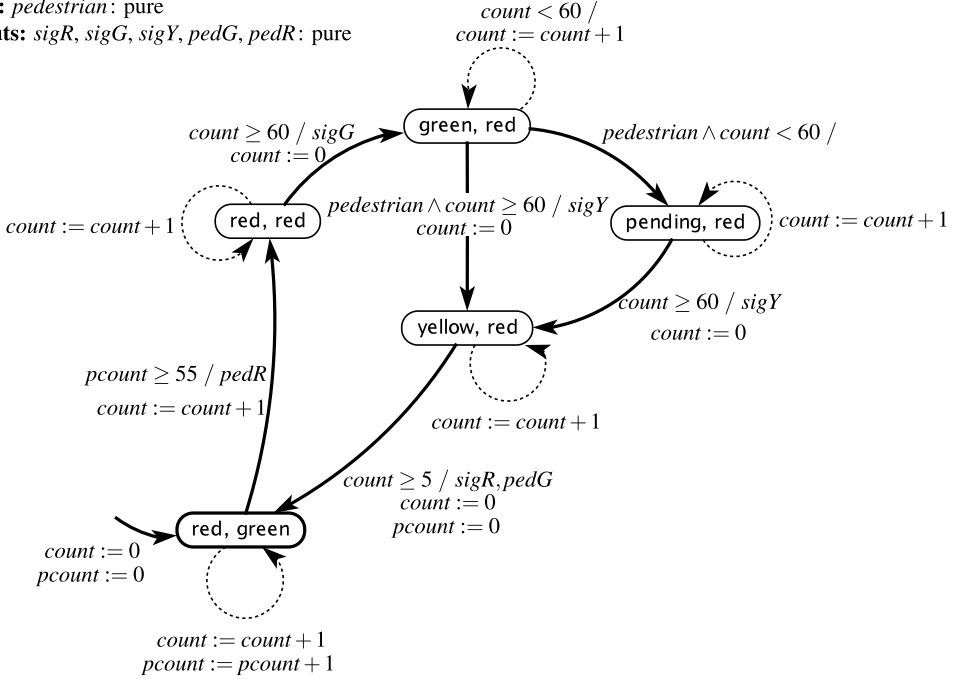


Figure 5.11: Semantics of a synchronous cascade composition of the traffic light model of Figure 3.10 with the pedestrian light model of Figure 5.10.

components. Since this ordering is well defined, we do not have as much difficulty with shared variables as we did with side-by-side composition. However, we will see that in more general compositions, the ordering is not so simple.

### 5.1.5 General Composition

Side-by-side and cascade composition provide the basic building blocks for building more complex compositions of machines. Consider for example the composition in Figure 5.12.  $A_1$  and  $A_3$  are a side-by-side composition that together define a machine  $B$ .  $B$  and  $A_2$  are a cascade composition, with  $B$  feeding events to  $A_2$ . However,  $B$  and  $A_2$  are also a cascade composition in the opposite order, with  $A_2$  feeding events to  $B$ . Cycles like

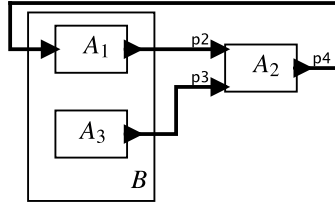


Figure 5.12: Arbitrary interconnections of state machines are combinations of side-by-side and cascade compositions, possibly creating cycles, as in this example.

this are called [feedback](#), and they introduce a conundrum; which machine should react first,  $B$  or  $A_2$ ? This conundrum will be resolved in the next chapter when we explain the [synchronous-reactive](#) model of computation.

## 5.2 Hierarchical State Machines

In this section, we consider **hierarchical FSMs**, which date back to Statecharts ([Harel, 1987](#)). There are many variants of Statecharts, often with subtle semantic differences between them ([von der Beeck, 1994](#)). Here, we will focus on some of the simpler aspects only, and we will pick a particular semantic variant.

The key idea in hierarchical state machines is [state refinement](#). In Figure 5.13, state  $B$  has a refinement that is another FSM with two states,  $C$  and  $D$ . What it means for the machine to be in state  $B$  is that it is in one of states  $C$  or  $D$ .

The meaning of the hierarchy in Figure 5.13 can be understood by comparing it to the equivalent flattened FSM in Figure 5.14. The machine starts in state  $A$ . When guard  $g_2$  evaluates to true, the machine transitions to state  $B$ , which means a transition to state  $C$ , the initial state of the refinement. Upon taking this transition to  $C$ , the machine performs action  $a_2$ , which may produce an output event or set a variable (if this is an [extended state machine](#)).

There are then two ways to exit  $C$ . Either guard  $g_1$  evaluates to true, in which case the machine exits  $B$  and returns to  $A$ , or guard  $g_4$  evaluates to true and the machine transitions to  $D$ . A subtle question is what happens if both guards  $g_1$  and  $g_4$  evaluate to true. Different

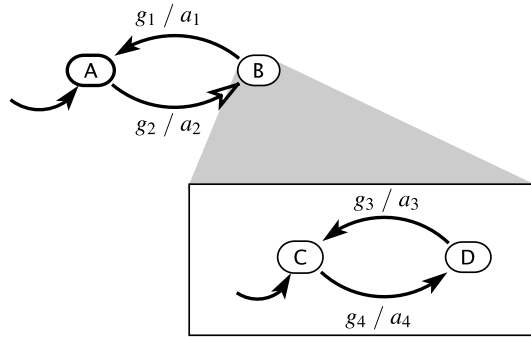


Figure 5.13: In a hierarchical FSM, a state may have a refinement that is another state machine.

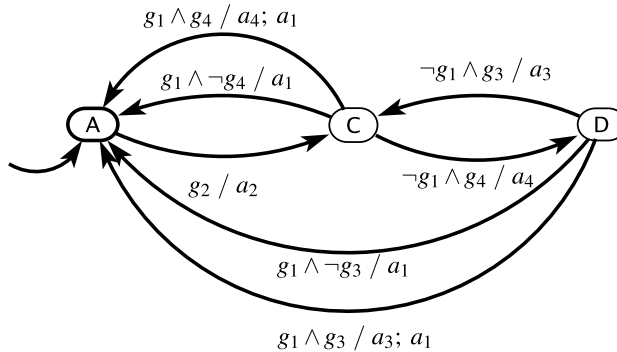


Figure 5.14: Semantics of the hierarchical FSM in Figure 5.13.

variants of Statecharts may make different choices at this point. It seems reasonable that the machine should end up in state A, but which of the actions should be performed,  $a_4$ ,  $a_1$ , or both? Such subtle questions help account for the proliferation of different variants of Statecharts.

We choose a particular semantics that has attractive modularity properties (Lee and Triakis, 2010). In this semantics, a reaction of a hierarchical FSM is defined in a depth-first fashion. The deepest refinement of the current state reacts first, then its container state machine, then its container, etc. In Figure 5.13, this means that if the machine is in state

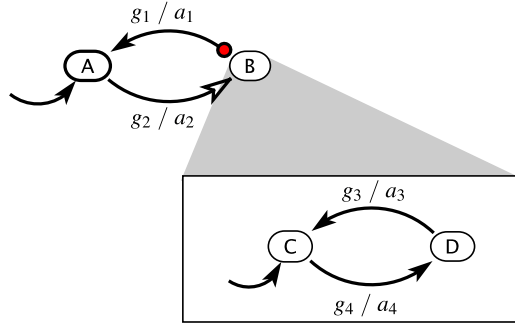


Figure 5.15: Variant of Figure 5.13 that uses a preemptive transition.

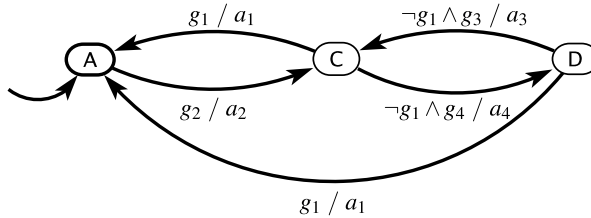


Figure 5.16: Semantics of Figure 5.15 with a preemptive transition.

B (which means that it is in either C or D), then the refinement machine reacts first. If it is C, and guard  $g_4$  is true, the transition is taken to D and action  $a_4$  is performed. But then, as part of the same reaction, the top-level FSM reacts. If guard  $g_1$  is also true, then the machine transitions to state A. It is important that logically these two transitions are *simultaneous and instantaneous*, so the machine does not actually go to state D. Nonetheless, action  $a_4$  is performed, and so is action  $a_1$ . This combination corresponds to the topmost transition of Figure 5.14.

Another subtlety is that if two (non-absent) actions are performed in the same reaction, they may conflict. For example, two actions may write different values to the same output port. Or they may set the same variable to different values. Our choice is that the actions are performed in sequence, as suggested by the semicolon in the action  $a_4; a_1$ . As in an *imperative* language like C, the semicolon denotes a sequence. If the two actions conflict, the later one dominates.



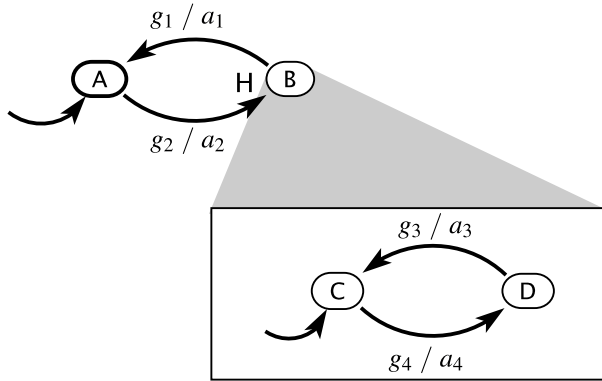


Figure 5.17: Variant of the hierarchical state machine of Figure 5.13 that has a history transition.

Such subtleties can be avoided by using a **preemptive transition**, shown in Figure 5.15, which has the semantics shown in Figure 5.16. The guards of a preemptive transition are evaluated *before* the refinement reacts, and if any guard evaluates to true, the refinement does not react. As a consequence, if the machine is in state B and  $g_1$  is true, then neither action  $a_3$  nor  $a_4$  is performed. A preemptive transition is shown with a (red) circle at the originating end of the transition.

Notice in Figures 5.13 and 5.14 that whenever the machine enters B, it always enters C, never D, even if it was previously in D when leaving B. The transition from A to B is called a **reset transition** because the destination refinement is reset to its initial state, regardless of where it had previously been. A reset transition is indicated in our notation with a hollow arrowhead at the destination end of a transition.

In Figure 5.17, the transition from A to B is a **history transition**, an alternative to a reset transition. In our notation, a solid arrowhead denotes a history transition. It may also be marked with an “H” for emphasis. When a history transition is taken, the destination refinement resumes in whatever state it was last in (or its initial state on the first entry).

The semantics of the history transition is shown in Figure 5.18. The initial state is labeled (A, C) to indicate that the machine is in state A, and if and when it next enters B it will go

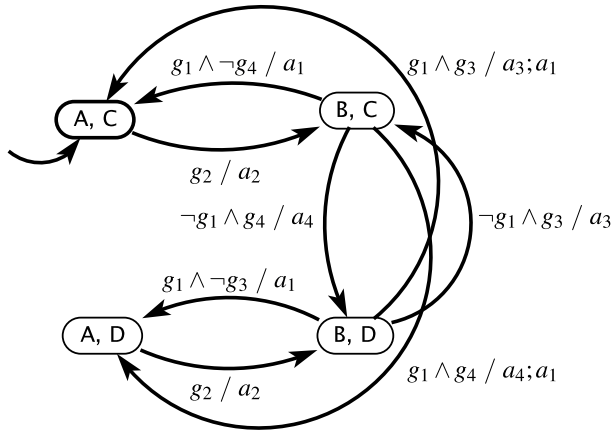


Figure 5.18: Semantics of the hierarchical state machine of Figure 5.17 that has a history transition.

to C. The first time it goes to B, it will be in the state labeled (B, C) to indicate that it is in state B and, more specifically, C. If it then transitions to (B, D), and then back to A, it will end up in the state labeled (A, D), which means it is in state A, but if and when it next enters B it will go to D. That is, it remembers its history, specifically where it was when it left B.

As with concurrent composition, hierarchical state machines admit many possible meanings. The differences can be subtle. Considerable care is required to ensure that models are clear and that their semantics match what is being modeled.

## 5.3 Summary

Any well-engineered system is a composition of simpler components. In this chapter, we have considered two forms of composition of state machines, concurrent composition and hierarchical composition.

For concurrent composition, we introduced both synchronous and asynchronous composition, but did not complete the story. We have deferred dealing with feedback to the

next chapter, because for synchronous composition, significant subtleties arise. For asynchronous composition, communication via ports requires additional mechanisms that are not (yet) part of our model of state machines. Even without communication via ports, significant subtleties arise because there are several possible semantics for asynchronous composition, and each has strengths and weaknesses. One choice of semantics may be suitable for one application and not for another. These subtleties motivate the topic of the next chapter, which provides more structure to concurrent composition and resolves most of these questions (in a variety of ways).

For hierarchical composition, we focus on a style originally introduced by [Harel \(1987\)](#) known as Statecharts. We specifically focus on the ability for states in an FSM to have refinements that are themselves state machines. The reactions of the refinement FSMs are composed with those of the machine that contains the refinements. As usual, there are many possible semantics.

## Exercises

1. Consider the extended state machine model of Figure 3.8, the garage counter. Suppose that the garage has two distinct entrance and exit points. Construct a side-by-side concurrent composition of two counters that share a variable  $c$  that keeps track of the number of cars in the garage. Specify whether you are using synchronous or asynchronous composition, and define exactly the semantics of your composition by giving a single machine modeling the composition. If you choose synchronous semantics, explain what happens if the two machines simultaneously modify the shared variable. If you choose asynchronous composition, explain precisely which variant of asynchronous semantics you have chosen and why. Is your composition machine deterministic?
2. For semantics 2 in Section 5.1.2, give the five tuple for a single machine representing the composition  $C$ ,

$$(\text{States}_C, \text{Inputs}_C, \text{Outputs}_C, \text{update}_C, \text{initialState}_C)$$

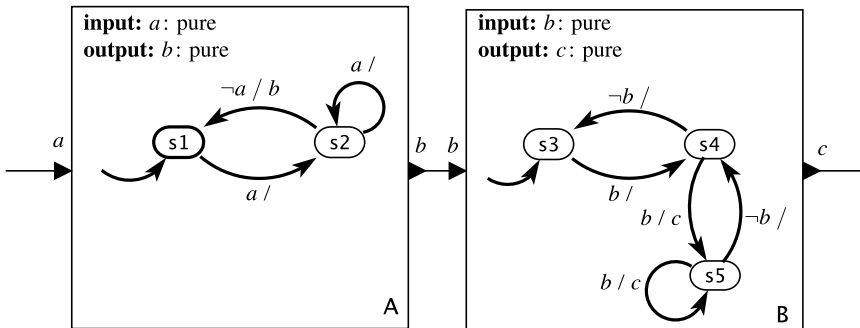
for the side-by-side asynchronous composition of two state machines  $A$  and  $B$ . Your answer should be in terms of the five-tuple definitions for  $A$  and  $B$ ,

$$(\text{States}_A, \text{Inputs}_A, \text{Outputs}_A, \text{update}_A, \text{initialState}_A)$$

and

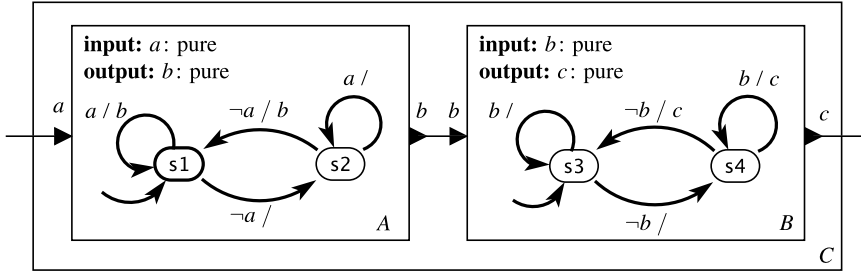
$$(\text{States}_B, \text{Inputs}_B, \text{Outputs}_B, \text{update}_B, \text{initialState}_B)$$

3. Consider the following synchronous composition of two state machines  $A$  and  $B$ :



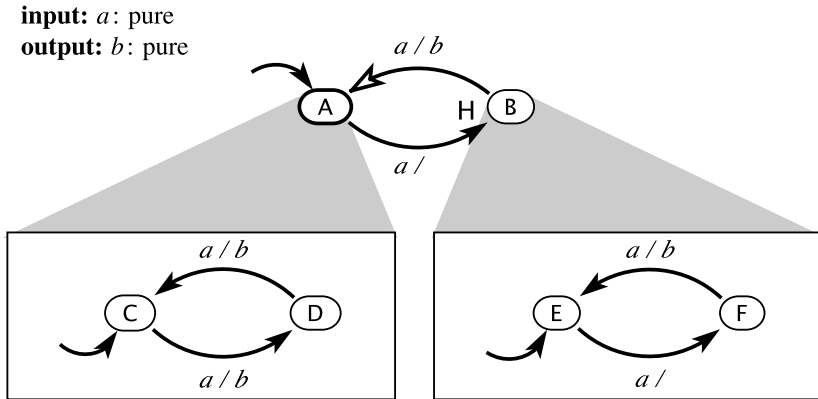
Construct a single state machine  $C$  representing the composition. Which states of the composition are unreachable?

4. Consider the following synchronous composition of two state machines  $A$  and  $B$ :



Construct a single state machine  $C$  representing the composition. Which states of the composition are unreachable?

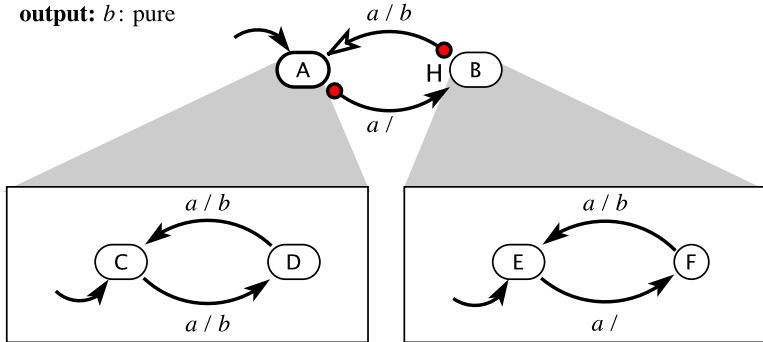
5. Consider the following hierarchical state machine:



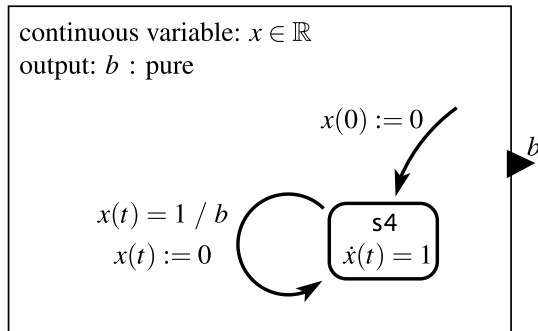
Construct an equivalent flat FSM giving the semantics of the hierarchy. Describe in words the input/output behavior of this machine. Is there a simpler machine that exhibits the same behavior? (Note that equivalence relations between state machines are considered in Chapter 14, but here, you can use intuition and just consider what the state machine does when it reacts.)

6. How many reachable states does the following state machine have?

**input:**  $a$ : pure  
**output:**  $b$ : pure



7. Suppose that the machine of Exercise 8 of Chapter 4 is composed in a synchronous side-by-side composition with the following machine:



Find a tight lower bound on the time between events  $a$  and  $b$ . That is, find a lower bound on the time gap during which there are no events in signals  $a$  or  $b$ . Give an argument that your lower bound is tight.

# Concurrent Models of Computation

<b>6.1</b>	<b>Structure of Models</b>	<b>137</b>
	<i>Sidebar: Actor Networks as a System of Equations</i>	139
	<i>Sidebar: Fixed-Point Semantics</i>	140
<b>6.2</b>	<b>Synchronous-Reactive Models</b>	<b>141</b>
6.2.1	Feedback Models	141
6.2.2	Well-Formed and Ill-Formed Models	143
6.2.3	Constructing a Fixed Point	145
<b>6.3</b>	<b>Dataflow Models of Computation</b>	<b>147</b>
	<i>Sidebar: Synchronous-Reactive Languages</i>	148
6.3.1	Dataflow Principles	149
6.3.2	Synchronous Dataflow	152
6.3.3	Dynamic Dataflow	157
6.3.4	Structured Dataflow	158
6.3.5	Process Networks	159
<b>6.4</b>	<b>Timed Models of Computation</b>	<b>162</b>
6.4.1	Time-Triggered Models	162
6.4.2	Discrete Event Systems	163
6.4.3	Continuous-Time Systems	164
<b>6.5</b>	<b>Summary</b>	<b>167</b>
	<i>Sidebar: Petri Nets</i>	168
	<i>Sidebar: Models of Time</i>	169
	<i>Sidebar: Probing Further: Discrete Event Semantics</i>	170
	<b>Exercises</b>	<b>172</b>

---

In sound engineering practice, systems are built by composing components. In order for the composition to be well understood, we need first for the individual components to be well understood, and then for the meaning of the interaction between components to be well understood. The previous chapter dealt with composition of finite state machines. With such composition, the components are well defined (they are [FSMs](#)), but there are many possible interpretations for the interaction between components. The meaning of a composition is referred to as its [semantics](#).

This chapter focuses on the semantics of **concurrent** composition. The word “concurrent” literally means “running together.” A system is said to be concurrent if different parts of the system (components) conceptually operate at the same time. There is no particular order to their operations. The semantics of such concurrent operation can be quite subtle, however.

The components we consider in this chapter are [actors](#), which react to stimuli at input ports and produce stimuli on output ports. In this chapter, we will be only minimally concerned with how the actors themselves are defined. They may be FSMs, hardware, or programs specified in an [imperative](#) programming language. We will need to impose some constraints on what these actors can do, but we need not constrain how they are specified.

The semantics of a concurrent composition of actors is governed by three sets of rules that we collectively call a **model of computation (MoC)**. The first set of rules specifies what constitutes a component. The second set specifies the concurrency mechanisms. The third specifies the communication mechanisms.

In this chapter, a component will be an actor with ports and a set of **execution actions**. An execution action defines how the actor reacts to inputs to produce outputs and change state. The ports will be interconnected to provide for communication between actors, and the execution actions will be invoked by the environment of the actor to cause the actor to perform its function. For example, for FSMs, one action is provided that causes a [reaction](#). The focus of this chapter is on introducing a few of the possible concurrency and communication mechanisms that can govern the interactions between such actors.

We begin by laying out the common structure of models that applies to all MoCs studied in this chapter. We then proceed to describe a suite of MoCs.



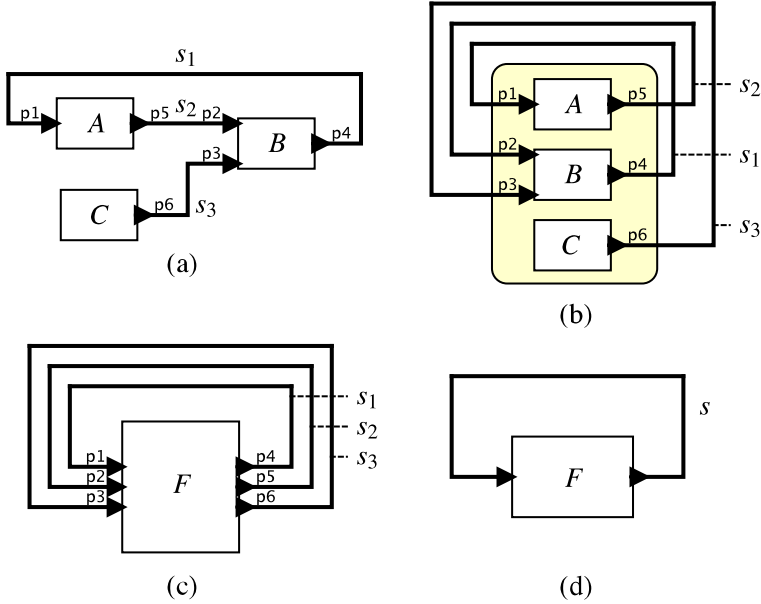


Figure 6.1: Any interconnection of actors can be modeled as a single (side-by-side composite) actor with feedback.

## 6.1 Structure of Models

In this chapter, we assume that models consist of fixed interconnections of actors like that shown in Figure 6.1(a). The interconnections between actors specify communication paths. The communication itself takes the form of a **signal**, which consists of one or more **communication events**. For the **discrete signals** of Section 3.1, for example, a signal  $s$  has the form of a function

$$s: \mathbb{R} \rightarrow V_s \cup \{absent\},$$

where  $V_s$  is a set of values called the **type** of the signal  $s$ . A communication event in this case is a non-absent value of  $s$ .

**Example 6.1:** Consider a **pure signal**  $s$  that is a discrete signal given by

$$s(t) = \begin{cases} \text{present} & \text{if } t \text{ is a multiple of } P \\ \text{absent} & \text{otherwise} \end{cases}$$

for all  $t \in \mathbb{R}$  and some  $P \in \mathbb{R}$ . Such a signal is called a **clock signal** with period  $P$ . Communication events occur every  $P$  time units.

In Chapter 2, a continuous-time signal has the form of a function

$$s: \mathbb{R} \rightarrow V_s,$$

in which case every one of the (uncountably) infinite set of values  $s(t)$ , for all  $t \in \mathbb{R}$ , is a communication event. In this chapter, we will also encounter signals of the form

$$s: \mathbb{N} \rightarrow V_s,$$

where there is no time line. The signal is simply a sequence of values.

A communication event has a type, and we require that a connection between actors **type check**. That is, if an output port  $y$  with type  $V_y$  is connected to an input port  $x$  with type  $V_x$ , then

$$V_y \subseteq V_x.$$

As suggested in Figure 6.1(b-d), any actor network can be reduced to a rather simple form. If we rearrange the actors as shown in Figure 6.1(b), then the actors form a **side-by-side composition** indicated by the box with rounded corners. This box is itself an actor  $F$  as shown in Figure 6.1(c) whose input is a three-tuple  $(s_1, s_2, s_3)$  of signals and whose output is *the same* three-tuple of signals. If we let  $s = (s_1, s_2, s_3)$ , then the actor can be depicted as in Figure 6.1(d), which hides all the complexity of the model.

Notice that Figure 6.1(d) is a **feedback** system. By following the procedure that we used to build it, every interconnection of actors can be structured as a similar feedback system (see Exercise 1).

### Actor Networks as a System of Equations

In a model, if the actors are **determinate**, then each actor is a function that maps input signals to output signals. For example, in Figure 6.1(a), actor  $A$  may be a function relating signals  $s_1$  and  $s_2$  as follows,

$$s_2 = A(s_1).$$

Similarly, actor  $B$  relates three signals by

$$s_1 = B(s_2, s_3).$$

Actor  $C$  is a bit more subtle, since it has no input ports. How can it be a function? What is the **domain** of the function? If the actor is determinate, then its output signal  $s_3$  is a constant signal. The function  $C$  needs to be a constant function, one that yields the same output for every input. A simple way to ensure this is to define  $C$  so that its domain is a **singleton set** (a set with only one element). Let  $\{\emptyset\}$  be the singleton set, so  $C$  can only be applied to  $\emptyset$ . The function  $C$  is then given by

$$C(\emptyset) = s_3.$$

Hence, Figure 6.1(a) gives a system of equations

$$\begin{aligned} s_1 &= B(s_2, s_3) \\ s_2 &= A(s_1) \\ s_3 &= C(\emptyset). \end{aligned}$$

The semantics of such a model, therefore, is a solution to such a system of equations. This can be represented compactly using the function  $F$  in Figure 6.1(d), which is

$$F(s_1, s_2, s_3) = (B(s_2, s_3), A(s_1), C(\emptyset)).$$

All actors in Figure 6.1(a) have output ports; if we had an actor with no output port, then we could similarly define it as a function whose **codomain** is  $\{\emptyset\}$ . The output of such function is  $\emptyset$  for all inputs.

### Fixed-Point Semantics

In a model, if the actors are **determinate**, then each actor is a function that maps input signals to output signals. The semantics of such a model is a system of equations (see sidebar on page 139) and the reduced form of Figure 6.1(d) becomes

$$s = F(s), \quad (6.1)$$

where  $s = (s_1, s_2, s_3)$ . Of course, this equation only *looks* simple. Its complexity lies in the definition of the function  $F$  and the structure of the domain and range of  $F$ .

Given any function  $F: X \rightarrow X$  for any set  $X$ , if there is an  $x \in X$  such that  $F(x) = x$ , then  $x$  is called a **fixed point**. Equation (6.1) therefore asserts that the semantics of a determinate actor network is a fixed point. Whether a fixed point exists, whether the fixed point is unique, and how to find the fixed point, all become interesting questions that are central to the model of computation.

In the **SR** model of computation, the execution of all actors is **simultaneous and instantaneous** and occurs at ticks of the global clock. If the actor is determinate, then each such execution implements a function called a **firing function**. For example, in the  $n$ -th tick of the global clock, actor  $A$  in Figure 6.1 implements a function of the form

$$a_n: V_1 \cup \{absent\} \rightarrow V_2 \cup \{absent\}$$

where  $V_i$  is the type of signal  $s_i$ . Hence, if  $s_i(n)$  is the value of  $s_i$  at the  $n$ -th tick, then

$$s_2(n) = a_n(s_1(n)).$$

Given such a firing function  $f_n$  for each actor  $F$  we can, just as in Figure 6.1(d) define the execution at a single tick by a fixed point,

$$s(n) = f_n(s(n)),$$

where  $s(n) = (s_1(n), s_2(n), s_3(n))$  and  $f_n$  is a function where

$$f_n(s_1(n), s_2(n), s_3(n)) = (b_n(s_2(n), s_3(n)), a_n(s_1(n)), c_n(\emptyset)).$$

Thus, for **SR**, the semantics at each tick of the global clock is a fixed point of the function  $f_n$ , just as its execution over all ticks is a fixed point of the function  $F$ .

## 6.2 Synchronous-Reactive Models

In Chapter 5 we studied synchronous composition of state machines, but we avoided the nuances of feedback compositions. For a model described as the feedback system of Figure 6.1(d), the conundrum discussed in Section 5.1.5 takes a particularly simple form. If  $F$  in Figure 6.1(d) is realized by a state machine, then in order for it to react, we need to know its inputs at the time of the reaction. But its inputs are the same as its outputs, so in order for  $F$  to react, we need to know its outputs. But we cannot know its outputs until after it reacts.

As shown in Section 6.1 above and Exercise 1, all actor networks can be viewed as feedback systems, so we really do have to resolve the conundrum. We do that now by giving a model of computation known as the **synchronous-reactive (SR)** MoC.

An SR model is a **discrete system** where signals are absent at all times except (possibly) at **ticks** of a **global clock**. Conceptually, execution of a model is a sequence of global reactions that occur at discrete times, and at each such reaction, the reaction of all actors is **simultaneous and instantaneous**.

### 6.2.1 Feedback Models

We focus first on feedback models of the form of Figure 6.1(d), where  $F$  in the figure is realized as a state machine. At the  $n$ -th tick of the global clock, we have to find the value of the signal  $s$  so that it is both a valid input and a valid output of the state machine, given its current state. Let  $s(n)$  denote the value of the signal  $s$  at the  $n$ -th reaction. The goal is to determine, at each tick of the global clock, the value of  $s(n)$ .

**Example 6.2:** Consider first a simpler example shown in Figure 6.2. (This is simpler than Figure 6.1(d) because the signal  $s$  is a single pure signal rather than an aggregation of three signals.) If  $A$  is in state  $s1$  when that reaction occurs, then the only possible value for  $s(n)$  is  $s(n) = \text{absent}$  because a reaction must take one of the transitions out of  $s1$ , and both of these transitions emit *absent*. Moreover, once we know that  $s(n) = \text{absent}$ , we know that the input port  $x$  has value *absent*, so we can determine that  $A$  will transition to state  $s2$ .

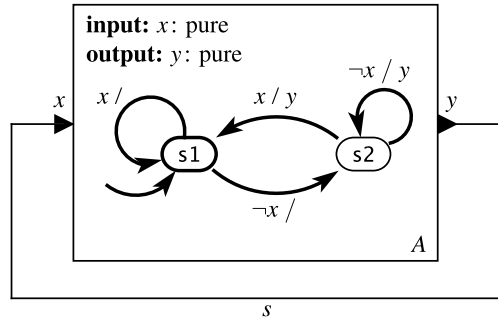


Figure 6.2: A simple well-formed feedback model.

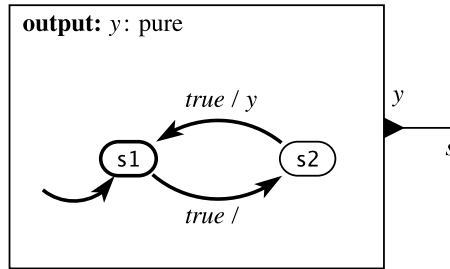


Figure 6.3: The semantics of the model in Figure 6.2.

If  $A$  is in state  $s2$  when the reaction occurs, then the only possible value for  $s(n)$  is  $s(n) = present$ , and the machine will transition to state  $s1$ . Therefore,  $s$  alternates between *absent* and *present*. The semantics of machine  $A$  in the feedback model is therefore given by Figure 6.3.

In the previous example, it is important to note that the input  $x$  and output  $y$  have the *same value* in every reaction. This is what is meant by the feedback connection. Any connection from an output port to an input port means that the value at the input port is the same as the value at the output port at all times.

Given a **deterministic** state machine in a feedback model like that of Figure 6.2, in each state  $i$  we can define a function  $a_i$  that maps input values to output values,

$$a_i: \{present, absent\} \rightarrow \{present, absent\},$$

where the function depends on the state the machine is in. This function is defined by the **update function**.

**Example 6.3:** For the example in Figure 6.2, if the machine is in state  $s_1$ , then  $a_{s_1}(x) = absent$  for all  $x \in \{present, absent\}$ .

The function  $a_i$  is called the **firing function** for state  $i$  (see box on page 140). Given a firing function, to find the value  $s(n)$  at the  $n$ -th reaction, we simply need to find a value  $s(n)$  such that

$$s(n) = a_i(s(n)).$$

Such a value  $s(n)$  is called a **fixed point** of the function  $a_i$ . It is easy to see how to generalize this so that the signal  $s$  can have any type. Signal  $s$  can even be an aggregation of signals, as in Figure 6.1(d) (see box on page 140).

## 6.2.2 Well-Formed and Ill-Formed Models

There are two potential problems that may occur when seeking a fixed point. First, there may be no fixed point. Second, there may be more than one fixed point. If either case occurs in a **reachable state**, we call the system **ill formed**. Otherwise, it is **well formed**.

**Example 6.4:** Consider machine  $B$  shown in Figure 6.4. In state  $s_1$ , we get the unique fixed point  $s(n) = absent$ . In state  $s_2$ , however, there is no fixed point. If we attempt to choose  $s(n) = present$ , then the machine will transition to  $s_1$  and its output will be *absent*. But the output has to be the same as the input, and the input is *present*, so we get a contradiction. A similar contradiction occurs if we attempt to choose  $s(n) = absent$ .

Since state  $s_2$  is reachable, this feedback model is ill formed.

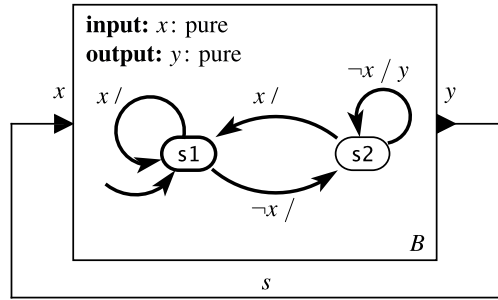


Figure 6.4: An ill-formed feedback model that has no fixed point in state s2.

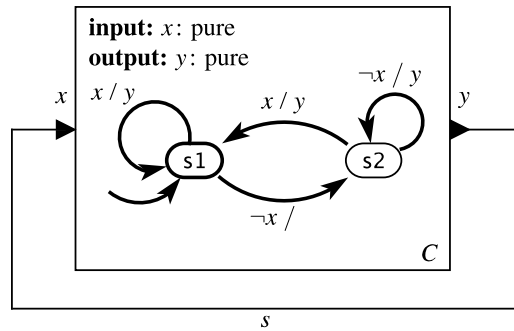


Figure 6.5: An ill-formed feedback model that has more than one fixed point in state s1.

**Example 6.5:** Consider machine *C* shown in Figure 6.5. In state s1, both  $s(n) = \text{absent}$  and  $s(n) = \text{present}$  are fixed points. Either choice is valid. Since state s1 is reachable, this feedback model is ill formed.

If in a reachable state there is more than one fixed point, we declare the machine to be ill formed. An alternative semantics would not reject such a model, but rather would declare it to be nondeterministic. This would be a valid semantics, but it would have the



disadvantage that a composition of deterministic state machines is not assured of being deterministic. In fact,  $C$  in Figure 6.5 is deterministic, and under this alternative semantics, the feedback composition in the figure would not be deterministic. Determinism would not be a **compositional** property. Hence, we prefer to reject such models.

### 6.2.3 Constructing a Fixed Point

If the type  $V_s$  of the signal  $s$  or the signals it is an aggregate of is finite, then one way to find a fixed point is by **exhaustive search**, which means to try all values. If exactly one fixed point is found, then the model is well formed. However, exhaustive search is expensive (and impossible if the types are not finite). We can develop instead a systematic procedure that for most, but not all, well-formed models will find a fixed point. The procedure is as follows. For each reachable state  $i$ ,

1. Start with  $s(n)$  *unknown*.
2. Determine as much as you can about  $f_i(s(n))$ , where  $f_i$  is the firing function in state  $i$ . Note that in this step, you should use only the firing function, which is given by the state machine; you should not use knowledge of how the state machine is connected on the outside.
3. Repeat step 2 until all values in  $s(n)$  become known (whether they are present and what their values are), or until no more progress can be made.
4. If unknown values remain, then reject the model.

This procedure may reject models that have a unique fixed point, as illustrated by the following example.

**Example 6.6:** Consider machine  $D$  shown in Figure 6.6. In state  $s1$ , if the input is unknown, we cannot immediately tell what the output will be. We have to try all the possible values for the input to determine that in fact  $s(n) = \text{absent}$  for all  $n$ .

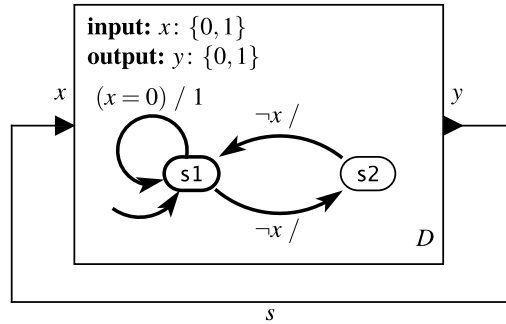


Figure 6.6: A well-formed feedback model that is not constructive.

A state machine for which the procedure works in all reachable states is said to be **constructive** (Berry, 1999). The example in Figure 6.6 is not constructive. For non-constructive machines, we are forced to do exhaustive search or to invent some more elaborate solution technique. Since exhaustive search is often too expensive for practical use, many SR languages and modeling tools (see box on page 148) reject non-constructive models.

Step 2 of the above procedure is key. How exactly can we determine the outputs if the inputs are not all known? This requires what is called a **must-may analysis** of the model. Examining the machine, we can determine what *must* be true of the outputs and what *may* be true of the outputs.

**Example 6.7:** The model in Figure 6.2 is constructive. In state s1, we can immediately determine that the machine *may not* produce an output. Therefore, we can immediately conclude that the output is *absent*, even though the input is unknown. Of course, once we have determined that the output is absent, we now know that the input is absent, and hence the procedure concludes.

In state s2, we can immediately determine that the machine *must* produce an output, so we can immediately conclude that the output is *present*.

The above procedure can be generalized to an arbitrary model structure. Consider for example Figure 6.1(a). There is no real need to convert it to the form of Figure 6.1(d). Instead, we can just begin by labeling all signals unknown, and then in arbitrary order, examine each actor to determine whatever can be determined about the outputs, given its initial state. We repeat this until no further progress can be made, at which point either all signals become known, or we reject the model as either ill-formed or non-constructive. Once we know all signals, then all actors can make state transitions, and we repeat the procedure in the new state for the next reaction.

The constructive procedure above can be adapted to support nondeterministic machines (see Exercise 4). But now, things become even more subtle, and there are variants to the semantics. One way to handle nondeterminism is that when executing the constructive procedure, when encountering a nondeterministic choice, make an arbitrary choice. If the result leads to a failure of the procedure to find a fixed point, then we could either reject the model (not all choices lead to a well-formed or constructive model) or reject the choice and try again.

In the SR model of computation, actors react simultaneously and instantaneously, at least conceptually. Achieving this with realistic computation requires tight coordination of the computation. We consider next a family of models of computation that require less coordination.

## 6.3 Dataflow Models of Computation

In this section, we consider MoCs that are much more asynchronous than SR. Reactions may occur simultaneously, or they may not. Whether they do or do not is not an essential part of the semantics. The decision as to when a reaction occurs can be much more decentralized, and can in fact reside with each individual actor. When reactions are dependent on one another, the dependence is due to the flow of data, rather than to the synchrony of events. If a reaction of actor  $A$  requires data produced by a reaction of actor  $B$ , then the reaction of  $A$  must occur after the reaction of  $B$ . A MoC where such data dependencies are the key constraints on reactions is called a **dataflow** model of computation. There are several variants of dataflow MoCs, a few of which we consider here.

### Synchronous-Reactive Languages

The synchronous-reactive MoC has a history dating at least back to the mid 1980s when a suite of programming languages were developed. The term “reactive” comes from a distinction in computational systems between **transformational systems**, which accept input data, perform computation, and produce output data, and **reactive systems**, which engage in an ongoing dialog with their environment (Harel and Pnueli, 1985). Manna and Pnueli (1992) state

“The role of a reactive program ... is not to produce a final result but to maintain some ongoing interaction with its environment.”

The distinctions between transformational and reactive systems led to the development of a number of innovative programming languages. The **synchronous languages** (Benveniste and Berry, 1991) take a particular approach to the design of reactive systems, in which pieces of the program react simultaneously and instantaneously at each tick of a global clock. First among these languages are Lustre (Halbwachs et al., 1991), Esterel (Berry and Gonthier, 1992), and Signal (Le Guernic et al., 1991). Statecharts (Harel, 1987) and its implementation in Statemate (Harel et al., 1990) also have a strongly synchronous flavor.

SCADE (Berry, 2003) (Safety Critical Application Development Environment), a commercial product of Esterel Technologies, builds on Lustre, borrows concepts from Esterel, and provides a graphical syntax, where state machines are drawn and actor models are composed in a similar manner to the figures in this text. One of the main attractions of synchronous languages is their strong formal properties that yield quite effectively to formal analysis and verification techniques. For this reason, SCADE models are used in the design of safety-critical flight control software systems for commercial aircraft made by Airbus.

The principles of synchronous languages can also be used in the style of a **coordination language** rather than a programming language, as done in Ptolemy II (Edwards and Lee, 2003) and ForSyDe (Sander and Jantsch, 2004). This allows for “primitives” in a system to be complex components rather than built-in language primitives. This approach allows heterogeneous combinations of MoCs, since the complex components may themselves be given as compositions of further subcomponents under some other MoC.

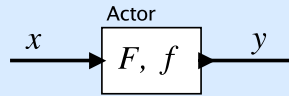
### 6.3.1 Dataflow Principles

In dataflow models, the signals providing communication between the actors are *sequences* of message, where each message is called a **token**. That is, a signal  $s$  is a **partial function** of the form

$$s: \mathbb{N} \rightharpoonup V_s,$$

where  $V_s$  is the **type** of the signal, and where the signal is defined on an **initial segment**  $\{0, 1, \dots, n\} \subset \mathbb{N}$ , or (for infinite executions) on the entire set  $\mathbb{N}$ . Each element  $s(n)$  of this sequence is a token. A (determinate) actor will be described as a function that maps input sequences to output sequences. We will actually use two functions, an **actor function**, which maps *entire* input sequences to *entire* output sequences, and a **firing function**, which maps a finite portion of the input sequences to output sequences, as illustrated in the following example.

**Example 6.8:** Consider an actor that has one input and one output port as shown below



Suppose that the input type is  $V_x = \mathbb{R}$ . Suppose that this is a Scale actor parameterized by a parameter  $a \in \mathbb{R}$ , similar to the one in Example 2.3, which multiplies inputs by  $a$ . Then

$$F(x_1, x_2, x_3, \dots) = (ax_1, ax_2, ax_3, \dots).$$

Suppose that when the actor fires, it performs one multiplication in the firing. Then the firing function  $f$  operates only on the first element of the input sequence, so

$$f(x_1, x_2, x_3, \dots) = f(x_1) = (ax_1).$$

The output is a sequence of length one.

As illustrated in the previous example, the actor function  $F$  combines the effects of multiple invocations of the firing function  $f$ . Moreover, the firing function can be invoked with only partial information about the input sequence to the actor. In the above example, the firing function can be invoked if one or more tokens are available on the input. The rule requiring one token is called a **firing rule** for the Scale actor. A firing rule specifies the number of tokens required on each input port in order to fire the actor.

The Scale actor in the above example is particularly simple because the firing rule and the firing function never vary. Not all actors are so simple.

**Example 6.9:** Consider now a different actor Delay with parameter  $d \in \mathbb{R}$ . The actor function is

$$D(x_1, x_2, x_3, \dots) = (d, x_1, x_2, x_3, \dots).$$

This actor prepends a sequence with a token with value  $d$ . This actor has two firing functions,  $d_1$  and  $d_2$ , and two firing rules. The first firing rule requires no input tokens at all and produces an output sequence of length one, so

$$d_1(s) = (d),$$

where  $s$  is a sequence of any length, including length zero (the empty sequence). This firing rule is initially the one used, and it is used exactly once. The second firing rule requires one input token and is used for all subsequent firings. It triggers the firing function

$$d_2(x_1, \dots) = (x_1).$$

The actor consumes one input token and produces on its output the same token. The actor can be modeled by a state machine, as shown in Figure 6.7. In that figure, the firing rules are implicit in the guards. The tokens required to fire are exactly those required to evaluate the guards. The firing function  $d_1$  is associated with state  $s_1$ , and  $d_2$  with  $s_2$ .

When dataflow actors are composed, with an output of one going to an input of another, the communication mechanism is quite different from that of the previous MoCs consid-

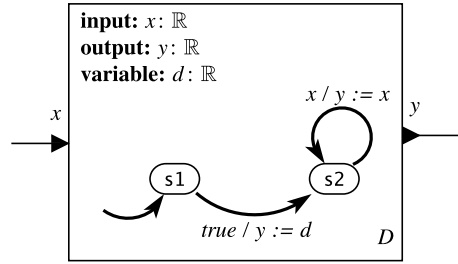
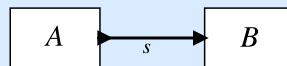


Figure 6.7: An FSM model for the Delay actor in Example 6.9.

ered in this chapter. Since the firing of the actors is asynchronous, a token sent from one actor to another must be buffered; it needs to be saved until the destination actor is ready to consume it. When the destination actor fires, it **consumes** one or more input tokens. After being consumed, a token may be discarded (meaning that the memory in which it is buffered can be reused for other purposes).

Dataflow models pose a few interesting problems. One question is how to ensure that the memory devoted to buffering of tokens is bounded. A dataflow model may be able to execute forever (or for a very long time); this is called an **unbounded execution**. For an unbounded execution, we may have to take measures to ensure that buffering of unconsumed tokens does not overflow the available memory.

**Example 6.10:** Consider the following [cascade composition](#) of dataflow actors:



Since  $A$  has no input ports, its firing rule is simple. It can fire at any time. Suppose that on each firing,  $A$  produces one token. What is to keep  $A$  from firing at a faster rate than  $B$ ? Such faster firing could result in an unbounded build up of unconsumed tokens on the buffer between  $A$  and  $B$ . This will eventually exhaust available memory.

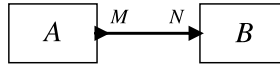


Figure 6.8: SDF actor  $A$  produces  $M$  tokens when it fires, and actor  $B$  consumes  $N$  tokens when it fires.

In general, for dataflow models that are capable of unbounded execution, we will need scheduling policies that deliver **bounded buffers**.

A second problem that may arise is **deadlock**. Deadlock occurs when there are cycles, as in Figure 6.1, and a directed loop has insufficient tokens to satisfy any of the firing rules of the actors in the loop. The Delay actor of Example 6.9 can help prevent deadlock because it is able to produce an initial output token without having any input tokens available. Dataflow models with **feedback** will generally require Delay actors (or something similar) in every cycle.

For general dataflow models, it can be difficult to tell whether the model will deadlock, and whether there exists an unbounded execution with bounded buffers. In fact, these two questions are **undecidable**, meaning that there is no algorithm that can answer the question in bounded time for all dataflow models (Buck, 1993). Fortunately, there are useful constraints that we can impose on the design of actors that make these questions decidable. We examine those constraints next.

### 6.3.2 Synchronous Dataflow

**Synchronous dataflow (SDF)** is a constrained form of dataflow where for each actor, every firing consumes a fixed number of input tokens on each input port and produces a fixed number of output tokens on each output port (Lee and Messerschmitt, 1987).<sup>1</sup>

---

<sup>1</sup>Despite the term, synchronous dataflow is not synchronous in the sense of SR. There is no global clock in SDF models, and firings of actors are asynchronous. For this reason, some authors use the term **static dataflow** rather than synchronous dataflow. This does not avoid all confusion, however, because Dennis (1974) had previously coined the term “static dataflow” to refer to dataflow graphs where buffers could hold at most one token. Since there is no way to avoid a collision of terminology, we stick with the original “synchronous dataflow” terminology used in the literature. The term SDF arose from a signal processing concept, where two signals with **sample rates** that are related by a rational multiple are deemed to be synchronous.



Consider a single connection between two actors,  $A$  and  $B$ , as shown in Figure 6.8. The notation here means that when  $A$  fires, it produces  $M$  tokens on its output port, and when  $B$  fires, it consumes  $N$  tokens on its input port.  $M$  and  $N$  are positive integers. Suppose that  $A$  fires  $q_A$  times and  $B$  fires  $q_B$  times. All tokens that  $A$  produces are consumed by  $B$  if and only if the following **balance equation** is satisfied,

$$q_A M = q_B N. \quad (6.2)$$

Given values  $q_A$  and  $q_B$  satisfying (6.2), we can find a schedule that delivers unbounded execution with bounded buffers. An example of such a schedule fires  $A$  repeatedly,  $q_A$  times, followed by  $B$  repeatedly,  $q_B$  times. It can repeat this sequence forever without exhausting available memory.

**Example 6.11:** Suppose that in Figure 6.8,  $M = 2$  and  $N = 3$ . Then  $q_A = 3$  and  $q_B = 2$  satisfy (6.2). Hence, the following schedule can be repeated forever,

$A, A, A, B, B.$

An alternative schedule is also available,

$A, A, B, A, B.$

In fact, this latter schedule has an advantage over the former one in that it requires less memory.  $B$  fires as soon as there are enough tokens, rather than waiting for  $A$  to complete its entire cycle.

Another solution to (6.2) is  $q_A = 6$  and  $q_B = 4$ . This solution includes more firings in the schedule than are strictly needed to keep the system in balance.

The equation is also satisfied by  $q_A = 0$  and  $q_B = 0$ , but if the number of firings of actors is zero, then no useful work is done. Clearly, this is not a solution we want. Negative solutions are also not desirable.

Generally we will be interested in finding the least positive integer solution to the balance equations.

In a more complicated SDF model, every connection between actors results in a balance equation. Hence, the model defines a system of equations.

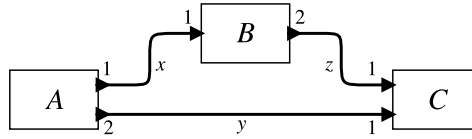


Figure 6.9: A consistent SDF model.

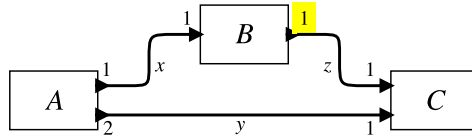


Figure 6.10: An inconsistent SDF model.

**Example 6.12:** Figure 6.9 shows a network with three SDF actors. The connections  $x$ ,  $y$ , and  $z$ , result in the following system of balance equations,

$$\begin{aligned} q_A &= q_B \\ 2q_B &= q_C \\ 2q_A &= q_C. \end{aligned}$$

The least positive integer solution to these equations is  $q_A = q_B = 1$ , and  $q_C = 2$ , so the following schedule can be repeated forever to get an unbounded execution with bounded buffers,

$$A, B, C, C.$$

The balance equations do not always have a non-trivial solution, as illustrated in the following example.

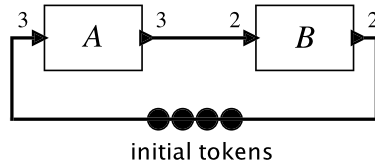


Figure 6.11: An SDF model with initial tokens on a feedback loop.

**Example 6.13:** Figure 6.10 shows a network with three SDF actors where the only solution to the balance equations is the trivial one,  $q_A = q_B = q_C = 0$ . A consequence is that there is no unbounded execution with bounded buffers for this model. It cannot be kept in balance.

An SDF model that has a non-zero solution to the balance equations is said to be **consistent**. If the only solution is zero, then it is **inconsistent**. An inconsistent model has no unbounded execution with bounded buffers.

Lee and Messerschmitt (1987) showed that if the balance equations have a non-zero solution, then they also have a solution where  $q_i$  is a positive integer for all actors  $i$ . Moreover, for connected models (where there is a communication path between any two actors), they gave a procedure for finding the least positive integer solution. Such a procedure forms the foundation for a scheduler for SDF models.

Consistency is sufficient to ensure bounded buffers, but it is not sufficient to ensure that an unbounded execution exists. In particular, when there is feedback, as in Figure 6.1, then **deadlock** may occur. Deadlock bounds an execution.

To allow for feedback, the SDF model treats Delay actors specially. Recall from Example 6.9, that the Delay actor is able to produce output tokens before it receives any input tokens, and then it subsequently behaves like a trivial SDF actor that copies inputs to outputs. But such a trivial actor is not really needed, and the cost of copying inputs to outputs is unnecessary. The Delay actor can be implemented very efficiently as a connection with initial tokens (those tokens that the actor is able to produce before receiving inputs). No actor is actually needed at run time. The scheduler must take the initial tokens into account.

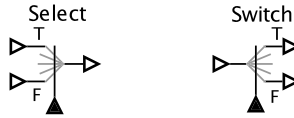


Figure 6.12: Dynamic dataflow actors.

**Example 6.14:** Figure 6.11 shows an SDF model with initial tokens on a feed-back loop. These replace a Delay actor that is able to initially produce four tokens. The balance equations are

$$\begin{aligned} 3q_A &= 2q_B \\ 2q_B &= 3q_A. \end{aligned}$$

The least positive integer solution is  $q_A = 2$ , and  $q_B = 3$ , so the model is consistent. With four initial tokens on the feedback connection, as shown, the following schedule can be repeated forever,

$$A, B, A, B, B.$$

Were there any fewer than four initial tokens, however, the model would deadlock. If there were only three tokens, for example, then  $A$  could fire, followed by  $B$ , but in the resulting state of the buffers, neither could fire again.

In addition to the procedure for solving the balance equations, Lee and Messerschmitt (1987) gave a procedure that will either provide a schedule for an unbounded execution or will prove that no such schedule exists. Hence, both bounded buffers and deadlock are decidable for SDF models.

### 6.3.3 Dynamic Dataflow

Although the ability to guarantee bounded buffers and rule out deadlock is valuable, it comes at a price. SDF is not very expressive. It cannot directly express, for example, conditional firing, where an actor fires only if, for example, a token has a particular value. Such conditional firing is supported by a more general dataflow MoC known as **dynamic dataflow (DDF)**. Unlike SDF actors, DDF actors can have multiple firing rules, and they are not constrained to produce the same number of output tokens on each firing. The Delay actor of Example 6.9 is directly supported by the DDF MoC, without any need for special treatment of initial tokens. So are two basic actors known as Switch and Select, shown in Figure 6.12.

The Select actor on the left has three firing rules. Initially, it requires one token on the bottom input port. The type of that port is Boolean, so the value of the token must be *true* or *false*. If a token with value *true* is received on that input port, then the actor produces no output, but instead activates the next firing rule, which requires one token for the top left input port, labeled *T*. When the actor next fires, it consumes the token on the *T* port and sends it to the output port. If a token with value *false* is received on the bottom input port, then the actor activates a firing rule that requires a token on the bottom left input port labeled *F*. When it consumes that token, it again sends it to the output port. Thus, it fires twice to produce one output.

The Switch actor performs a complementary function. It has only one firing rule, which requires a single token on both input ports. The token on the left input port will be sent to either the *T* or the *F* output port, depending on the Boolean value of the token received on the bottom input port. Hence, Switch and Select accomplish conditional routing of tokens, as illustrated in the following example.

**Example 6.15:** Figure 6.13 uses Switch and Select to accomplish conditional firing. Actor *B* produces a stream of Boolean-valued tokens *x*. This stream is replicated by the fork to provide the control inputs *y* and *z* to the Switch and Select actors. Based on the value of the control tokens on these streams, the tokens produced by actor *A* are sent to either *C* or *D*, and the resulting outputs are collected and sent to *E*. This model is the DDF equivalent of the familiar `if-then-else` programming construct in **imperative** languages.

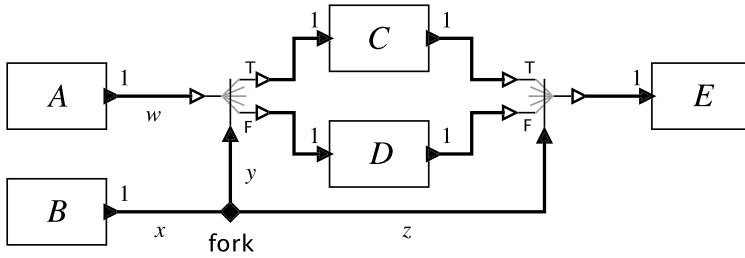


Figure 6.13: A DDF model that accomplishes conditional firing.

Addition of Switch and Select to the actor library means that we can no longer always find a bounded buffer schedule, nor can we provide assurances that the model will not deadlock. Buck (1993) showed that bounded buffers and deadlock are *undecidable* for DDF models. Thus, in exchange for the increased expressiveness and flexibility, we have paid a price. The models are not as readily analyzed.

Switch and Select are dataflow analogs of the **goto** statement in *imperative* languages. They provide low-level control over execution by conditionally routing tokens. Like goto statements, using them can result in models that are very difficult to understand. Dijkstra (1968) indicted the goto statement, discouraging its use, advocating instead the use of **structured programming**. Structured programming replaces goto statements with nested for loops, if-then-else, do-while, and recursion. Fortunately, structured programming is also available for dataflow models, as we discuss next.

### 6.3.4 Structured Dataflow

Figure 6.14 shows an alternative way to accomplish conditional firing that has many advantages over the DDF model in Figure 6.13. The grey box in the figure is an example of a **higher-order actor** called Conditional. A higher-order actor is an actor that has one or more models as parameters. In the example in the figure, Conditional is parameterized by two sub-models, one containing the actor C and the other containing the actor D. When Conditional fires, it consumes one token from each input port and produces one token on its output port, so it is an SDF actor. The action it performs when it fires, however, is dependent on the value of the token that arrives at the lower input port. If that value is true, then actor C fires. Otherwise, actor D fires.

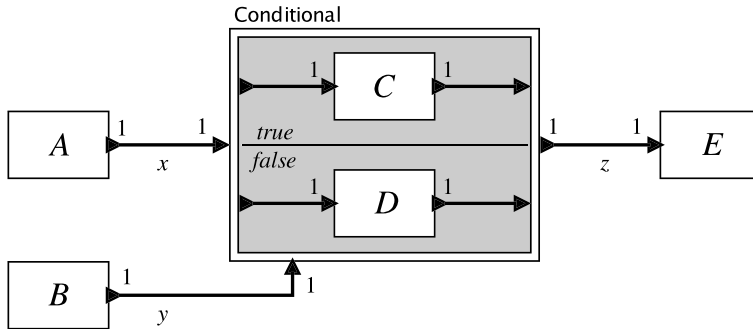


Figure 6.14: Structured dataflow approach to conditional firing.

This style of conditional firing is called **structured dataflow**, because, much like structured programming, control constructs are nested hierarchically. Arbitrary data-dependent token routing is avoided (which is analogous to avoiding arbitrary branches using goto instructions). Moreover, when using such Conditional actors, the overall model is still an SDF model. In the example in Figure 6.14, every actor consumes and produces exactly one token on every port. Hence, the model is analyzable for deadlock and bounded buffers.

This style of structured dataflow was introduced in LabVIEW, a design tool developed by National Instruments (Kodosky et al., 1991). In addition to a conditional similar to that in Figure 6.14, LabVIEW provides structured dataflow constructs for iterations (analogous to for and do-while loops in an imperative language), for case statements (which have an arbitrary number of conditionally executed submodels), and for sequences (which cycle through a finite set of submodels). It is also possible to support recursion using structured dataflow (Lee and Parks, 1995), but without careful constraints, boundedness again becomes undecidable.

### 6.3.5 Process Networks

A model of computation that is closely related to dataflow models is **Kahn process networks** (or simply, **process networks** or **PN**), named after Gilles Kahn, who introduced them (Kahn, 1974). The relationship between dataflow and PN is studied in detail by Lee and Parks (1995) and Lee and Matsikoudis (2009), but the short story is quite sim-

ple. In PN, each actor executes concurrently in its own **process**. That is, instead of being defined by its firing rules and firing functions, a PN actor is defined by a (typically non-terminating) program that reads data tokens from input ports and writes data tokens to output ports. All actors execute simultaneously (conceptually, whether they actually execute simultaneously or are interleaved is irrelevant).

In the original paper, [Kahn \(1974\)](#) gave very elegant mathematical conditions on the actors that would ensure that a network of such actors was determinate in the sense that the sequence of tokens on every connection between actors is unique, and specifically independent of how the processes are scheduled. Thus, Kahn showed that concurrent execution was possible without nondeterminacy.

Three years later, [Kahn and MacQueen \(1977\)](#) gave a simple, easily implemented mechanism for programs that ensures that the mathematical conditions are met to ensure determinacy. A key part of the mechanism is to perform **blocking reads** on input ports whenever a process is to read input data. Specifically, blocking reads mean that if the process chooses to access data through an input port, it issues a read request and blocks until the data becomes available. It cannot test the input port for the availability of data and then perform a conditional branch based on whether data are available, because such a branch would introduce schedule-dependent behavior.

Blocking reads are closely related to firing rules. Firing rules specify the tokens required to continue computing (with a new firing function). Similarly, a blocking read specifies a single token required to continue computing (by continuing execution of the process).

When a process writes to an output port, it performs a **nonblocking write**, meaning that the write succeeds immediately and returns. The process does not block to wait for the receiving process to be ready to receive data. This is exactly how writes to output ports work in dataflow MoCs as well. Thus, the only material difference between dataflow and PN is that with PN, the actor is not broken down into firing functions. It is designed as a continuously executing program.

[Kahn and MacQueen \(1977\)](#) called the processes in a PN network **coroutines** for an interesting reason. A **routine** or **subroutine** is a program fragment that is “called” by another program. The subroutine executes to completion before the calling fragment can continue executing. The interactions between processes in a PN model are more symmetric, in that there is no caller and callee. When a process performs a blocking read, it is in a sense invoking a routine in the upstream process that provides the data. Similarly, when it performs a write, it is in a sense invoking a routine in the downstream process to



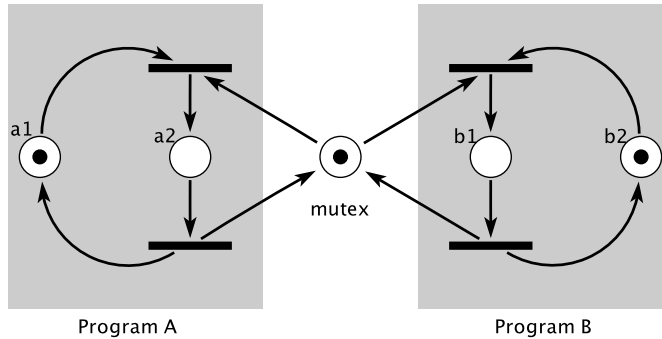


Figure 6.15: A Petri net model of two concurrent programs with a mutual exclusion protocol.

process the data. But the relationship between the producer and consumer of the data is much more symmetric than with subroutines.

Just like dataflow, the PN MoC poses challenging questions about boundedness of buffers and about deadlock. PN is expressive enough that these questions are *undecidable*. An elegant solution to the boundedness question is given by Parks (1995) and elaborated by Geilen and Basten (2003).

An interesting variant of process networks performs **blocking writes** rather than non-blocking writes. That is, when a process writes to an output port, it blocks until the receiving process is ready to receive the data. Such an interaction between processes is called a **rendezvous**. Rendezvous forms the basis for well known process formalisms such as **communicating sequential processes (CSP)** (Hoare, 1978) and the **calculus of communicating systems (CCS)** (Milner, 1980). It also forms the foundation for the **Occam** programming language (Galletly, 1996), which enjoyed some success for a period of time in the 1980s and 1990s for programming parallel computers.

In both the SR and dataflow models of computation considered so far, time plays a minor role. In dataflow, time plays no role. In SR, computation occurs simultaneously and instantaneously at each of a sequence of ticks of a global clock. Although the term “clock” implies that time plays a role, it actually does not. In the SR MoC, all that matters is the sequence. The physical time at which the ticks occur is irrelevant to the MoC. It is just a *sequence* of ticks. Many modeling tasks, however, require a more explicit notion of time. We examine next MoCs that have such a notion.

## 6.4 Timed Models of Computation

For [cyber-physical systems](#), the time at which things occur in software can matter, because the software interacts with physical processes. In this section, we consider a few concurrent MoCs that explicitly refer to time. We describe three timed MoCs, each of which have many variants. Our treatment here is necessarily brief. A complete study of these MoCs would require a much bigger volume.

### 6.4.1 Time-Triggered Models

[Kopetz and Grunsteidl \(1994\)](#) introduced mechanisms for periodically triggering distributed computations according to a distributed clock that measures the passage of time. The result is a system architecture called a **time-triggered architecture (TTA)**. A key contribution was to show how a TTA could tolerate certain kinds of faults, such that failures in part of the system could not disrupt the behaviors in other parts of the system (see also [Kopetz \(1997\)](#) and [Kopetz and Bauer \(2003\)](#)). [Henzinger et al. \(2003\)](#) lifted the key idea of TTA to the programming language level, providing a well-defined semantics for modeling distributed time-triggered systems. Since then, these techniques have come into practical use in the design of safety-critical avionics and automotive systems, becoming a key part of standards such as FlexRay, a networking standard developed by a consortium of automotive companies.

A time-triggered MoC is similar to [SR](#) in that there is a global clock that coordinates the computation. But computations take time instead of being [simultaneous and instantaneous](#). Specifically, time-triggered MoCs associate with a computation a **logical execution time**. The inputs to the computation are provided at ticks of the global clock, but the outputs are not visible to other computations until the *next* tick of the global clock. Between ticks, there is no interaction between the computations, so concurrency difficulties such as [race conditions](#) do not exist. Since the computations are not (logically) instantaneous, there are no difficulties with feedback, and all models are [constructive](#).

The Simulink modeling system, sold by The MathWorks, supports a time-triggered MoC, and in conjunction with another product called Real-Time Workshop, can translate such models in embedded C code. In LabVIEW, from National Instruments, timed loops accomplish a similar capability within a [dataflow](#) MoC.

In the simplest form, a time-triggered model specifies periodic computation with a fixed time interval (the **period**) between ticks of the clock. Giotto (Henzinger et al., 2003) supports **modal models**, where the periods differ in different modes. Some authors have further extended the concept of logical execution time to non-periodic systems (Liu and Lee, 2003; Ghosal et al., 2004).

Time triggered models are conceptually simple, but computations are tied closely to a periodic clock. The model becomes awkward when actions are not periodic. DE systems, considered next, encompass a richer set of timing behaviors.

### 6.4.2 Discrete Event Systems

**Discrete-event systems** (DE systems) have been used for decades as a way to build simulations for an enormous variety of applications, including for example digital networks, military systems, and economic systems. A pioneering formalism for DE models is due to Zeigler (1976), who called the formalism **DEVS**, abbreviating discrete event system specification. DEVS is an extension of **Moore machines** that associates a non-zero lifespan with each state, thus endowing the Moore machines with an explicit notion of the passage of time (vs. a sequence of reactions).

The key idea in a DE MoC is that events are endowed with a **time stamp**, a value in some model of time (see box on page 169). Normally, two distinct time stamps must be comparable. That is, they are either equal, or one is earlier than the other. A DE model is a network of actors where each actor reacts to input events in time-stamp order and produces output events in time-stamp order.

**Example 6.16:** The **clock signal** with period  $P$  of Example 6.1 consists of events with time stamps  $nP$  for all  $n \in \mathbb{Z}$ .

To execute a DE model, we can use an **event queue**, which is a list of events sorted by time stamp. The list begins empty. Each actor in the network is interrogated for any initial events it wishes to place on the event queue. These events may be destined for another actor, or they may be destined for the actor itself, in which case they will cause a reaction of the actor to occur at the appropriate time. The execution continues by selecting the

earliest event in the event queue and determining which actor should receive that event. The value of that event (if any) is presented as an input to the actor, and the actor reacts (“fires”). The reaction can produce output events, and also events that simply request a later firing of the same actor at some specified time stamp.

At this point, variants of DE MoCs behave differently. Some variants, such as DEVS, require that outputs produced by the actor have a strictly larger time stamp than that of the input just presented. From a modeling perspective, every actor imposes some non-zero delay, in that its reactions (the outputs) become visible to other actors strictly later than the inputs that triggered the reaction. Other variants permit the actor to produce output events with the same time stamp as the input. That is, they can react instantaneously. As with SR models of computation, such instantaneous reactions can create significant subtleties because inputs become simultaneous with outputs.

The subtleties introduced by simultaneous events can be resolved by treating DE as a generalization of SR (Lee and Zheng, 2007). In this variant of a DE semantics, execution proceeds as follows. Again, we use an event queue and interrogate the actors for initial events to place on the queue. We select the event from the queue with the least time stamp, and all other events with the same time stamp, present those events to actors in the model as inputs, and then fire all actors in the manner of a *constructive* fixed-point iteration, as normal with SR. In this variant of the semantics, any outputs produced by an actor *must* be simultaneous with the inputs (they have the same time stamp), so they participate in the fixed point. If the actor wishes to produce an output event at a later time, it does so by requesting a firing at a later time (which results in the posting of an event on the event queue).

### 6.4.3 Continuous-Time Systems

In Chapter 2 we consider models of continuous-time systems based on ordinary differential equations (ODEs). Specifically, we consider equations of the form

$$\dot{\mathbf{x}}(t) = f(\mathbf{x}(t), t),$$

where  $\mathbf{x}: \mathbb{R} \rightarrow \mathbb{R}^n$  is a vector-valued continuous-time function. An equivalent model is an integral equation of the form

$$\mathbf{x}(t) = \mathbf{x}(0) + \int_0^t \dot{\mathbf{x}}(\tau) d\tau \tag{6.3}$$

$$= \mathbf{x}(0) + \int_0^t f(\mathbf{x}(\tau), \tau) d\tau. \tag{6.4}$$

In Chapter 2, we show that a model of a system given by such ODEs can be described as an interconnection of **actors**, where the communication between actors is via continuous-time signals. Equation (6.4) can be represented as the interconnection shown in Figure 6.16, which conforms to the feedback pattern of Figure 6.1(d).

**Example 6.17:** The feedback control system of Figure 2.3, using the helicopter model of Example 2.3, can be redrawn as shown in Figure 6.17, which conforms to the pattern of Figure 6.16. In this case,  $\mathbf{x} = \dot{\theta}_y$  is a scalar-valued continuous-time function (or a vector of length one). The function  $f$  is defined as follows,

$$f(\mathbf{x}(t), t) = (K/I_{yy})(\psi(t) - \mathbf{x}(t)),$$

and the initial value of the integrator is

$$\mathbf{x}(0) = \dot{\theta}_y(0).$$

Such models, in fact, are actor compositions under a **continuous-time model of computation**, but unlike the previous MoCs, this one cannot strictly be executed on a digital computer. A digital computer cannot directly deal with the time continuum. It can, however, be approximated, often quite accurately.

The approximate execution of a continuous-time model is accomplished by a **solver**, which constructs a numerical approximation to the solution of an ODE. The study of algorithms for solvers is quite old, with the most commonly used techniques dating back to the 19th century. Here, we will consider only one of the simplest of solvers, which is known as a **forward Euler** solver.

A forward Euler solver estimates the value of  $\mathbf{x}$  at time points  $0, h, 2h, 3h, \dots$ , where  $h$  is called the **step size**. The integration is approximated as follows,

$$\begin{aligned} \mathbf{x}(h) &= \mathbf{x}(0) + hf(\mathbf{x}(0), 0) \\ \mathbf{x}(2h) &= \mathbf{x}(h) + hf(\mathbf{x}(h), h) \\ \mathbf{x}(3h) &= \mathbf{x}(2h) + hf(\mathbf{x}(2h), 2h) \\ &\dots \\ \mathbf{x}((k+1)h) &= \mathbf{x}(kh) + hf(\mathbf{x}(kh), kh). \end{aligned}$$

This process is illustrated in Figure 6.18(a), where the “true” value of  $\dot{\mathbf{x}}$  is plotted as a function of time. The true value of  $\mathbf{x}(t)$  is the area under that curve between 0 and  $t$ , plus the initial value  $\mathbf{x}(0)$ . At the first step of the algorithm, the increment in area is approximated as the area of a rectangle of width  $h$  and height  $f(\mathbf{x}(0), 0)$ . This increment yields an estimate for  $\mathbf{x}(h)$ , which can be used to calculate  $\dot{\mathbf{x}}(h) = f(\mathbf{x}(h), h)$ , the height of the second rectangle. And so on.

You can see that the errors in approximation will accumulate over time. The algorithm can be improved considerably by two key techniques. First, a **variable-step solver** will vary the step size based on estimates of the error to keep the error small. Second, a more sophisticated solver will take into account the slope of the curve and use trapezoidal approximations as suggested in Figure 6.18(b). A family of such solvers known as Runge-Kutta solvers are widely used. But for our purposes here, it does not matter what solver is used. All that matters is that (a) the solver determines the step size, and (b) at each step, the solver performs some calculation to update the approximation to the integral.

When using such a solver, we can interpret the model in Figure 6.16 in a manner similar to SR and DE models. The  $f$  actor is [memoryless](#), so it simply performs a calculation to produce an output that depends only on the input and the current time. The integrator is a [state machine](#) whose state is updated at each reaction by the solver, which uses the input to determine what the update should be. The state space of this state machine is infinite, since the state variable  $\mathbf{x}(t)$  is a vector of real numbers.

Hence, a continuous-time model can be viewed as an [SR](#) model with a time step between global reactions determined by a solver ([Lee and Zheng, 2007](#)). Specifically, a continuous-time model is a network of actors, each of which is a cascade composition of a simple memoryless computation actor and a state machine, and the actor reactions are [simultaneous and instantaneous](#). The times of the reactions are determined by a solver. The solver will typically consult the actors in determining the time step, so that for example events like level crossings (when a continuous signal crosses a threshold) can be captured precisely. Hence, despite the additional complication of having to provide a solver, the mechanisms required to achieve a continuous-time model of computation are not much different from those required to achieve SR and DE.

A popular software tool that uses a continuous-time MoC is Simulink, from The MathWorks. Simulink represents models similarly as block diagrams, which are interconnections of actors. Continuous-time models can also be simulated using the textual tool MATLAB from the same vendor. MATRIXx, from National Instruments, also supports graphical continuous-time modeling. Continuous-time models can also be integrated within

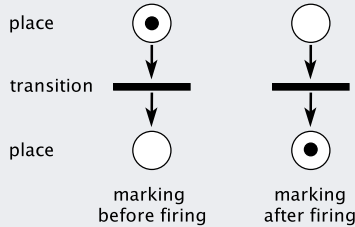
LabVIEW models, either graphically using the Control Design and Simulation Module or textually using the programming language MathScript.

## 6.5 Summary

This chapter provides a whirlwind tour of a rather large topic, concurrent models of computation. It begins with synchronous-reactive models, which are closest to the synchronous composition of state machines considered in the previous chapter. It then considers dataflow models, where execution can be more loosely coordinated. Only data precedences impose constraints on the order of actor computations. The chapter then concludes with a quick view of a few models of computation that explicitly include a notion of time. Such MoCs are particularly useful for modeling cyber-physical systems.

## Petri Nets

**Petri nets**, named after Carl Adam Petri, are a popular modeling formalism related to [dataflow](#) (Murata, 1989). They have two types of elements, **places** and **transitions**, depicted as white circles and rectangles, respectively:



A place can contain any number of tokens, depicted as black circles. A transition is **enabled** if all places connected to it as inputs contain at least one token. Once a transition is enabled, it can **fire**, consuming one token from each input place and putting one token on each output place. The state of a network, called its **marking**, is the number of tokens on each place in the network. The figure above shows a simple network with its marking before and after the firing of the transition. If a place provides input to more than one transition, then the network is nondeterministic. A token on that place may trigger a firing of either destination transition.

An example of a Petri net model is shown in Figure 6.15, which models two concurrent programs with a [mutual exclusion](#) protocol. Each of the two programs has a critical section, meaning that only one of the programs can be in its critical section at any time. In the model, program A is in its critical section if there is a token on place a2, and program B is in its critical section if there is a token on place b1. The job of the mutual exclusion protocol is to ensure that these two places cannot simultaneously have a token.

If the initial marking of the model is as shown in the figure, then both top transitions are enabled, but only one can fire (there is only one token in the place labeled mutex). Which one fires is chosen nondeterministically. Suppose program A fires. After this firing, there will be a token in place a2, so the corresponding bottom transition becomes enabled. Once that transition fires, the model returns to its initial marking. It is easy to see that the mutual exclusion protocol is correct in this model.

Unlike dataflow buffers, places do not preserve an ordering of tokens. Petri nets with a finite number of markings are equivalent to [FSMs](#).



## Models of Time

How to model physical time is surprisingly subtle. How should we define simultaneity across a distributed system? A thoughtful discussion of this question is considered by Galison (2003). What does it mean for one event to cause another? Can an event that causes another be simultaneous with it? Several thoughtful essays on this topic are given in Price and Corry (2007).

In Chapter 2, we assume time is represented by a variable  $t \in \mathbb{R}$  or  $t \in \mathbb{R}_+$ . This model is sometimes referred to as **Newtonian time**. It assumes a globally shared absolute time, where any reference anywhere to the variable  $t$  will yield the same value. This notion of time is often useful for modeling even if it does not perfectly reflect physical realities, but it has its deficiencies. Consider for example Newton's cradle, a toy with five steel balls suspended by strings. If you lift one ball and release it, it strikes the second ball, which does not move. Instead, the fifth ball reacts by rising. Consider the momentum of the middle ball as a function of time. The middle ball does not move, so its momentum must be everywhere zero. But the momentum of the first ball is somehow transferred to the fifth ball, passing through the middle ball. So the momentum cannot be always zero. Let  $m: \mathbb{R} \rightarrow \mathbb{R}$  represent the momentum of this ball and  $\tau$  be the time of the collision. Then

$$m(t) = \begin{cases} M & \text{if } t = \tau \\ 0 & \text{otherwise} \end{cases}$$

for all  $t \in \mathbb{R}$ . In a cyber-physical system, we may, however, want to represent this function in software, in which case a sequence of samples will be needed. But how can such sample unambiguously represent the rather unusual structure of this signal?

One option is to use **superdense time** (Manna and Pnueli, 1993; Maler et al., 1992; Lee and Zheng, 2005; Cataldo et al., 2006), where instead of  $\mathbb{R}$ , time is represented by a set  $\mathbb{R} \times \mathbb{N}$ . A time value is a tuple  $(t, n)$ , where  $t$  represents Newtonian time and  $n$  represents a sequence index within an instant. In this representation, the momentum of the middle ball can be unambiguously represented by a sequence where  $m(\tau, 0) = 0$ ,  $m(\tau, 1) = M$ , and  $m(\tau, 2) = 0$ . Such a representation also handles events that are **simultaneous and instantaneous** but also causally related.

Another alternative is **partially ordered time**, where two time values may or may not be ordered relative to each other. When there is a chain of causal relationships between them, then they must be ordered, and otherwise not.

### Probing Further: Discrete Event Semantics

Discrete-event models of computation have been a subject of study for many years, with several textbooks available (Zeigler et al., 2000; Cassandras, 1993; Fishman, 2001). The subtleties in the semantics are considerable (see Lee (1999); Cataldo et al. (2006); Liu et al. (2006); Liu and Lee (2008)). Instead of discussing the formal semantics here, we describe how a DE model is executed. Such a description is, in fact, a valid way of giving the semantics of a model. The description is called an **operational semantics** (Scott and Strachey, 1971; Plotkin, 1981).

DE models are often quite large and complex, so execution performance becomes very important. Because of the use of a single event queue, parallelizing or distributing execution of DE models can be challenging (Misra, 1986; Fujimoto, 2000). A recently proposed strategy called **PTIDES** (for programming temporally integrated distributed embedded systems), leverages network time synchronization to provide efficient distributed execution (Zhao et al., 2007; Lee et al., 2009). The claim is that the execution is efficient enough that DE can be used not only as a simulation technology, but also as an implementation technology. That is, the DE event queue and execution engine become part of the deployed embedded software. As of this writing, that claim has not been proven on any practical examples.

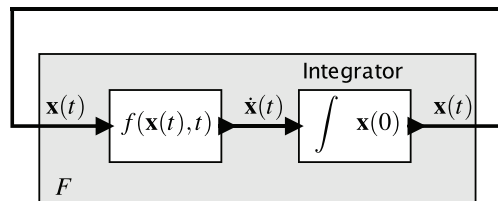


Figure 6.16: Actor model of a system described by equation (6.4).

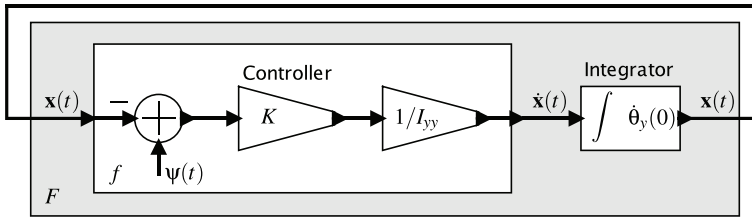


Figure 6.17: The feedback control system of Figure 2.3, using the helicopter model of Example 2.3, redrawn to conform to the pattern of Figure 6.16.

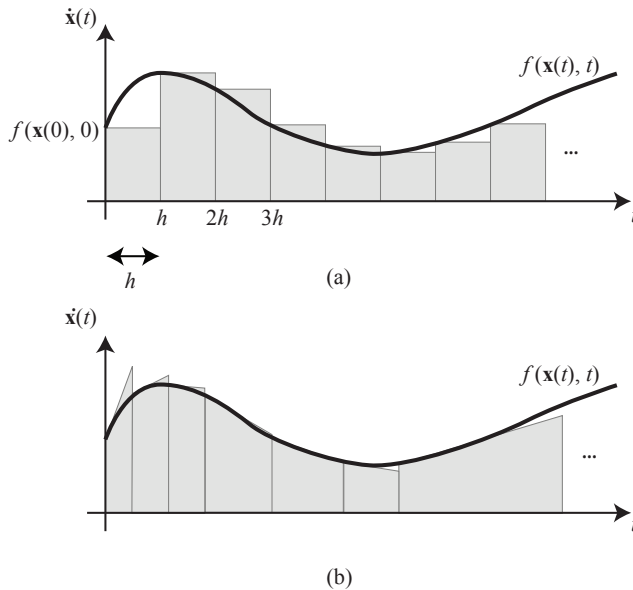
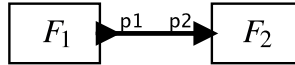


Figure 6.18: (a) Forward Euler approximation to the integration in (6.4), where  $x$  is assumed to be a scalar. (b) A better approximation that uses a variable step size and takes into account the slope of the curve.

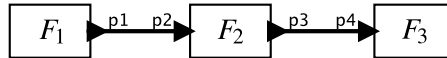
## Exercises

- Show how each of the following actor models can be transformed into a **feedback** system by using a reorganization similar to that in Figure 6.1(b). That is, the actors should be aggregated into a single side-by-side composite actor.

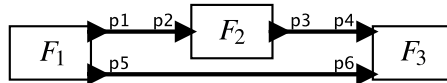
(a)



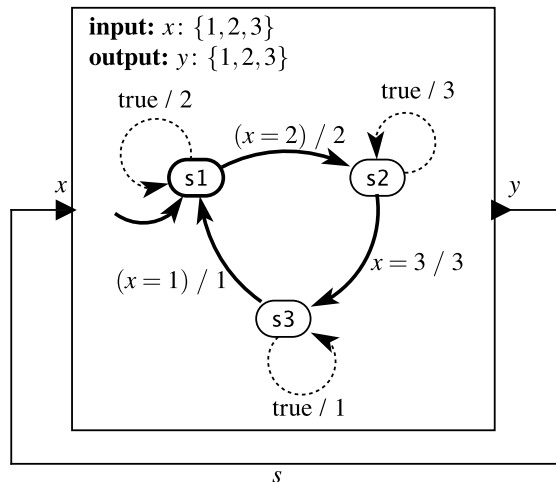
(b)



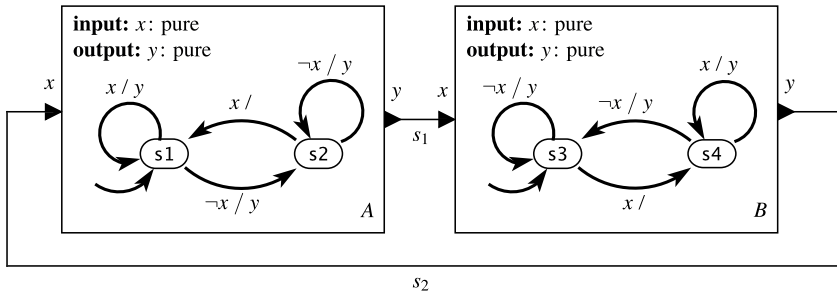
(c)



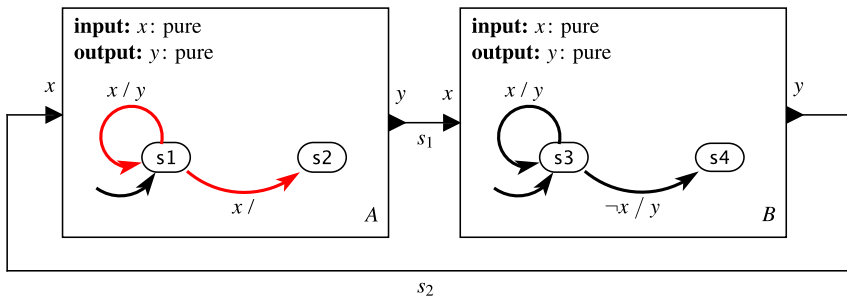
- Consider the following state machine in a synchronous feedback composition:



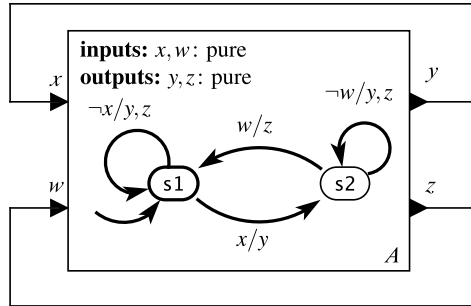
- (a) Is it well-formed? Is it constructive?
  - (b) If it is well-formed and constructive, then find the output symbols for the first 10 reactions. If not, explain where the problem is.
  - (c) Show the composition machine, assuming that the composition has no input and that the only output is  $y$ .
3. For the following synchronous model, determine whether it is well formed and constructive, and if so, determine the sequence of values of the signals  $s_1$  and  $s_2$ .



4. For the following synchronous model, determine whether it is well formed and constructive, and if so, determine the possible sequences of values of the signals  $s_1$  and  $s_2$ . Note that machine A is nondeterministic.



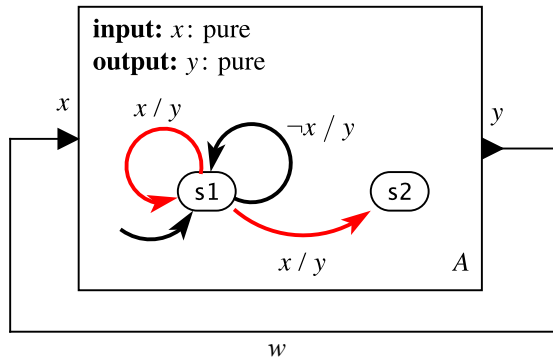
5. (a) Determine whether the following synchronous model is well formed and constructive:



Explain.

- (b) For the model in part (a), give the semantics by giving an equivalent flat state machine with no inputs and two outputs.

6. Consider the following synchronous feedback composition:

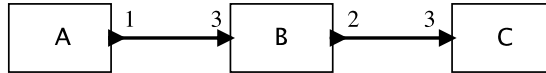


Notice that the FSM  $A$  is nondeterministic.

- (a) Is this composition well formed? Constructive? Explain.
- (b) Give an equivalent flat FSM (with no input and no connections) that produces exactly the same possible sequences  $w$ .
7. Recall the traffic light controller of Figure 3.10. Consider connecting the outputs of this controller to a pedestrian light controller, whose FSM is given in Figure 5.10. Using your favorite modeling software that supports state machines (such as Ptolemy II, LabVIEW Statecharts, or Simulink/Stateflow), construct the composition of the above two FSMs along with a deterministic **extended state machine**

modeling the environment and generating input symbols *timeR*, *timeG*, *timeY*, and *isCar*. For example, the environment FSM can use an internal counter to decide when to generate these symbols.

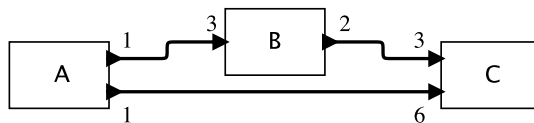
8. Consider the following SDF model:



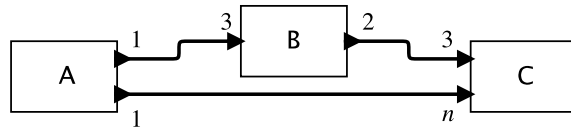
The numbers adjacent to the ports indicate the number of tokens produced or consumed by the actor when it fires. Answer the following questions about this model.

- (a) Let  $q_A$ ,  $q_B$ , and  $q_C$  denote the number of firings of actors A, B, and C, respectively. Write down the balance equations and find the least positive integer solution.
  - (b) Find a schedule for an unbounded execution that minimizes the buffer sizes on the two communication channels. What is the resulting size of the buffers?
9. For each of the following dataflow models, determine whether there is an unbounded execution with bounded buffers. If there is, determine the minimum buffer size.

- (a)

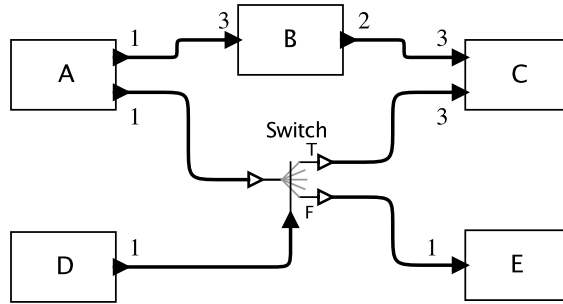


(b)



where  $n$  is some integer.

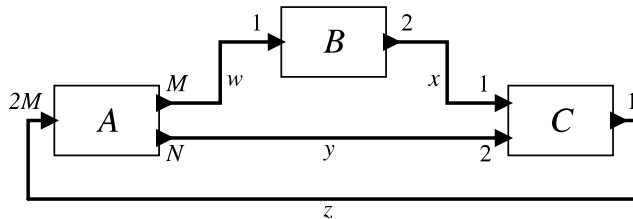
(c)



where D produces an arbitrary boolean sequence.

- (d) For the same dataflow model as in part (c), assume you can specify a periodic boolean output sequence produced by D. Find such a sequence that yields bounded buffers, give a schedule that minimizes buffer sizes, and give the buffer sizes.

10. Consider the SDF graph shown below:



In this figure,  $A$ ,  $B$ , and  $C$  are actors. Adjacent to each port is the number of tokens consumed or produced by a firing of the actor on that port, where  $N$  and  $M$  are variables with positive integer values. Assume the variables  $w$ ,  $x$ ,  $y$ , and  $z$  represent the number of initial tokens on the connection where these variables appear in the diagram. These variables have non-negative integer values.



- (a) Derive a simple relationship between  $N$  and  $M$  such that the model is consistent, or show that no positive integer values of  $N$  and  $M$  yield a consistent model.
- (b) Assume that  $w = x = y = 0$  and that the model is consistent and find the minimum value of  $z$  (as a function  $N$  and  $M$ ) such that the model does not deadlock.
- (c) Assume that  $z = 0$  and that the model is consistent. Find values for  $w$ ,  $x$ , and  $y$  such that the model does not deadlock and  $w + x + y$  is minimized.
- (d) Assume that  $w = x = y = 0$  and  $z$  is whatever value you found in part (b). Let  $b_w$ ,  $b_x$ ,  $b_y$ , and  $b_z$  be the buffer sizes for connections  $w$ ,  $x$ ,  $y$ , and  $z$ , respectively. What is the minimum for these buffer sizes?

## Part II

# Design of Embedded Systems

This part of this text studies the [design](#) of embedded systems, with emphasis on the techniques used to build [concurrent](#), [real-time](#) embedded software. We proceed bottom up, discussing first in Chapter [7](#) sensors and actuators, with emphasis on how to model them. Chapter [8](#) covers the design of embedded processors, with emphasis on parallelism in the hardware and its implications for programmers. Chapter [9](#) covers memory architectures, with particular emphasis on the effect they have on program timing. Chapter [10](#) covers the input and output mechanisms that enable programs to interact with the external physical world, with emphasis on how to reconcile the sequential nature of software with the concurrent nature of the physical world. Chapter [11](#) describes mechanisms for achieving concurrency in software, threads and processes, and synchronization of concurrent software tasks, including semaphores and mutual exclusion. Finally, Chapter [12](#) covers scheduling, with particular emphasis on controlling timing in concurrent programs.

# Sensors and Actuators

<b>7.1</b>	<b>Models of Sensors and Actuators</b>	<b>181</b>
7.1.1	Linear and Affine Models	181
7.1.2	Range	182
7.1.3	Dynamic Range	183
7.1.4	Quantization	183
7.1.5	Noise	186
7.1.6	Sampling	188
	<i>Sidebar: Decibels</i>	189
7.1.7	Harmonic Distortion	192
7.1.8	Signal Conditioning	193
<b>7.2</b>	<b>Common Sensors</b>	<b>195</b>
7.2.1	Measuring Tilt and Acceleration	195
7.2.2	Measuring Position and Velocity	197
7.2.3	Measuring Rotation	199
7.2.4	Measuring Sound	199
7.2.5	Other Sensors	200
<b>7.3</b>	<b>Actuators</b>	<b>200</b>
7.3.1	Light-Emitting Diodes	200
7.3.2	Motor Control	202
<b>7.4</b>	<b>Summary</b>	<b>206</b>
	<b>Exercises</b>	<b>207</b>

A **sensor** is a device that measures a physical quantity. An **actuator** is a device that alters a physical quantity. In electronic systems, sensors often produce a voltage that is proportional to the physical quantity being measured. The voltage may then be converted

---

to a number by an **analog-to-digital converter (ADC)**. A sensor that is packaged with an ADC is called a **digital sensor**, whereas a sensor without an ADC is called an **analog sensor**. A digital sensor will have a limited precision, determined by the number of bits used to represent the number (this can be as few as one!). Conversely, an actuator is commonly driven by a voltage that may be converted from a number by a **digital-to-analog converter (DAC)**. An actuator that is packaged with a DAC is called a **digital actuator**.

Today, sensors and actuators are often packaged with microprocessors and network interfaces, enabling them to appear on the Internet as services. The trend is towards a technology that deeply connects our physical world with our information world through such smart sensors and actuators. This integrated world is variously called the **Internet of Things (IoT)**, **Industry 4.0**, the **Industrial Internet**, **Machine-to-Machine (M2M)**, the **Internet of Everything**, the **Smarter Planet**, **TSensors** (Trillion Sensors), or **The Fog** (like The Cloud, but closer to the ground).

Some technologies for interfacing to sensors and actuators have emerged that leverage established mechanisms originally developed for ordinary Internet usage. For example, a sensor or actuator may be accessible via a web server using the so-called **Representational State Transfer (REST)** architectural style (Fielding and Taylor, 2002). In this style, data may be retrieved from a sensor or commands may be issued to an actuator by constructing a URL (uniform resource locator), as if you were accessing an ordinary web page from a browser, and then transmitting the URL directly to the sensor or actuator device, or to a web server that serves as an intermediary.

In this chapter, we focus not on such high-level interfaces, but rather on foundational properties of sensors and actuators as bridges between the physical and the cyber worlds. Key low-level properties include the rate at which measurements are taken or actuations are performed, the proportionality constant that relates the physical quantity to the measurement or control signal, the offset or bias, and the dynamic range. For many sensors and actuators, it is useful to model the degree to which a sensor or actuator deviates from a proportional measurement (its **nonlinearity**), and the amount of random variation introduced by the measurement process (its **noise**).

A key concern for sensors and actuators is that the physical world functions in a multidimensional continuum of time and space. It is an **analog** world. The world of software, however, is **digital**, and strictly quantized. Measurements of physical phenomena must be quantized in both magnitude and time before software can operate on them. And

commands to the physical world that originate from software will also be intrinsically quantized. Understanding the effects of this quantization is essential.

This chapter begins in Section 7.1 with an outline of how to construct models of sensors and actuators, specifically focusing on linearity (and nonlinearity), bias, dynamic range, quantization, noise, and sampling. That section concludes with a brief introduction to signal conditioning, a signal processing technique to improve the quality of sensor data and actuator controls. Section 7.2 then discusses a number of common sensing problems, including measuring tilt and acceleration (accelerometers), measuring position and velocity (anemometers, inertial navigation, GPS, and other ranging and triangulation techniques), measuring rotation (gyroscopes), measuring sound (microphones), and measuring distance (rangefinders). The chapter concludes with Section 7.3, which shows how to apply the modeling techniques to actuators, focusing specifically on LEDs and motor controllers.

Higher-level properties that are not addressed in this chapter, but are equally important, include security (specifically access control), privacy (particularly for data flowing over the open Internet), name-space management, and commissioning. The latter is particularly big issue when the number of sensors or actuators get large. Commissioning is the process of associating a sensor or actuator device with a physical location (e.g., a temperature sensor gives the temperature of what?), enabling and configuring network interfaces, and possibly calibrating the device for its particular environment.

## 7.1 Models of Sensors and Actuators

Sensors and actuators connect the cyber world with the physical world. Numbers in the cyber world bear a relationship with quantities in the physical world. In this section, we provide models of that relationship. Having a good model of a sensor or actuator is essential to effectively using it.

### 7.1.1 Linear and Affine Models

Many sensors may be approximately modeled by an affine function. Suppose that a physical quantity  $x(t)$  at time  $t$  is reported by the sensor to have value  $f(x(t))$ , where  $f: \mathbb{R} \rightarrow \mathbb{R}$  is a function. The function  $f$  is **linear** if there exists a **proportionality con-**

**stant**  $a \in \mathbb{R}$  such that for all  $x(t) \in \mathbb{R}$

$$f(x(t)) = ax(t).$$

It is an **affine function** if there exists a proportionality constant  $a \in \mathbb{R}$  and a **bias**  $b \in \mathbb{R}$  such that

$$f(x(t)) = ax(t) + b. \quad (7.1)$$

Clearly, every linear function is an affine function (with  $b = 0$ ), but not vice versa.

Interpreting the readings of such a sensor requires knowledge of the proportionality constant and bias. The proportionality constant represents the **sensitivity** of the sensor, since it specifies the degree to which the measurement changes when the physical quantity changes.

Actuators may also be modeled by affine functions. The affect that a command to the actuator has on the physical environment may be reasonably approximated by a relation like (7.1).

### 7.1.2 Range

No sensor or actuator truly realizes an affine function. In particular, the **range** of a sensor, the set of values of a physical quantity that it can measure, is always limited. Similarly for an actuator. Outside that range, an affine function model is no longer valid. For example, a thermometer designed for weather monitoring may have a range of  $-20^\circ$  to  $50^\circ$  Celsius. Physical quantities outside this range will typically **saturate**, meaning that they yield a maximum or a minimum reading outside their range. An **affine function** model of a sensor may be augmented to take this into account as follows,

$$f(x(t)) = \begin{cases} ax(t) + b & \text{if } L \leq x(t) \leq H \\ aH + b & \text{if } x(t) > H \\ aL + b & \text{if } x(t) < L, \end{cases} \quad (7.2)$$

where  $L, H \in \mathbb{R}$ ,  $L < H$ , are the low and high end of the sensor range, respectively.

A relation between a physical quantity  $x(t)$  and a measurement given by (7.2) is not an affine relation (it is, however, piecewise affine). In fact, this is a simple form of **non-linearity** that is shared by all sensors. The sensor is reasonably modeled by an affine function within an **operating range**  $(L, H)$ , but outside that operating range, its behavior is distinctly different.

### 7.1.3 Dynamic Range

Digital sensors are unable to distinguish between two closely-spaced values of the physical quantity. The **precision**  $p$  of a sensor is the smallest absolute difference between two values of a physical quantity whose sensor readings are distinguishable. The **dynamic range**  $D \in \mathbb{R}_+$  of a digital sensor is the ratio

$$D = \frac{H - L}{p},$$

where  $H$  and  $L$  are the limits of the range in (7.2). Dynamic range is usually measured in **decibels** (see sidebar on page 189), as follows:

$$D_{dB} = 20 \log_{10} \left( \frac{H - L}{p} \right). \quad (7.3)$$

### 7.1.4 Quantization

A digital sensor represents a physical quantity using an  $n$ -bit number, where  $n$  is a small integer. There are only  $2^n$  distinct such numbers, so such a sensor can produce only  $2^n$  distinct measurements. The actual physical quantity may be represented by a real number  $x(t) \in \mathbb{R}$ , but for each such  $x(t)$ , the sensor must pick one of the  $2^n$  numbers to represent it. This process is called **quantization**. For an ideal digital sensor, two physical quantities that differ by the precision  $p$  will be represented by digital quantities that differ by one bit, so precision and quantization become intertwined.

We can further augment the function  $f$  in (7.2) to include quantization, as illustrated in the following example.

**Example 7.1:** Consider a 3-bit digital sensor that can measure a voltage between zero and one volt. Such a sensor may be modeled by the function  $f: \mathbb{R} \rightarrow \{0, 1, \dots, 7\}$  shown in Figure 7.1. The horizontal axis is the input to the sensor (in volts), and the vertical axis is the output, with the value shown in binary to emphasize that this is a 3-bit digital sensor.

In the figure, the low end of the measurable range is  $L = 0$ , and the high end is  $H = 1$ . The precision is  $p = 1/8$ , because within the operating range, any

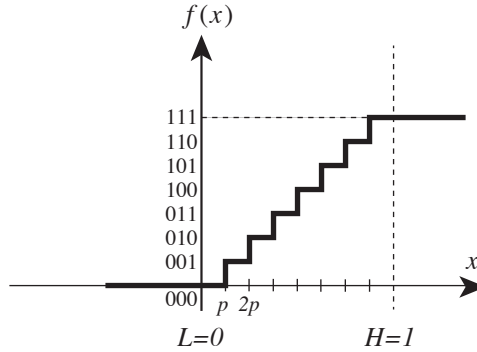


Figure 7.1: Sensor distortion function for a 3-bit digital sensor capable of measuring a range of zero to one volt, where the precision  $p = 1/8$ .

two inputs that differ by more than  $1/8$  of a volt will yield different outputs. The dynamic range, therefore, is

$$D_{dB} = 20 \log_{10} \left( \frac{H - L}{p} \right) \approx 18dB.$$

A function  $f$  like the one in Figure 7.1 that defines the output of a sensor as a function of its input is called the **sensor distortion function**. In general, an ideal  $n$ -bit digital sensor with a sensor distortion function like that shown in Figure 7.1 will have a precision given by

$$p = (H - L)/2^n$$

and a dynamic range of

$$D_{dB} = 20 \log_{10} \left( \frac{H - L}{p} \right) = 20 \log_{10}(2^n) = 20n \log_{10}(2) \approx 6n \text{ dB}. \quad (7.4)$$

Each additional bit yields approximately 6 decibels of dynamic range.



**Example 7.2:** An extreme form of quantization is performed by an **analog comparator**, which compares a signal value against a threshold and produces a binary value, zero or one. Here, the sensor function  $f: \mathbb{R} \rightarrow \{0, 1\}$  is given by

$$f(x(t)) = \begin{cases} 0 & \text{if } x(t) \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

Such extreme quantization is often useful, because the resulting signal is a very simple digital signal that can be connected directly to a **GPIO** input pin of a microprocessor, as discussed in Chapter 10.

The analog comparator of the previous example is a one-bit ADC. The quantization error is high for such a converter, but using **signal conditioning**, as described below in Section 7.1.8, if the **sample rate** is high enough, the noise can be reduced considerably by digital low-pass filtering. Such a process is called **oversampling**; it is commonly used today because processing signals digitally is often less costly than analog processing.

Actuators are also subject to quantization error. A digital actuator takes a digital command and converts it to an analog physical action. A key part of this is the digital to analog converter (DAC). Because the command is digital, it has only a finite number of possible values. The precision with which an analog action can be taken, therefore, will depend on the number of bits of the digital signal and the range of the actuator.

As with ADCs, however, it is possible to trade off precision and speed. A **bang-bang controller**, for example, uses a one-bit digital actuation signal to drive an actuator, but updates that one-bit command very quickly. An actuator with a relatively slow response time, such as a motor, does not have much time to react to each bit, so the reaction to each bit is small. The overall reaction will be an average of the bits over time, much smoother than what you would expect from a one-bit control. This is the mirror image of oversampling.

The design of ADC and DAC hardware is itself quite an art. The effects of choices of sampling interval and number of bits are quite nuanced. Considerable expertise in signal processing is required to fully understand the implications of choices (see [Lee and Varaiya \(2011\)](#)). Below, we give a cursory view of this rather sophisticated topic. Section 7.1.8 discusses how to mitigate the noise in the environment and noise due to quantization,

showing the intuitive result that it is beneficial to filter out frequency ranges that are not of interest. These frequency ranges are related to the sample rate. Hence, noise and sampling are the next topics.

### 7.1.5 Noise

By definition, **noise** is the part of a signal that we do not want. If we want to measure  $x(t)$  at time  $t$ , but we actually measure  $x'(t)$ , then the noise is the difference,

$$n(t) = x'(t) - x(t).$$

Equivalently, the actual measurement is

$$x'(t) = x(t) + n(t), \tag{7.5}$$

a sum of what we want plus the noise.

**Example 7.3:** Consider using an [accelerometer](#) to measure the orientation of a slowly moving object (see Section 7.2.1 below for an explanation of why an accelerometer can measure orientation). The accelerometer is attached to the moving object and reacts to changes in orientation, which change the direction of the gravitational field with respect to the axis of the accelerometer. But it will also report acceleration due to vibration. Let  $x(t)$  be the signal due to orientation and  $n(t)$  be the signal due to vibration. The accelerometer measures the sum.

In the above example, noise is a side effect of the fact that the sensor is not measuring exactly what we want. We want orientation, but it is measuring acceleration. We can also model sensor imperfections and quantization as noise. In general, a sensor distortion function can be modeled as additive noise,

$$f(x(t)) = x(t) + n(t), \tag{7.6}$$

where  $n(t)$  by definition is just  $f(x(t)) - x(t)$ .

It is useful to be able to characterize how much noise there is in a measurement. The **root mean square (RMS)**  $N \in \mathbb{R}_+$  of the noise is equal to the square root of the average value of  $n(t)^2$ . Specifically,

$$N = \lim_{T \rightarrow \infty} \sqrt{\frac{1}{2T} \int_{-T}^T (n(\tau))^2 d\tau}. \quad (7.7)$$

This is a measure of (the square root of) **noise power**. An alternative (statistical) definition of noise power is the square root of the expected value of the square of  $n(t)$ . Formula (7.7) defines the noise power as an *average* over time rather than an expected value.

The **signal to noise ratio (SNR)**, in decibels) is defined in terms of RMS noise,

$$SNR_{dB} = 20 \log_{10} \left( \frac{X}{N} \right),$$

where  $X$  is the RMS value of the input signal  $x$  (again, defined either as a time average as in (7.7) or using the expected value). In the next example, we illustrate how to calculate SNR using expected values, leveraging elementary probability theory.

**Example 7.4:** We can find the SNR that results from quantization by using (7.6) as a model of the quantizer. Consider Example 7.1 and Figure 7.1, which show a 3-bit digital sensor with an operating range of zero to one volt. Assume that the input voltage is equally likely to be anywhere in the range of zero to one volt. That is,  $x(t)$  is a random variable with uniform distribution ranging from 0 to 1. Then the RMS value of the input  $x$  is given by the square root of the expected value of the square of  $x(t)$ , or

$$X = \sqrt{\int_0^1 x^2 dx} = \frac{1}{\sqrt{3}}.$$

Examining Figure 7.1, we see that if  $x(t)$  is a random variable with uniform distribution ranging from 0 to 1, then the error  $n(t)$  in the measurement (7.6) is equally likely to be anywhere in the range from  $-1/8$  to 0. The RMS noise is therefore given by

$$N = \sqrt{\int_{-1/8}^0 8n^2 dn} = \sqrt{\frac{1}{3 \cdot 64}} = \frac{1}{8\sqrt{3}}.$$

The SNR is therefore

$$SNR_{dB} = 20 \log_{10} \left( \frac{X}{N} \right) = 20 \log_{10} (8) \approx 18dB.$$

Notice that this matches the 6 dB per bit dynamic range predicted by (7.4)!

To calculate the SNR in the previous example, we needed a statistical model of the input  $x$  (uniformly distributed from 0 to 1) and the quantization function. In practice, it is difficult to calibrate ADC hardware so that the input  $x$  makes full use of its range. That is, the input is likely to be distributed over less than the full range 0 to 1. It is also unlikely to be uniformly distributed. Hence, the actual SNR achieved in a system will likely be considerably less than the 6 dB per bit predicted by (7.4).

### 7.1.6 Sampling

A physical quantity  $x(t)$  is a function of time  $t$ . A digital sensor will **sample** the physical quantity at particular points in time to create a **discrete signal**. In **uniform sampling**, there is a fixed time interval  $T$  between samples;  $T$  is called the **sampling interval**. The resulting signal may be modeled as a function  $s: \mathbb{Z} \rightarrow \mathbb{R}$  defined as follows,

$$\forall n \in \mathbb{Z}, \quad s(n) = f(x(nT)), \quad (7.8)$$

where  $\mathbb{Z}$  is the set of integers. That is, the physical quantity  $x(t)$  is observed only at times  $t = nT$ , and the measurement is subjected to the sensor distortion function. The **sampling rate** is  $1/T$ , which has units of **samples per second**, often given as **Hertz** (written **Hz**, meaning cycles per second).

In practice, the smaller the sampling interval  $T$ , the more costly it becomes to provide more bits in an ADC. At the same cost, faster ADCs typically produce fewer bits and hence have either higher quantization error or smaller range.

**Example 7.5:** The **ATSC** digital video coding standard includes a format where the frame rate is 30 frames per second and each frame contains  $1080 \times 1920 =$

## Decibels

The term “**decibel**” is literally one tenth of a **bel**, which is named after Alexander Graham Bell. This unit of measure was originally developed by telephone engineers at Bell Telephone Labs to designate the ratio of the **power** of two signals.

Power is a measure of energy dissipation (work done) per unit time. It is measured in **watts** for electronic systems. One bel is defined to be a factor of 10 in power. Thus, a 1000 watt hair dryer dissipates 1 bel, or 10 dB, more power than a 100 watt light bulb. Let  $p_1 = 1000$  watts be the power of the hair dryer and  $p_2 = 100$  be the power of the light bulb. Then the ratio is

$$\log_{10}(p_1/p_2) = 1 \text{ bel, or}$$

$$10 \log_{10}(p_1/p_2) = 10 \text{ dB.}$$

Comparing against (7.3) we notice a discrepancy. There, the multiplying factor is 20, not 10. That is because the ratio in (7.3) is a ratio of amplitude (magnitude), not powers. In electronic circuits, if an amplitude represents the voltage across a resistor, then the power dissipated by the resistor is proportional to the *square* of the amplitude. Let  $a_1$  and  $a_2$  be two such amplitudes. Then the ratio of their powers is

$$10 \log_{10}(a_1^2/a_2^2) = 20 \log_{10}(a_1/a_2).$$

Hence the multiplying factor of 20 instead of 10 in (7.3). A 3 dB power ratio amounts to a factor of 2 in power. In amplitudes, this is a ratio of  $\sqrt{2}$ .

In audio, decibels are used to measure sound pressure. A statement like “a jet engine at 10 meters produces 120 dB of sound,” by convention, compares sound pressure to a defined reference of 20 micropascals, where a pascal is a pressure of 1 newton per square meter. For most people, this is approximately the threshold of hearing at 1 kHz. Thus, a sound at 0 dB is barely audible. A sound at 10 dB has 10 times the power. A sound at 100 dB has  $10^{10}$  times the power. The jet engine, therefore, would probably make you deaf without ear protection.

2,073,600 pixels. An ADC that is converting one color channel to a digital representation must therefore perform  $2,073,600 \times 30 = 62,208,000$  conversions per second, which yields a sampling interval  $T$  of approximately 16 nsec. With such a short sampling interval, increasing the number of bits in the ADC becomes expensive. For video, a choice of  $b = 8$  bits is generally adequate to yield good visual fidelity and can be realized at reasonable cost.

An important concern when sampling signals is that there are many distinct functions  $x$  that when sampled will yield the same signal  $s$ . This phenomenon is known as **aliasing**.

**Example 7.6:** Consider a sinusoidal sound signal at 1 kHz (kilohertz, or thousands of cycles per second),

$$x(t) = \cos(2000\pi t).$$

Suppose there is no sensor distortion, so the function  $f$  in (7.8) is the identity function. If we sample at 8000 samples per second (a rate commonly used in telephony), we get a sampling interval of  $T = 1/8000$ , which yields the samples

$$s(n) = f(x(nT)) = \cos(\pi n/4).$$

Suppose instead that we are given a sound signal at 9 kHz,

$$x'(t) = \cos(18,000\pi t).$$

Sampling at the same 8kHz rate yields

$$s'(n) = \cos(9\pi n/4) = \cos(\pi n/4 + 2\pi n) = \cos(\pi n/4) = s(n).$$

The 1 kHz and 9 kHz sound signals yield exactly the same samples, as illustrated in Figure 7.2. Hence, at this sampling rate, these two signals are aliases of one another. They cannot be distinguished.

Aliasing is a complex and subtle phenomenon (see [Lee and Varaiya \(2011\)](#) for details), but a useful rule of thumb for uniform sampling is provided by the **Nyquist-Shannon sampling theorem**. A full study of the subject requires the machinery of Fourier transforms, and is beyond the scope of this text. Informally, this theorem states that a set of samples at sample rate  $R = 1/T$  uniquely defines a continuous-time signal that is a sum of sinusoidal components with frequencies less than  $R/2$ . That is, among all continuous-time signals that are sums of sinusoids with frequencies less than  $R/2$ , there is only one that matches any given set of samples taken at rate  $R$ . The rule of thumb, therefore, is that if you sample a signal where the most rapid expected variation occurs at frequency  $R/2$ , then sampling the signal at a rate at least  $R$  will result in samples that uniquely represent the signal.

**Example 7.7:** In traditional telephony, engineers have determined that intelligible human speech signals do not require frequencies higher than 4 kHz. Hence, removing the frequencies above 4 kHz and sampling an audio signal with human speech at 8kHz is sufficient to enable reconstruction of an intelligible audio signal from the samples. The removal of the high frequencies is accomplished by a frequency selective filter called an **anti-aliasing filter**, because it prevents frequency components above 4 kHz from masquerading as frequency components below 4 kHz.

The human ear, however, can easily discern frequencies up to about 15 kHz, or 20 kHz in young people. Digital audio signals intended for music, therefore, are sampled at frequencies above 40 kHz; 44.1 kHz is a common choice, a rate defined originally for use in compact discs (CDs).

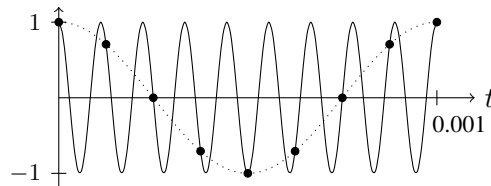


Figure 7.2: Illustration of aliasing, where samples of a 9 kHz sinusoid taken at 8,000 samples per second are the same as samples of a 1 kHz sinusoid taken at 8,000 samples per second.

**Example 7.8:** Air temperature in a room varies quite slowly compared to sound pressure. We might assume, for example, that the most rapid expected air temperature variations occur at rates measured in minutes, not seconds. If we want to capture variations on the scale of a minute or so, then we should take at least two samples per minute of the temperature measurement.

### 7.1.7 Harmonic Distortion

A form of **nonlinearity** that occurs even within the **operating range** of sensors and actuators is **harmonic distortion**. It typically occurs when the **sensitivity** of the sensor or actuator is not constant and depends on the magnitude of the signal. For example, a microphone may be less responsive to high sound pressure than to lower sound pressure.

Harmonic distortion is a nonlinear effect that can be modeled by powers of the physical quantity. Specifically, **second harmonic distortion** is a dependence on the square of the physical quantity. That is, given a physical quantity  $x(t)$ , the measurement is modeled as

$$f(x(t)) = ax(t) + b + d_2(x(t))^2, \quad (7.9)$$

where  $d_2$  is the amount of second harmonic distortion. If  $d_2$  is small, then the model is nearly affine. If  $d_2$  is large, then it is far from affine. The  $d_2(x(t))^2$  term is called second harmonic distortion because of the effect it has the frequency content of a signal  $x(t)$  that is varying in time.

**Example 7.9:** Suppose that a **microphone** is stimulated by a purely sinusoidal input sound

$$x(t) = \cos(\omega_0 t),$$

where  $t$  is time in seconds and  $\omega_0$  is the frequency of the sinusoid in radians per second. If the frequency is within the human auditory range, then this will sound like a pure tone.



A sensor modeled by (7.9) will produce at time  $t$  the measurement

$$\begin{aligned} x'(t) &= ax(t) + b + d_2(x(t))^2 \\ &= a \cos(\omega_0 t) + b + d_2 \cos^2(\omega_0 t) \\ &= a \cos(\omega_0 t) + b + \frac{d_2}{2} + \frac{d_2}{2} \cos(2\omega_0 t), \end{aligned}$$

where we have used the trigonometric identity

$$\cos^2(\theta) = \frac{1}{2}(1 + \cos(2\theta)).$$

To humans, the bias term  $b + d_2/2$  is not audible. Hence, this signal consists of a pure tone, scaled by  $a$ , and a distortion term at twice the frequency, scaled by  $d_2/2$ . This distortion term is audible as harmonic distortion as long as  $2\omega_0$  is in the human auditory range.

A cubic term will introduce **third harmonic distortion**, and higher powers will introduce higher harmonics.

The importance of harmonic distortion depends on the application. The human auditory system is very sensitive to harmonic distortion, but the human visual system much less so, for example.

### 7.1.8 Signal Conditioning<sup>1</sup>

Noise and harmonic distortion often have significant differences from the desired signal. We can exploit those differences to reduce or even eliminate the noise or distortion. The easiest way to do this is with **frequency selective filtering**. Such filtering relies on Fourier theory, which states that a signal is an additive composition of sinusoidal signals of different frequencies. While Fourier theory is beyond the scope of this text (see [Lee and Varaiya \(2003\)](#) for details), it may be useful to some readers who have some background to see how to apply that theory in the context of embedded systems. We do that in this section.

<sup>1</sup>This section may be skipped on a first reading. It requires a background in signals and systems at the level typically covered in a sophomore or junior engineering course.

**Example 7.10:** The **accelerometer** discussed in Example 7.3 is being used to measure the orientation of a slowly moving object. But instead it measures the sum of orientation and vibration. We may be able to reduce the effect of the vibration by **signal conditioning**. If we assume that the vibration  $n(t)$  has higher frequency content than the orientation  $x(t)$ , then frequency-selective filtering will reduce the effects of vibration. Specifically, vibration may be mostly rapidly changing acceleration, whereas orientation changes more slowly, and filtering can remove the rapidly varying components, leaving behind only the slowly varying components.

To understand the degree to which frequency-selective filtering helps, we need to have a model of both the desired signal  $x$  and the noise  $n$ . Reasonable models are usually statistical, and analysis of the signals requires using the techniques of random processes, estimation, and machine learning. Although such analysis is beyond the scope of this text, we can gain insight that is useful in many practical circumstances through a purely deterministic analysis.

Our approach will be to condition the signal  $x' = x + n$  by filtering it with an **LTI** system  $S$  called a **conditioning filter**. Let the output of the conditioning filter be given by

$$y = S(x') = S(x + n) = S(x) + S(n),$$

where we have used the linearity assumption on  $S$ . Let the residual error signal after filtering be defined to be

$$r = y - x = S(x) + S(n) - x. \quad (7.10)$$

This signal tells us how far off the filtered output is from the desired signal. Let  $R$  denote the **RMS** value of  $r$ , and  $X$  the RMS value of  $x$ . Then the SNR after filtering is

$$SNR_{dB} = 20 \log_{10} \left( \frac{X}{R} \right),$$

We would like to design the conditioning filter  $S$  to maximize this SNR. Since  $X$  does not depend on  $S$ , we maximize this SNR if we minimize  $R$ . That is, we choose  $S$  to minimize the RMS value of  $r$  in (7.10).

Although determination of this filter requires statistical methods beyond the scope of this text, we can draw some intuitively appealing conclusions by examining (7.10). It is easy to show that the denominator is bounded as follows,

$$R = \text{RMS}(r) \leq \text{RMS}(S(x) - x) + \text{RMS}(n) \quad (7.11)$$

where RMS is the function defined by (7.7). This suggests that we may be able to minimize  $R$  by making  $S(x)$  close to  $x$  (i.e., make  $S(x) \approx x$ ) while making  $\text{RMS}(n)$  small. That is, the filter  $S$  should do minimal damage to the desired signal  $x$  while filtering out as much as possible of the noise.

As illustrated in Example 7.3,  $x$  and  $n$  often differ in frequency content. In that example,  $x$  contains only low frequencies, and  $n$  contains only higher frequencies. Therefore, the best choice for  $S$  will be a lowpass filter.

## 7.2 Common Sensors

In this section, we describe a few sensors and show how to obtain and use reasonable models of these sensors.

### 7.2.1 Measuring Tilt and Acceleration

An **accelerometer** is a sensor that measures **proper acceleration**, which is the acceleration of an object as observed by an observer in free fall. As we explain here, gravitational force is indistinguishable from acceleration, and therefore an accelerometer measures not just acceleration, but also gravitational force. This result is a precursor to Albert Einstein's Theory of General Relativity and is known as Einstein's **equivalence principle** (Einstein, 1907).

A schematic view of an accelerometer is shown in Figure 7.3. A movable mass is attached via a spring to a fixed frame. Assume that the sensor circuitry can measure the position of the movable mass relative to the fixed frame (this can be done, for example, by measuring capacitance). When the frame accelerates in the direction of the double arrow in the figure, the acceleration results in displacement of the movable mass, and hence this acceleration can be measured.

The movable mass has a neutral position, which is its position when the spring is not deformed at all. It will occupy this neutral position if the entire assembly is in free fall,

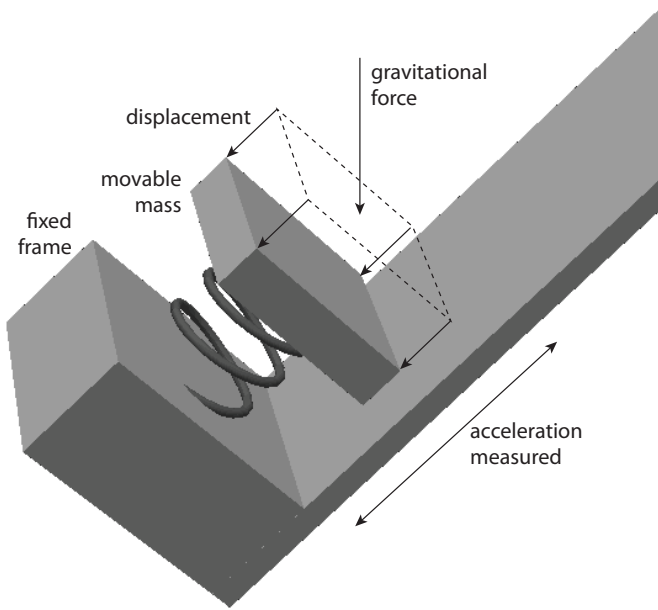


Figure 7.3: A schematic of an accelerometer as a spring-mass system.

or if the assembly is lying horizontally. If the assembly is instead aligned vertically, then gravitational force will compress the spring and displace the mass. To an observer in free fall, this looks exactly as if the assembly were accelerating upwards at the **acceleration of gravity**, which is approximately  $g = 9.8 \text{ meters/second}^2$ .

An accelerometer, therefore, can measure the tilt (relative to gravity) of the fixed frame. Any acceleration experienced by the fixed frame will add or subtract from this measurement. It can be challenging to separate these two effects, gravity and acceleration. The combination of the two is what we call **proper acceleration**.

Assume  $x$  is the proper acceleration of the fixed frame of an accelerometer at a particular time. A digital accelerometer will produce a measurement  $f(x)$  where

$$f: (L, H) \rightarrow \{0, \dots, 2^b - 1\}$$

where  $L \in \mathbb{R}$  is the minimum measurable proper acceleration and  $H \in \mathbb{R}$  is the maximum, and  $b \in \mathbb{N}$  is the number of bits of the ADC.

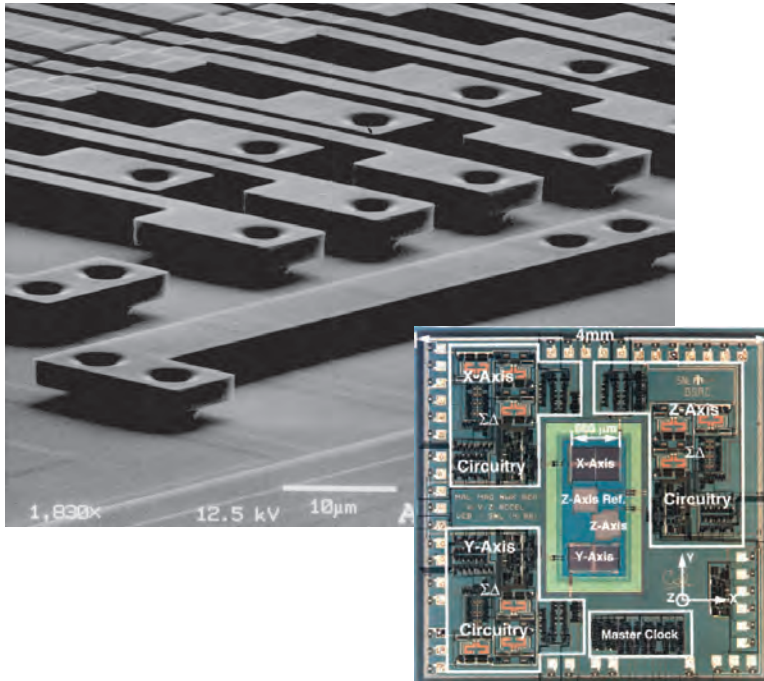


Figure 7.4: A silicon accelerometer consists of flexible silicon fingers that deform under gravitational pull or acceleration ([Lemkin and Boser, 1999](#)).

Today, accelerometers are typically implemented in silicon (see Figure 7.4), where silicon fingers deform under gravitational pull or acceleration (see for example [Lemkin and Boser \(1999\)](#)). Circuitry measures the deformation and provides a digital reading. Often, three accelerometers are packaged together, giving a three-axis accelerometer. This can be used to measure orientation of an object relative to gravity, plus acceleration in any direction in three-dimensional space.

## 7.2.2 Measuring Position and Velocity

In theory, given a measurement  $x$  of acceleration over time, it is possible to determine the velocity and location of an object. Consider an object moving in a one-dimensional space.

Let the position of the object over time be  $p: \mathbb{R}_+ \rightarrow \mathbb{R}$ , with initial position  $p(0)$ . Let the velocity of the object be  $v: \mathbb{R}_+ \rightarrow \mathbb{R}$ , with initial velocity  $v(0)$ . And let the acceleration be  $x: \mathbb{R}_+ \rightarrow \mathbb{R}$ . Then

$$p(t) = p(0) + \int_0^t v(\tau) d\tau,$$

and

$$v(t) = v(0) + \int_0^t x(\tau) d\tau.$$

Note, however, that if there is a non-zero **bias** in the measurement of acceleration, then  $p(t)$  will have an error that grows proportionally to  $t^2$ . Such an error is called **drift**, and it makes using an accelerometer alone to determine position not very useful. However, if the position can be periodically reset to a known-good value, using for example GPS, then an accelerometer becomes useful to approximate the position between such settings.

In some circumstances, we can measure velocity of an object moving through a medium. For example, an **anemometer** (which measures air flow) can estimate the velocity of an aircraft relative to the surrounding air. But using this measurement to estimate position is again subject to drift, particularly since the movement of the surrounding air ensures bias.

Direct measurements of position are difficult. The **global positioning system (GPS)** is a sophisticated satellite-based navigation system using triangulation. A GPS receiver listens for signals from four or more GPS satellites that carry extremely precise clocks. The satellites transmit a signal that includes the time of transmission and the location of the satellite at the time of transmission. If the receiver were to have an equally precise clock, then upon receiving such a signal from a satellite, it would be able to calculate its distance from the satellite using the speed of light. Given three such distances, it would be able to calculate its own position. However, such precise clocks are extremely expensive. Hence, the receiver uses a fourth such distance measurement to get a system of four equations with four unknowns, the three dimensions of its location and the error in its own local clock.

The signal from GPS satellites is relatively weak and is easily blocked by buildings and other obstacles. Other mechanisms must be used, therefore, for indoor localization. One such mechanism is **WiFi fingerprinting**, where a device uses the known location of WiFi access points, the signal strength from those access points, and other local information. Another technology that is useful for indoor localization is **bluetooth**, a short-distance wireless communication standard. Bluetooth signals can be used as beacons, and signal strength can give a rough indication of distance to the beacon.

Strength of a radio signal is a notoriously poor measure of distance because it is subject to local diffraction and reflection effects on the radio signal. In an indoor environment, a radio signal is often subject to **multipath**, where it propagates along more than one path to a target, and at the target experiences either constructive or destructive interference. Such interference introduces wide variability in signal strength that can lead to misleading measures of distance. As of this writing, mechanisms for accurate indoor localization are not widely available, in notable contrast to outdoor localization, where GPS is available worldwide.

### 7.2.3 Measuring Rotation

A **gyroscope** is a device that measures changes in orientation (rotation). Unlike an accelerometer, it is (mostly) unaffected by a gravitational field. Traditional gyroscopes are bulky rotating mechanical devices on a double gimbal mount. Modern gyroscopes are either MEMS devices (microelectromechanical systems) using small resonating structures, or optical devices that measure the difference in distance traveled by a laser beam around a closed path in opposite directions, or (for extremely high precision) devices that leverage quantum effects.

Gyroscopes and accelerometers may be combined to improve the accuracy of **inertial navigation**, where position is estimated by **dead reckoning**. Also called **ded reckoning** (for deduced reckoning), dead reckoning starts from a known initial position and orientation, and then uses measurements of motion to estimate subsequent position and orientation. An **inertial measurement unit (IMU)** or **inertial navigation system (INS)** uses a gyroscope to measure changes in orientation and an accelerometer to measure changes in velocity. Such units are subject to drift, of course, so they are often combined with GPS units, which can periodically provide “known good” location information (though not orientation). IMUs can get quite sophisticated and expensive.

### 7.2.4 Measuring Sound

A **microphone** measures changes in sound pressure. A number of techniques are used, including electromagnetic induction (where the sound pressure causes a wire to move in a magnetic field), capacitance (where the distance between a plate deformed by the sound pressure and a fixed plate varies, causing a measurable change in capacitance), or the piezoelectric effect (where charge accumulates in a crystal due to mechanical stress).

Microphones for human audio are designed to give low distortion and low noise within the human hearing frequency range, about 20 to 20,000 Hz. But microphones are also used outside this range. For example, an **ultrasonic rangefinder** emits a sound outside the human hearing range and listens for an echo. It can be used to measure the distance to a sound-reflecting surface.

### 7.2.5 Other Sensors

There are many more types of sensors. For example, measuring temperature is central to **HVAC** systems, automotive engine controllers, overcurrent protection, and many industrial chemical processes. Chemical sensors can pick out particular pollutants, measure alcohol concentration, etc. Cameras and photodiodes measure light levels and color. Clocks measure the passage of time.

A switch is a particularly simple sensor. Properly designed, it can sense pressure, tilt, or motion, for example, and it can often be directly connected to the **GPIO** pins of a microcontroller. One issue with switches, however, is that they may **bounce**. A mechanical switch that is based on closing an electrical contact has metal colliding with metal, and the establishment of the contact may not occur cleanly in one step. As a consequence, system designers need to be careful when reacting to the establishment of an electrical contact or they may inadvertently react several times to a single throwing of the switch.

## 7.3 Actuators

As with sensors, the variety of available actuators is enormous. Since we cannot provide comprehensive coverage here, we discuss two common examples, LEDs and motor control. Further details may be found in Chapter 10, which discusses particular microcontroller I/O designs.

### 7.3.1 Light-Emitting Diodes

Very few actuators can be driven directly from the digital I/O pins (**GPIO** pins) of a microcontroller. These pins can source or sink a limited amount of current, and any attempt to exceed this amount risks damaging the circuits. One exception is **light-emitting diodes (LEDs)**, which when put in series with a resistor, can often be connected directly to a



GPIO pin. This provides a convenient way for an embedded system to provide a visual indication of some activity.

**Example 7.11:** Consider a microcontroller that operates at 3 volts from a coin-cell battery and specifies that its GPIO pins can sink up to 18 mA. Suppose that you wish to turn on and off an LED under software control (see Chapter 10 for how to do this). Suppose you use an LED that, when forward biased (turned on), has a voltage drop of 2 volts. Then what is the smallest resistor you can put in series with the LED to safely keep the current within the 18 mA limit? **Ohm's law** states

$$V_R = IR, \quad (7.12)$$

where  $V_R$  is the voltage across the resistor,  $I$  is the current, and  $R$  is the resistance. The resistor will have a voltage drop of  $V_R = 3 - 2 = 1$  volt across it (two of the 3 supply volts drop across the LED), so the current flowing through it will be

$$I = 1/R.$$

To limit this current to 18 mA, we require a resistance

$$R \geq 1/0.018 \approx 56 \text{ ohms}.$$

If you choose a 100 ohm resistor, then the current flowing through the resistor and the LED is

$$I = V_R/100 = 10\text{mA}.$$

If the battery capacity is 200 mAh (milliamp-hours), then driving the LED for 20 hours will completely deplete the battery, not counting any power dissipation in the microcontroller or other circuits. The power dissipated in the resistor will be

$$P_R = V_R I = 10\text{mW}.$$

The power dissipated in the LED will be

$$P_L = 2I = 20\text{mW}.$$

These numbers give an indication of the heat generated by the LED circuit (which will be modest).

The calculations in the previous example are typical of what you need to do to connect any device to a micro controller.

#### 7.3.2 Motor Control

A **motor** applies a **torque** (angular force) to a load proportional to the current through the motor windings. It might be tempting, therefore, to apply a voltage to the motor proportional to the desired torque. However, this is rarely a good idea. First, if the voltage is digitally controlled through a **DAC**, then we have to be very careful to not exceed the current limits of the DAC. Most DACs cannot deliver much power, and require a power amplifier between the DAC and the device being powered. The input to a power amplifier has high impedance, meaning that at a given voltage, it draws very little current, so it can usually be connected directly to a DAC. The output, however, may involve substantial current.

**Example 7.12:** An audio amplifier designed to drive 8 ohm speakers can often deliver 100 watts (peak) to the speaker. Power is equal to the product of voltage and current. Combining that with Ohm's law, we get that power is proportional to the square of current,

$$P = RI^2,$$

where  $R$  is the resistance. Hence, at 100 watts, the current through an 8 ohm speaker is

$$I = \sqrt{P/R} = \sqrt{100/8} \approx 3.5\text{amps},$$

which is a substantial current. The circuitry in the power amplifier that can deliver such a current without overheating and without introducing distortion is quite sophisticated.

Power amplifiers with good linearity (where the output voltage and current are proportional to the input voltage) can be expensive, bulky, and inefficient (the amplifier itself dissipates significant energy). Fortunately, when driving a motor, we do not usually need such a power amplifier. It is sufficient to use a switch that we can turn on and off with a digital signal from a microcontroller. Making a switch that tolerates high currents is much easier than making a power amplifier.

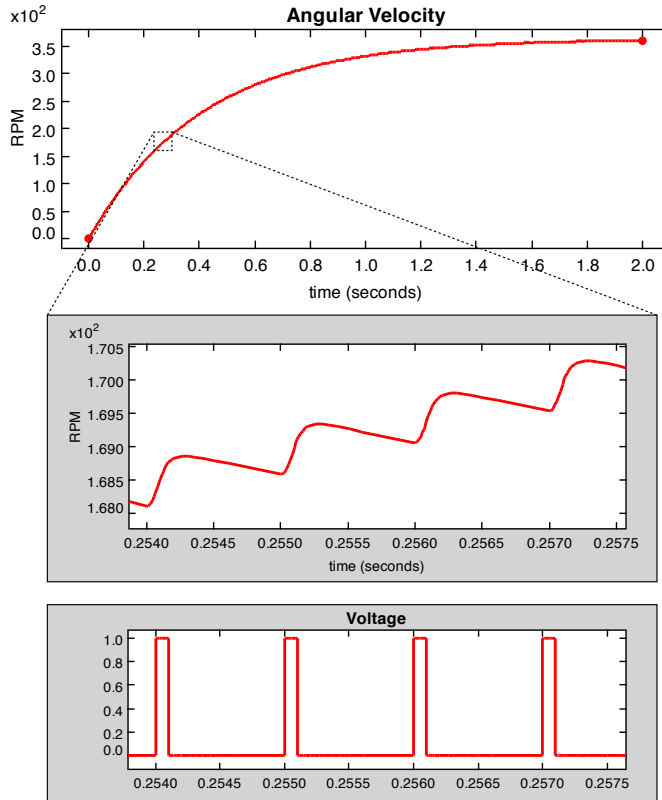


Figure 7.5: PWM control of a DC motor.

We use a technique called **pulse width modulation (PWM)**, which can efficiently deliver large amounts of power under digital control, as long as the device to which the power is being delivered can tolerate rapidly switching on and off its power source. Devices that tolerate this include LEDs, incandescent lamps (this is how dimmers work), and DC motors. A PWM signal, as shown on the bottom of Figure 7.5, switches between a high level and a low level at a specified frequency. It holds the signal high for a fraction of the cycle period. This fraction is called the **duty cycle**, and in Figure 7.5, is 0.1, or 10%.

A **DC motor** consists of an electromagnet made by winding wires around a core placed in a magnetic field made with permanent magnets or electromagnets. When current flows

through the wires, the core spins. Such a motor has both inertia and inductance that smooth its response when the current is abruptly turned on and off, so such motors tolerate PWM signals well.

Let  $\omega: \mathbb{R} \rightarrow \mathbb{R}$  represent the angular velocity of the motor as a function of time. Assume we apply a voltage  $v$  to the motor, also a function of time. Then using basic circuit theory, we expect the voltage and current through the motor to satisfy the following equation,

$$v(t) = Ri(t) + L \frac{di(t)}{dt},$$

where  $R$  is the resistance and  $L$  the inductance of the coils in the motor. That is, the coils of the motor are modeled as series connection of a resistor and an inductor. The voltage drop across the resistor is proportional to current, and the voltage drop across the inductor is proportional to the rate of change of current.

However, motors exhibit a phenomenon that when a coil rotates in a magnetic field, it *generates* a current (and corresponding voltage). In fact, a motor can also function as an electrical generator; if instead of mechanically coupling it to a passive load, you couple it to a source of power that applies a **torque** to the motor, then the motor will generate electricity. Even when the motor is being used a motor rather than a generator, there will be some torque resisting the rotation, called the **back electromagnetic force**, due to this tendency to generate electricity when rotated. To account for this, the above equation becomes

$$v(t) = Ri(t) + L \frac{di(t)}{dt} + k_b \omega(t), \quad (7.13)$$

where  $k_b$  is an empirically determined **back electromagnetic force constant**, typically expressed in units of volts/RPM (volts per revolutions per minute).

Having described the electrical behavior of the motor in (7.13), we can use the techniques of Section 2.1 to describe the mechanical behavior. We can use the rotational version of **Newton's second law**,  $F = ma$ , which replaces the force  $F$  with torque, the mass  $m$  with **moment of inertia** and the acceleration  $a$  with angular acceleration. The torque  $T$  on the motor is proportional to the current flowing through the motor, adjusted by friction and any torque that might be applied by the mechanical load,

$$T(t) = k_T i(t) - \eta \omega(t) - \tau(t),$$

where  $k_T$  is an empirically determined **motor torque constant**,  $\eta$  is the kinetic friction of the motor, and  $\tau$  is the torque applied by the load. By Newton's second law, this needs

to be equal to the moment of inertia  $I$  times the angular acceleration, so

$$I \frac{d\omega(t)}{dt} = k_T i(t) - \eta \omega(t) - \tau(t). \quad (7.14)$$

Together, (7.14) and (7.13) describe how the motor responds to an applied voltage and mechanical torque.

**Example 7.13:** Consider a particular motor with the following parameters,

$$\begin{aligned} I &= 3.88 \times 10^{-7} \text{ kg} \cdot \text{meters}^2 \\ k_b &= 2.75 \times 10^{-4} \text{ volts/RPM} \\ k_T &= 5.9 \times 10^{-3} \text{ newton} \cdot \text{meters/amp} \\ R &= 1.71 \text{ ohms} \\ L &= 1.1 \times 10^{-4} \text{ henrys} \end{aligned}$$

Assume that there is no additional load on the motor, and we apply a PWM signal with frequency 1 kHz and duty cycle 0.1. Then the response of the motor is as shown in Figure 7.5, which has been calculated by numerically simulating according to equations (7.14) and (7.13). Notice that the motor settles at a bit more than 350 RPM after 2 seconds. As shown in the detailed plot, the angular velocity of the motor jitters at a rate of 1 kHz. It accelerates rapidly when the PWM signal is high, and decelerates when it is low, the latter due to friction and back electromagnetic force. If we increase the frequency of the PWM signal, then we can reduce the magnitude of this jitter.

In a typical use of a PWM controller to drive a motor, we will use the feedback control techniques of Section 2.4 to set the speed of the motor to a desired RPM. To do this, we require a measurement of the speed of the motor. We can use a sensor called a **rotary encoder**, or just **encoder**, which reports either the angular position or velocity (or both) of a rotary shaft. There are many different designs for such encoders. A very simple one provides an electrical pulse each time the shaft rotates by a certain angle, so that counting pulses per unit time will provide a measurement of the angular velocity.

### 7.4 Summary

The variety of sensors and actuators that are available to engineers is enormous. In this chapter, we emphasize *models* of these sensors and actuators. Such models are an essential part of the toolkit of embedded systems designers. Without such models, engineers would be stuck with guesswork and experimentation.

## Exercises

1. Show that the composition  $f \circ g$  of two **affine functions**  $f$  and  $g$  is affine.
2. The dynamic range of human hearing is approximately 100 decibels. Assume that the smallest difference in sound levels that humans can effectively discern is a sound pressure of about  $20 \mu\text{Pa}$  (micropascals).
  - (a) Assuming a dynamic range of 100 decibels, what is the sound pressure of the loudest sound that humans can effectively discriminate?
  - (b) Assume a perfect microphone with a range that matches the human hearing range. What is the minimum number of bits that an **ADC** should have to match the dynamic range of human hearing?

3. The following questions are about how to determine the function

$$f: (L, H) \rightarrow \{0, \dots, 2^B - 1\},$$

for an accelerometer, which given a **proper acceleration**  $x$  yields a digital number  $f(x)$ . We will assume that  $x$  has units of “g’s,” where  $1g$  is the **acceleration of gravity**, approximately  $g = 9.8\text{meters/second}^2$ .

- (a) Let the **bias**  $b \in \{0, \dots, 2^B - 1\}$  be the output of the ADC when the accelerometer measures no **proper acceleration**. How can you measure  $b$ ?
- (b) Let  $a \in \{0, \dots, 2^B - 1\}$  be the *difference* in output of the ADC when the accelerometer measures  $0g$  and  $1g$  of acceleration. This is the ADC conversion of the **sensitivity** of the accelerometer. How can you measure  $a$ ?
- (c) Suppose you have measurements of  $a$  and  $b$  from parts (3b) and (3a). Give an **affine function** model for the accelerometer, assuming the proper acceleration is  $x$  in units of g’s. Discuss how accurate this model is.
- (d) Given a measurement  $f(x)$  (under the affine model), find  $x$ , the proper acceleration in g’s.
- (e) The process of determining  $a$  and  $b$  by measurement is called **calibration** of the sensor. Discuss why it might be useful to individually calibrate each particular accelerometer, rather than assume fixed calibration parameters  $a$  and  $b$  for a collection of accelerometers.

- (f) Suppose you have an ideal 8-bit digital accelerometer that produces the value  $f(x) = 128$  when the proper acceleration is  $0g$ , value  $f(x) = 1$  when the proper acceleration is  $3g$  to the right, and value  $f(x) = 255$  when the proper acceleration is  $3g$  to the left. Find the sensitivity  $a$  and bias  $b$ . What is the **dynamic range** (in decibels) of this accelerometer? Assume the accelerometer never yields  $f(x) = 0$ .

4. (this problem is due to Eric Kim)

You are a Rebel Alliance fighter pilot evading pursuit from the Galactic Empire by hovering your space ship beneath the clouds of the planet Cory. Let the positive  $z$  direction point upwards and be your ship's position relative to the ground and  $v$  be your vertical velocity. The gravitational force is strong with this planet and induces an acceleration (in a vacuum) with absolute value  $g$ . The force from air resistance is linear with respect to velocity and is equal to  $rv$ , where the drag coefficient  $r \leq 0$  is a constant parameter of the model. The ship has mass  $M$ . Your engines provide a vertical force.

- (a) Let  $L(t)$  be the input be the vertical lift force provided from your engines. Write down the dynamics for your ship for the position  $z(t)$  and velocity  $v(t)$ . Ignore the scenario when your ship crashes. The right hand sides should contain  $v(t)$  and  $L(t)$ .
- (b) Given your answer to the previous problem, write down the explicit solution to  $z(t)$  and  $v(t)$  when the air resistance force is negligible and  $r = 0$ . At initial time  $t = 0$ , you are  $30m$  above the ground and have an initial velocity of  $-10\frac{m}{s}$ . *Hint: Write  $v(t)$  first then write  $z(t)$  in terms of  $v(t)$ .*
- (c) Draw an actor model using integrators, adders, etc. for the system that generates your vertical position and velocity. Make sure to label all variables in your actor model.
- (d) Your engine is slightly damaged and you can only control it by giving a pure input, switch, that when present instantaneously switches the state of the engine from on to off and vice versa. When on, the engine creates a positive lift force  $L$  and when off  $L = 0$ . Your instrumentation panel contains an accelerometer. Assume your spaceship is level (i.e. zero pitch angle) and the accelerometer's positive  $z$  axis points upwards. Let the input sequence of engine switch commands be

$$\text{switch}(t) = \left\{ \begin{array}{ll} \text{present} & \text{if } t \in \{.5, 1.5, 2.5, \dots\} \\ \text{absent} & \text{otherwise} \end{array} \right\}.$$



To resolve ambiguity at switching times  $t = .5, 1.5, 2.5, \dots$ , at the moment of transition the engine's force takes on the new value instantaneously. Assume that air resistance is negligible (i.e.  $r = 0$ ), ignore a crashed state, and the engine is on at  $t = 0$ .

Sketch the vertical component of the accelerometer reading as a function of time  $t \in \mathbb{R}$ . Label important values on the axes. *Hint: Sketching the graph for force first would be helpful.*

- (e) If the spaceship is flying at a constant height, what is the value read by the accelerometer?

# Embedded Processors

<b>8.1</b>	<b>Types of Processors</b>	<b>211</b>
8.1.1	Microcontrollers	212
8.1.2	DSP Processors	212
	<i>Sidebar: Microcontrollers</i>	213
	<i>Sidebar: Programmable Logic Controllers</i>	214
	<i>Sidebar: The x86 Architecture</i>	215
	<i>Sidebar: DSP Processors</i>	216
8.1.3	Graphics Processors	220
<b>8.2</b>	<b>Parallelism</b>	<b>220</b>
8.2.1	Parallelism vs. Concurrency	220
	<i>Sidebar: Circular Buffers</i>	221
8.2.2	Pipelining	225
8.2.3	Instruction-Level Parallelism	228
8.2.4	Multicore Architectures	233
	<i>Sidebar: Fixed-Point Numbers</i>	234
	<i>Sidebar: Fixed-Point Numbers (continued)</i>	235
<b>8.3</b>	<b>Summary</b>	<b>236</b>
	<i>Sidebar: Fixed-Point Arithmetic in C</i>	237
	<b>Exercises</b>	<b>238</b>

In **general-purpose computing**, the variety of instruction set architectures today is limited, with the Intel x86 architecture overwhelmingly dominating all. There is no such dominance in embedded computing. On the contrary, the variety of processors can be daunting to a system designer. Our goal in this chapter is to give the reader the tools and

vocabulary to understand the options and to critically evaluate the properties of processors. We particularly focus on the mechanisms that provide concurrency and control over timing, because these issues loom large in the design of cyber-physical systems.

When deployed in a product, embedded processors typically have a dedicated function. They control an automotive engine or measure ice thickness in the Arctic. They are not asked to perform arbitrary functions with user-defined software. Consequently, the processors can be more specialized. Making them more specialized can bring enormous benefits. For example, they may consume far less energy, and consequently be usable with small batteries for long periods of time. Or they may include specialized hardware to perform operations that would be costly to perform on general-purpose hardware, such as image analysis.

When evaluating processors, it is important to understand the difference between an **instruction set architecture (ISA)** and a **processor realization** or a **chip**. The latter is a piece of silicon sold by a semiconductor vendor. The former is a definition of the instructions that the processor can execute and certain structural constraints (such as word size) that realizations must share. x86 is an ISA. There are many realizations. An ISA is an abstraction shared by many realizations. A single ISA may appear in many different chips, often made by different manufacturers, and often having widely varying performance profiles.

The advantage of sharing an ISA in a family of processors is that software tools, which are costly to develop, may be shared, and (sometimes) the same programs may run correctly on multiple realizations. This latter property, however, is rather treacherous, since an ISA does not normally include any constraints on timing. Hence, although a program may execute logically the same way on multiple chips, the system behavior may be radically different when the processor is embedded in a cyber-physical system.

## 8.1 Types of Processors

As a consequence of the huge variety of embedded applications, there is a huge variety of processors that are used. They range from very small, slow, inexpensive, low-power devices, to high-performance, special-purpose devices. This section gives an overview of some of the available types of processors.

### 8.1.1 Microcontrollers

A **microcontroller** ( $\mu\text{C}$ ) is a small computer on a single integrated circuit consisting of a relatively simple **central processing unit** (CPU) combined with peripheral devices such as memories, I/O devices, and timers. By some accounts, more than half of all CPUs sold worldwide are microcontrollers, although such a claim is hard to substantiate because the difference between microcontrollers and general-purpose processors is indistinct. The simplest microcontrollers operate on 8-bit words and are suitable for applications that require small amounts of memory and simple logical functions (vs. performance-intensive arithmetic functions). They may consume extremely small amounts of energy, and often include a **sleep mode** that reduces the power consumption to nanowatts. Embedded components such as sensor network nodes and surveillance devices have been demonstrated that can operate on a small battery for several years.

Microcontrollers can get quite elaborate. Distinguishing them from general-purpose processors can get difficult. The Intel Atom, for example, is a family of x86 CPUs used mainly in netbooks and other small mobile computers. Because these processors are designed to use relatively little energy without losing too much performance relative to processors used in higher-end computers, they are suitable for some embedded applications and in servers where cooling is problematic. AMD's Geode is another example of a processor near the blurry boundary between general-purpose processors and microcontrollers.

### 8.1.2 DSP Processors

Many embedded applications do quite a bit of signal processing. A signal is a collection of sampled measurements of the physical world, typically taken at a regular rate called the sample rate. A motion control application, for example, may read position or location information from sensors at sample rates ranging from a few Hertz (Hz, or samples per second) to a few hundred Hertz. Audio signals are sampled at rates ranging from 8,000 Hz (or 8 kHz, the sample rate used in telephony for voice signals) to 44.1 kHz (the sample rate of CDs). Ultrasonic applications (such as medical imaging) and high-performance music applications may sample sound signals at much higher rates. Video typically uses sample rates of 25 or 30 Hz for consumer devices to much higher rates for specialty measurement applications. Each sample, of course, contains an entire image (called a frame), which itself has many samples (called pixels) distributed in space rather than time. Software-defined radio applications have sample rates that can range from hundreds of kHz (for baseband processing) to several GHz (billions of Hertz). Other embedded applications

## Microcontrollers

Most semiconductor vendors include one or more families of microcontrollers in their product line. Some of the architectures are quite old. The **Motorola 6800** and **Intel 8080** are 8-bit microcontrollers that appeared on the market in 1974. Descendants of these architectures survive today, for example in the form of the **Freescall 6811**. The **Zilog Z80** is a fully-compatible descendant of the 8080 that became one of the most widely manufactured and widely used microcontrollers of all time. A derivative of the Z80 is the Rabbit 2000 designed by Rabbit Semiconductor.

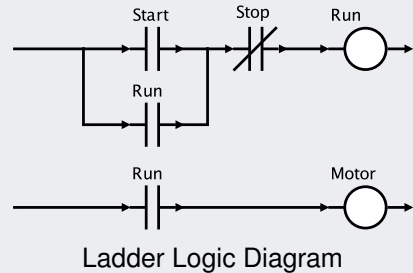
Another very popular and durable architecture is the **Intel 8051**, an 8-bit microcontroller developed by Intel in 1980. The 8051 **ISA** is today supported by many vendors, including Atmel, Infineon Technologies, Dallas Semiconductor, NXP, ST Microelectronics, Texas Instruments, and Cypress Semiconductor. The **Atmel AVR** 8-bit microcontroller, developed by Atmel in 1996, was one of the first microcontrollers to use on-chip **flash memory** for program storage. Although Atmel says AVR is not an acronym, it is believed that the architecture was conceived by two students at the Norwegian Institute of Technology, Alf-Egil Bogen and Vegard Wollan, so it may have originated as Alf and Vegard's **RISC**.

Many 32-bit microcontrollers implement some variant of an **ARM** instruction set, developed by ARM Limited. ARM originally stood for Advanced RISC Machine, and before that Acorn RISC Machine, but today it is simply ARM. Processors that implement the ARM ISA are widely used in mobile phones to realize the user interface functions, as well as in many other embedded systems. Semiconductor vendors license the instruction set from ARM Limited and produce their own chips. ARM processors are currently made by Alcatel, Atmel, Broadcom, Cirrus Logic, Freescale, LG, Marvell Technology Group, NEC, NVIDIA, NXP, Samsung, Sharp, ST Microelectronics, Texas Instruments, VLSI Technology, Yamaha, and others.

Other notable embedded microcontroller architectures include the **Motorola ColdFire** (later the Freescale ColdFire), the **Hitachi H8** and SuperH, the **MIPS** (originally developed by a team led by John Hennessy at Stanford University), the **PIC** (originally Programmable Interface Controller, from Microchip Technology), and the **PowerPC** (created in 1991 by an alliance of Apple, IBM, and Motorola).

## Programmable Logic Controllers

A **programmable logic controller (PLC)** is a specialized form of a micro-controller for industrial automation. PLCs originated as replacements for control circuits using electrical relays to control machinery. They are typically designed for continuous operation in hostile environments (high temperature, humidity, dust, etc.).



PLCs are often programmed using **ladder logic**, a notation originally used to specify logic constructed with relays and switches. A **relay** is a switch where the contact is controlled by coil. When a voltage is applied to the coil, the contact closes, enabling current to flow through the relay. By interconnecting contacts and coils, relays can be used to build digital controllers that follow specified patterns.

In common notation, a contact is represented by two vertical bars, and a coil by a circle, as shown in the diagram above. The above diagram has two **rungs**. The Motor coil on the lower rung turns a motor on or off. The Start and Stop contacts represent pushbutton switches. When an operator pushes the Start button, the contact is closed, and current can flow from the left (the power rail) to the right (ground). Start is a **normally open** contact. The Stop contact is **normally closed**, indicated by the slash, meaning that it becomes open when the operator pushes the switch. The logic in the upper rung is interesting. When the operator pushes Start, current flows to the Run coil, causing both Run contacts to close. The motor will run, even after the Start button is released. When the operator pushes Stop, current is interrupted, and both Run contacts become open, causing the motor to stop. Contacts wired in parallel perform a logical OR function, and contacts wired in series perform a logical AND. The upper rung has feedback; the meaning of the rung is a **fixed point** solution to the logic equation implied by the diagram.

Today, PLCs are just microcontrollers in rugged packages with I/O interfaces suitable for industrial control, and ladder logic is a graphical programming notation for programs. These diagrams can get quite elaborate, with thousands of rungs. For details, we recommend [Kamen \(1999\)](#).

that make heavy use of signal processing include interactive games; radar, sonar, and LIDAR (light detection and ranging) imaging systems; video analytics (the extraction of information from video, for example for surveillance); driver-assist systems for cars; medical electronics; and scientific instrumentation.

Signal processing applications all share certain characteristics. First, they deal with large amounts of data. The data may represent samples in time of a physical processor (such as samples of a wireless radio signal), samples in space (such as images), or both (such as video and radar). Second, they typically perform sophisticated mathematical operations on the data, including filtering, system identification, frequency analysis, machine learning, and feature extraction. These operations are mathematically intensive.

Processors designed specifically to support numerically intensive signal processing applications are called **DSP processors**, or **DSPs (digital signal processors)**, for short. To get some insight into the structure of such processors and the implications for the embedded software designer, it is worth understanding the structure of typical signal processing algorithms.

A canonical signal processing algorithm, used in some form in all of the above applications, is **finite impulse response (FIR)** filtering. The simplest form of this algorithm is straightforward, but has profound implications for hardware. In this simplest form, an input signal  $x$  consists of a very long sequence of numerical values, so long that for design

### The x86 Architecture

The dominant ISA for desktop and portable computers is known as the **x86**. This term originates with the Intel 8086, a 16-bit microprocessor chip designed by Intel in 1978. A variant of the 8086, designated the 8088, was used in the original IBM PC, and the processor family has dominated the PC market ever since. Subsequent processors in this family were given names ending in “86,” and generally maintained backward compatibility. The Intel 80386 was the first 32-bit version of this instruction set, introduced in 1985. Today, the term “x86” usually refers to the 32-bit version, with 64-bit versions designated “x86-64.” The **Intel Atom**, introduced in 2008, is an x86 processor with significantly reduced energy consumption. Although it is aimed primarily at netbooks and other small mobile computers, it is also an attractive option for some embedded applications. The x86 architecture has also been implemented in processors from AMD, Cyrix, and several other manufacturers.

purposes it should be considered infinite. Such an input can be modeled as a function  $x: \mathbb{N} \rightarrow D$ , where  $D$  is a set of values in some data type.<sup>1</sup> For example,  $D$  could be the set of all 16-bit integers, in which case,  $x(0)$  is the first input value (a 16-bit integer),  $x(1)$  is the second input value, etc. For mathematical convenience, we can augment this to  $x: \mathbb{Z} \rightarrow D$  by defining  $x(n) = 0$  for all  $n < 0$ . For each input value  $x(n)$ , an FIR filter must compute an output value  $y(n)$  according to the formula,

$$y(n) = \sum_{i=0}^{N-1} a_i x(n-i), \quad (8.1)$$

---

<sup>1</sup>For a review of this notation, see Appendix A on page 493.

## DSP Processors

Specialized computer architectures for signal processing have been around for quite some time (Allen, 1975). Single-chip DSP microprocessors first appeared in the early 1980s, beginning with the Western Electric DSP1 from Bell Labs, the S28211 from AMI, the TMS32010 from Texas Instruments, the uPD7720 from NEC, and a few others. Early applications of these devices included voiceband data modems, speech synthesis, consumer audio, graphics, and disk drive controllers. A comprehensive overview of DSP processor generations through the mid-1990s can be found in Lapsley et al. (1997).

Central characteristics of DSPs include a hardware multiply-accumulate unit; several variants of the Harvard architecture (to support multiple simultaneous data and program fetches); and addressing modes supporting auto increment, circular buffers, and bit-reversed addressing (the latter to support FFT calculation). Most support fixed-point data precisions of 16-24 bits, typically with much wider accumulators (40-56 bits) so that a large number of successive multiply-accumulate instructions can be executed without overflow. A few DSPs have appeared with floating point hardware, but these have not dominated the marketplace.

DSPs are difficult to program compared to RISC architectures, primarily because of complex specialized instructions, a pipeline that is exposed to the programmer, and asymmetric memory architectures. Until the late 1990s, these devices were almost always programmed in assembly language. Even today, C programs make extensive use of libraries that are hand-coded in assembly language to take advantage of the most esoteric features of the architectures.



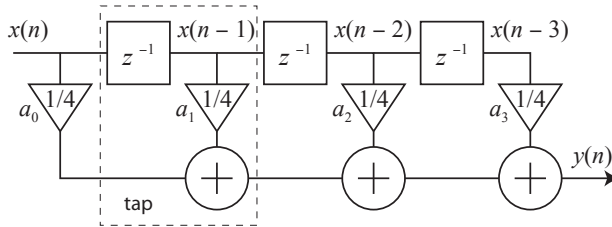


Figure 8.1: Structure of a tapped delay line implementation of the FIR filter of example 8.1. This diagram can be read as a dataflow diagram. For each  $n \in \mathbb{N}$ , each component in the diagram consumes one input value from each input path and produces one output value on each output path. The boxes labeled  $z^{-1}$  are unit delays. Their task is to produce on the output path the previous value of the input (or an initial value if there was no previous input). The triangles multiply their input by a constant, and the circles add their inputs.

where  $N$  is the length of the FIR filter, and the coefficients  $a_i$  are called its **tap values**. You can see from this formula why it is useful to augment the domain of the function  $x$ , since the computation of  $y(0)$ , for example, involves values  $x(-1)$ ,  $x(-2)$ , etc.

**Example 8.1:** Suppose  $N = 4$  and  $a_0 = a_1 = a_2 = a_3 = 1/4$ . Then for all  $n \in \mathbb{N}$ ,

$$y(n) = (x(n) + x(n-1) + x(n-2) + x(n-3))/4.$$

Each output sample is the average of the most recent four input samples. The structure of this computation is shown in Figure 8.1. In that figure, input values come in from the left and propagate down the **delay line**, which is tapped after each delay element. This structure is called a **tapped delay line**.

The rate at which the input values  $x(n)$  are provided and must be processed is called the **sample rate**. If you know the sample rate and  $N$ , you can determine the number of arithmetic operations that must be computed per second.

**Example 8.2:** Suppose that an FIR filter is provided with samples at a rate of 1 MHz (one million samples per second), and that  $N = 32$ . Then outputs must be computed at a rate of 1 MHz, and each output requires 32 multiplications and 31 additions. A processor must be capable of sustaining a computation rate of 63 million arithmetic operations per second to implement this application. Of course, to sustain the computation rate, it is necessary not only that the arithmetic hardware be fast enough, but also that the mechanisms for getting data in and out of memory and on and off chip be fast enough.

An image can be similarly modeled as a function  $x: H \times V \rightarrow D$ , where  $H \subset \mathbb{N}$  represents the horizontal index,  $V \subset \mathbb{N}$  represents the vertical index, and  $D$  is the set of all possible pixel values. A **pixel** (or picture element) is a sample representing the color and intensity of a point in an image. There are many ways to do this, but all use one or more numerical values for each pixel. The sets  $H$  and  $V$  depend on the **resolution** of the image.

**Example 8.3:** Analog television is steadily being replaced by digital formats such as **ATSC**, a set of standards developed by the Advanced Television Systems Committee. In the US, the vast majority of over-the-air **NTSC** transmissions (National Television System Committee) were replaced with ATSC on June 12, 2009. ATSC supports a number of frame rates ranging from just below 24 Hz to 60 Hz and a number of resolutions. High-definition video under the ATSC standard supports, for example, a resolution of 1080 by 1920 pixels at a frame rate of 30 Hz. Hence,  $H = \{0, \dots, 1919\}$  and  $V = \{0, \dots, 1079\}$ . This resolution is called 1080p in the industry. Professional video equipment today goes up to four times this resolution (4320 by 7680). Frame rates can also be much higher than 30 Hz. Very high frame rates are useful for capturing extremely fast phenomena in slow motion.

For a grayscale image, a typical filtering operation will construct a new image  $y$  from an original image  $x$  according to the following formula,

$$\forall i \in H, j \in V, \quad y(i, j) = \sum_{n=-N}^N \sum_{m=-M}^M a_{n,m} x(i-n, j-m), \quad (8.2)$$

where  $a_{n,m}$  are the filter coefficients. This is a two-dimensional FIR filter. Such a calculation requires defining  $x$  outside the region  $H \times V$ . There is quite an art to this (to avoid edge effects), but for our purposes here, it suffices to get a sense of the structure of the computation without being concerned for this detail.

A color image will have multiple **color channels**. These may represent luminance (how bright the pixel is) and chrominance (what the color of the pixel is), or they may represent colors that can be composed to get an arbitrary color. In the latter case, a common choice is an **RGBA** format, which has four channels representing red, green, blue, and the alpha channel, which represents transparency. For example, a value of zero for R, G, and B represents the color black. A value of zero for A represents fully transparent (invisible). Each channel also has a maximum value, say 1.0. If all four channels are at the maximum, the resulting color is a fully opaque white.

The computational load of the filtering operation in (8.2) depends on the number of channels, the number of filter coefficients (the values of  $N$  and  $M$ ), the resolution (the sizes of the sets  $H$  and  $V$ ), and the frame rate.

**Example 8.4:** Suppose that a filtering operation like (8.2) with  $N = 1$  and  $M = 1$  (minimal values for useful filters) is to be performed on a high-definition video signal as in Example 8.3. Then each pixel of the output image  $y$  requires performing 9 multiplications and 8 additions. Suppose we have a color image with three channels (say, RGB, without transparency), then this will need to be performed 3 times for each pixel. Thus, each frame of the resulting image will require  $1080 \times 1920 \times 3 \times 9 = 55,987,200$  multiplications, and a similar number of additions. At 30 frames per second, this translates into 1,679,616,000 multiplications per second, and a similar number of additions. Since this is about the simplest operation one may perform on a high-definition video signal, we can see that processor architectures handling such video signals must be quite fast indeed.

In addition to the large number of arithmetic operations, the processor has to handle the movement of data down the delay line, as shown in Figure 8.1 (see box on page 221). By providing support for delay lines and multiply-accumulate instructions, as shown in Example 8.6, DSP processors can realize one tap of an FIR filter in one cycle. In that cycle, they multiply two numbers, add the result to an accumulator, and increment or decrement two pointers using modulo arithmetic.

### 8.1.3 Graphics Processors

A **graphics processing unit (GPU)** is a specialized processor designed especially to perform the calculations required in graphics rendering. Such processors date back to the 1970s, when they were used to render text and graphics, to combine multiple graphic patterns, and to draw rectangles, triangles, circles, and arcs. Modern GPUs support 3D graphics, shading, and digital video. Dominant providers of GPUs today are Intel, NVIDIA and AMD.

Some embedded applications, particularly games, are a good match for GPUs. Moreover, GPUs have evolved towards more general programming models, and hence have started to appear in other compute-intensive applications, such as instrumentation. GPUs are typically quite power hungry, and therefore today are not a good match for energy constrained embedded applications.

## 8.2 Parallelism

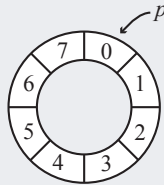
Most processors today provide various forms of parallelism. These mechanisms strongly affect the timing of the execution of a program, so embedded system designers have to understand them. This section provides an overview of the several forms and their consequences for system designers.

### 8.2.1 Parallelism vs. Concurrency

Concurrency is central to embedded systems. A computer program is said to be **concurrent** if different parts of the program *conceptually* execute simultaneously. A program is said to be **parallel** if different parts of the program *physically* execute simultaneously

## Circular Buffers

An FIR filter requires a delay-line like that shown in Figure 8.1. A naive implementation would allocate an array in memory, and each time an input sample arrives, move each element in the array to the next higher location to make room for the new element in the first location. This would be enormously wasteful of memory bandwidth. A better approach is to use a **circular buffer**, where an array in memory is interpreted as having a ring-like structure, as shown below for a length-8 delay line:



Here, 8 successive memory locations, labeled 0 to 7, store the values in the delay line. A pointer  $p$ , initialized to location 0, provides access.

An FIR filter can use this circular buffer to implement the summation of (8.1). One implementation first accepts a new input value  $x(n)$ , and then calculates the summation backwards, beginning with the  $i = N - 1$  term, where in our example,  $N = 8$ . Suppose that when the  $n^{\text{th}}$  input arrives, the value of  $p$  is some number  $p_i \in \{0, \dots, 7\}$  (for the first input  $x(0)$ ,  $p_i = 0$ ). The program writes the new input  $x(n)$  into the location given by  $p$  and then increments  $p$ , setting  $p = p_i + 1$ . All arithmetic on  $p$  is done modulo 8, so for example, if  $p_i = 7$ , then  $p_i + 1 = 0$ . The FIR filter calculation then reads  $x(n - 7)$  from location  $p = p_i + 1$  and multiplies it by  $a_7$ . The result is stored in an **accumulator** register. It again increments  $p$  by one, setting it to  $p = p_i + 2$ . It next reads  $x(n - 6)$  from location  $p = p_i + 2$ , multiplies it by  $a_6$ , and adds the result to the accumulator (this explains the name “accumulator” for the register, since it accumulates the products in the tapped delay line). It continues until it reads  $x(n)$  from location  $p = p_i + 8$ , which because of the modulo operation is the same location that the latest input  $x(n)$  was written to, and multiplies that value by  $a_0$ . It again increments  $p$ , getting  $p = p_i + 9 = p_i + 1$ . Hence, at the conclusion of this operation, the value of  $p$  is  $p_i + 1$ , which gives the location into which the next input  $x(n + 1)$  should be written.

on distinct hardware (such as on multicore machines, on servers in a server farm, or on distinct microprocessors).

Non-concurrent programs specify a *sequence* of instructions to execute. A programming language that expresses a computation as a sequence of operations is called an **imperative** language. C is an imperative language. When using C to write concurrent programs, we must step outside the language itself, typically using a **thread library**. A thread library uses facilities provided not by C, but rather provided by the operating system and/or the hardware. Java is a mostly imperative language extended with constructs that directly support threads. Thus, one can write concurrent programs in Java without stepping outside the language.

Every (correct) execution of a program in an imperative language must behave as if the instructions were executed exactly in the specified sequence. It is often possible, however, to execute instructions in parallel or in an order different from that specified by the program and still get behavior that matches what would have happened had they been executed in sequence.

**Example 8.5:** Consider the following C statements:

```
double pi, piSquared, piCubed;
pi = 3.14159;
piSquared = pi * pi ;
piCubed = pi * pi * pi;
```

The last two assignment statements are independent, and hence can be executed in parallel or in reverse order without changing the behavior of the program. Had we written them as follows, however, they would no longer be independent:

```
double pi, piSquared, piCubed;
pi = 3.14159;
piSquared = pi * pi ;
piCubed = piSquared * pi;
```

In this case, the last statement depends on the third statement in the sense that the third statement must complete execution before the last statement starts.

A compiler may analyze the dependencies between operations in a program and produce parallel code, if the target machine supports it. This analysis is called **dataflow analysis**. Many microprocessors today support parallel execution, using multi-issue instruction streams or **VLIW** (very large instruction word) architectures. Processors with multi-issue instruction streams can execute independent instructions simultaneously. The hardware analyzes instructions on-the-fly for dependencies, and when there is no dependency, executes more than one instruction at a time. In the latter, VLIW machines have assembly-level instructions that specify multiple operations to be performed together. In this case, the compiler is usually required to produce the appropriate parallel instructions. In these cases, the dependency analysis is done at the level of assembly language or at the level of individual operations, not at the level of lines of C. A line of C may specify multiple operations, or even complex operations like procedure calls. In both cases (multi-issue and VLIW), an imperative program is analyzed for concurrency in order to enable parallel execution. The overall objective is to speed up execution of the program. The goal is improved **performance**, where the presumption is that finishing a task earlier is always better than finishing it later.

In the context of embedded systems, however, concurrency plays a part that is much more central than merely improving performance. Embedded programs interact with physical processes, and in the physical world, many activities progress at the same time. An embedded program often needs to monitor and react to multiple concurrent sources of stimulus, and simultaneously control multiple output devices that affect the physical world. Embedded programs are almost always concurrent programs, and concurrency is an intrinsic part of the logic of the programs. It is not just a way to get improved performance. Indeed, finishing a task earlier is not necessarily better than finishing it later. *Timeliness* matters, of course; actions performed in the physical world often need to be done at the *right time* (neither early nor late). Picture for example an engine controller for a gasoline engine. Firing the spark plugs earlier is most certainly not better than firing them later. They must be fired at the *right time*.

Just as imperative programs can be executed sequentially or in parallel, concurrent programs can be executed sequentially or in parallel. Sequential execution of a concurrent program is done typically today by a **multitasking operating system**, which interleaves the execution of multiple tasks in a single sequential stream of instructions. Of course, the hardware may parallelize that execution if the processor has a multi-issue or VLIW architecture. Hence, a concurrent program may be converted to a sequential stream by an operating system and back to concurrent program by the hardware, where the latter translation is done to improve performance. These multiple translations greatly complicate

the problem of ensuring that things occur at the *right* time. This problem is addressed in Chapter 12.

Parallelism in the hardware, the main subject of this chapter, exists to improve performance for computation-intensive applications. From the programmer's perspective, concurrency arises as a consequence of the hardware designed to improve performance, not as a consequence of the application problem being solved. In other words, the application does not (necessarily) demand that multiple activities proceed simultaneously, it just demands that things be done very quickly. Of course, many interesting applications will combine both forms of concurrency, arising from parallelism and from application requirements.

The sorts of algorithms found in compute-intensive embedded programs has a profound affect on the design of the hardware. In this section, we focus on hardware approaches that deliver parallelism, namely pipelining, instruction-level parallelism, and multicore architectures. All have a strong influence on the programming models for embedded software. In Chapter 9, we give an overview of memory systems, which strongly influence how parallelism is handled.

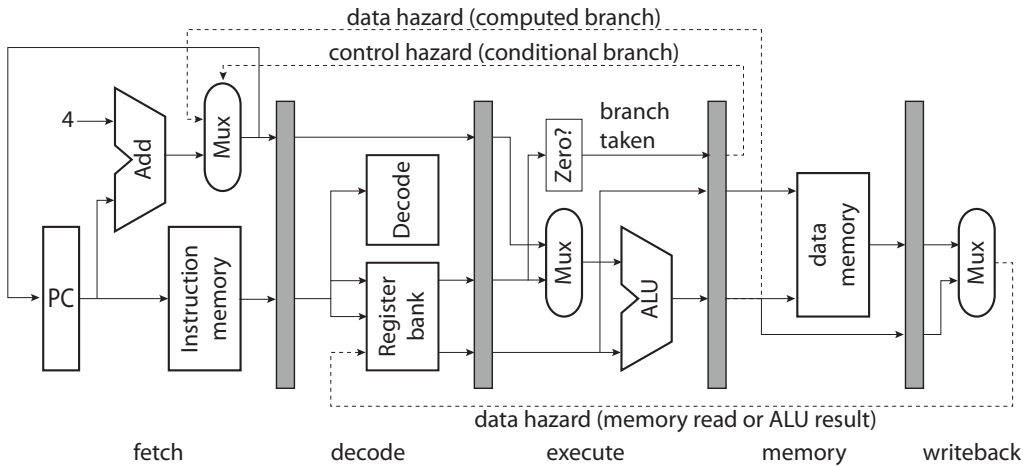


Figure 8.2: Simple pipeline (after [Patterson and Hennessy \(1996\)](#)).



### 8.2.2 Pipelining

Most modern processors are **pipelined**. A simple five-stage pipeline for a 32-bit machine is shown in Figure 8.2. In the figure, the shaded rectangles are latches, which are clocked at processor clock rate. On each edge of the clock, the value at the input is stored in the latch register. The output is then held constant until the next edge of the clock, allowing the circuits between the latches to settle. This diagram can be viewed as a **synchronous-reactive** model of the behavior of the processor.

In the fetch (leftmost) stage of the pipeline, a **program counter (PC)** provides an address to the instruction memory. The instruction memory provides encoded instructions, which in the figure are assumed to be 32 bits wide. In the fetch stage, the PC is incremented by 4 (bytes), to become the address of the next instruction, unless a conditional branch instruction is providing an entirely new address for the PC. The decode pipeline stage extracts register addresses from the 32-bit instruction and fetches the data in the specified registers from the register bank. The execute pipeline stage operates on the data fetched from the registers or on the PC (for a computed branch) using an **arithmetic logic unit (ALU)**, which performs arithmetic and logical operations. The memory pipeline stage reads or writes to a memory location given by a register. The writeback pipeline stage stores results in the register file.

**DSP** processors normally add an extra stage or two that performs a multiplication, provide separate ALUs for address calculation, and provide a dual data memory for simultaneous access to two operands (this latter design is known as a **Harvard architecture**). But the simple version without the separate ALUs suffices to illustrate the issues that an embedded system designer faces.

The portions of the pipeline between the latches operate in parallel. Hence, we can see immediately that there are simultaneously five instructions being executed, each at a different stage of execution. This is easily visualized with a **reservation table** like that in Figure 8.3. The table shows hardware resources that may be simultaneously used on the left. In this case, the register bank appears three times because the pipeline of Figure 8.2 assumes that two reads and write of the register file can occur in each cycle.

The reservation table in Figure 8.3 shows a sequence  $A, B, C, D, E$  of instructions in a program. In cycle 5,  $E$  is being fetched while  $D$  is reading from the register bank, while  $C$  is using the ALU, while  $B$  is reading from or writing to data memory, while  $A$  is writing results to the register bank. The write by  $A$  occurs in cycle 5, but the read by  $B$  occurs in cycle 3. Thus, the value that  $B$  reads will not be the value that  $A$  writes.

This phenomenon is known as a **data hazard**, one form of **pipeline hazard**. Pipeline hazards are caused by the dashed lines in Figure 8.2. Programmers normally expect that if instruction *A* is before instruction *B*, then any results computed by *A* will be available to *B*, so this behavior may not be acceptable.

Computer architects have tackled the problem of pipeline hazards in a variety of ways. The simplest technique is known as an **explicit pipeline**. In this technique, the pipeline hazard is simply documented, and the programmer (or compiler) must deal with it. For the example where *B* reads a register written by *A*, the compiler may insert three **no-op** instructions (which do nothing) between *A* and *B* to ensure that the write occurs before the read. These no-op instructions form a **pipeline bubble** that propagates down the pipeline.

A more elaborate technique is to provide **interlocks**. In this technique, the instruction decode hardware, upon encountering instruction *B* that reads a register written by *A*, will detect the hazard and delay the execution of *B* until *A* has completed the writeback stage. For this pipeline, *B* should be delayed by three clock cycles to permit *A* to complete, as shown in Figure 8.4. This can be reduced to two cycles if slightly more complex **forwarding** logic is provided, which detects that *A* is writing the same location that *B* is reading, and directly provides the data rather than requiring the write to occur before the read. Interlocks therefore provide hardware that automatically inserts pipeline bubbles.

A still more elaborate technique is **out-of-order execution**, where hardware is provided that detects a hazard, but instead of simply delaying execution of *B*, proceeds to fetch *C*, and if *C* does not read registers written by either *A* or *B*, and does not write registers read

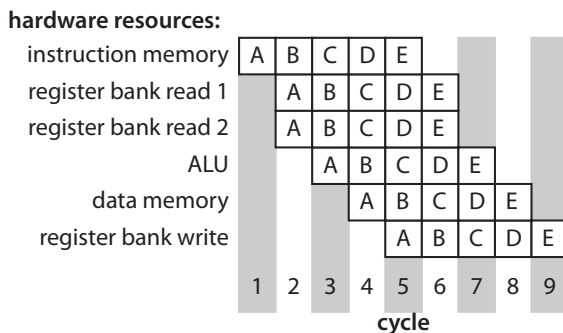


Figure 8.3: Reservation table for the pipeline shown in Figure 8.2.

by  $B$ , then proceeds to execute  $C$  before  $B$ . This further reduces the number of pipeline bubbles.

Another form of pipeline hazard illustrated in Figure 8.2 is a **control hazard**. In the figure, a conditional branch instruction changes the value of the PC if a specified register has value zero. The new value of the PC is provided (optionally) by the result of an ALU operation. In this case, if  $A$  is a conditional branch instruction, then  $A$  has to have reached the memory stage before the PC can be updated. The instructions that follow  $A$  in memory will have been fetched and will be at the decode and execute stages already by the time it is determined that those instructions should not in fact be executed.

Like data hazards, there are multiple techniques for dealing with control hazards. A **delayed branch** simply documents the fact that the branch will be taken some number of cycles after it is encountered, and leaves it up to the programmer (or compiler) to ensure that the instructions that follow the conditional branch instruction are either harmless (like no-ops) or do useful work that does not depend on whether the branch is taken. An interlock provides hardware to insert pipeline bubbles as needed, just as with data hazards. In the most elaborate technique, **speculative execution**, hardware estimates whether the branch is likely to be taken, and begins executing the instructions it expects to execute. If its expectation is not met, then it undoes any side effects (such as register writes) that the speculatively executed instructions caused.

Except for explicit pipelines and delayed branches, all of these techniques introduce variability in the timing of execution of an instruction sequence. Analysis of the timing of

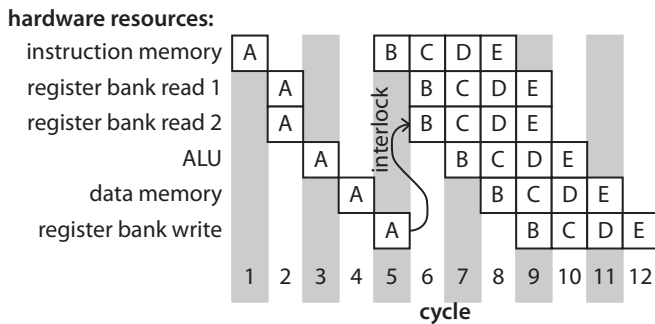


Figure 8.4: Reservation table for the pipeline shown in Figure 8.2 with interlocks, assuming that instruction  $B$  reads a register that is written by instruction  $A$ .

a program can become extremely difficult when there is a deep pipeline with elaborate forwarding and speculation. Explicit pipelines are relatively common in DSP processors, which are often applied in contexts where precise timing is essential. Out-of-order and speculative execution are common in general-purpose processors, where timing matters only in an aggregate sense. An embedded system designer needs to understand the requirements of the application and avoid processors where the requisite level of timing precision is unachievable.

### 8.2.3 Instruction-Level Parallelism

Achieving high performance demands parallelism in the hardware. Such parallelism can take two broad forms, multicore architectures, described later in Section 8.2.4, or **instruction-level parallelism (ILP)**, which is the subject of this section. A processor supporting ILP is able to perform multiple independent operations in each instruction cycle. We discuss four major forms of ILP: CISC instructions, subword parallelism, superscalar, and VLIW.

#### CISC Instructions

A processor with complex (and typically, rather specialized) instructions is called a **CISC** machine (**complex instruction set computer**). The philosophy behind such processors is distinctly different from that of **RISC** machines (**reduced instruction set computers**) (Patterson and Ditzel, 1980). DSPs are typically CISC machines, and include instructions specifically supporting **FIR** filtering (and often other algorithms such as FFTs (fast Fourier transforms) and Viterbi decoding). In fact, to qualify as a DSP, a processor must be able to perform FIR filtering in one instruction cycle per tap.

**Example 8.6:** The Texas Instruments TMS320c54x family of DSP processors is intended to be used in power-constrained embedded applications that demand high signal processing performance, such as wireless communication systems and personal digital assistants (**PDA**s). The inner loop of an FIR computation (8.1) is

```
1 RPT numberOfTaps - 1
2 MAC *AR2+, *AR3+, A
```

The first instruction illustrates the **zero-overhead loops** commonly found in DSPs. The instruction that comes after it will execute a number of times equal to one plus the argument of the RPT instruction. The MAC instruction is a **multiply-accumulate instruction**, also prevalent in DSP architectures. It has three arguments specifying the following calculation,

$$a := a + x * y ,$$

where  $a$  is the contents of an **accumulator** register named A, and  $x$  and  $y$  are values found in memory. The addresses of these values are contained by auxiliary registers AR2 and AR3. These registers are incremented automatically after the access. Moreover, these registers can be set up to implement **circular buffers**, as described in the box on page 221. The c54x processor includes a section of on-chip memory that supports two accesses in a single cycle, and as long as the addresses refer to this section of the memory, the MAC instruction will execute in a single cycle. Thus, each cycle, the processor performs two memory fetches, one multiplication, one ordinary addition, and two (possibly modulo) address increments. All DSPs have similar capabilities.

CISC instructions can get quite esoteric.

**Example 8.7:** The coefficients of the FIR filter in (8.1) are often symmetric, meaning that  $N$  is even and

$$a_i = a_{N-i-1} .$$

The reason for this is that such filters have linear phase (intuitively, this means that symmetric input signals result in symmetric output signals, or that all frequency components are delayed by the same amount). In this case, we can reduce the number of multiplications by rewriting (8.1) as

$$y(n) = \sum_{i=0}^{(N/2)-1} a_i(x(n-i) + x(n-N+i+1)) .$$

The Texas Instruments TMS320c54x instruction set includes a FIRS instruction that functions similarly to the MAC in Example 8.6, but using this calculation

rather than that of (8.1). This takes advantage of the fact that the c54x has two ALUs, and hence can do twice as many additions as multiplications. The time to execute an FIR filter now reduces to 1/2 cycle per tap.

CISC instruction sets have their disadvantages. For one, it is extremely challenging (perhaps impossible) for a compiler to make optimal use of such an instruction set. As a consequence, DSP processors are commonly used with code libraries written and optimized in assembly language.

In addition, CISC instruction sets can have subtle timing issues that can interfere with achieving [hard real-time scheduling](#). In the above examples, the layout of data in memory strongly affects execution times. Even more subtle, the use of zero-overhead loops (the RPT instruction above) can introduce some subtle problems. On the TI c54x, interrupts are disabled during repeated execution of the instruction following the RPT. This can result in unexpectedly long latencies in responding to interrupts.

### Subword Parallelism

Many embedded applications operate on data types that are considerably smaller than the word size of the processor.

**Example 8.8:** In Examples 8.3 and 8.4, the data types are typically 8-bit integers, each representing a color intensity. The color of a pixel may be represented by three bytes in the RGB format. Each of the RGB bytes has a value ranging from 0 to 255 representing the intensity of the corresponding color. It would be wasteful of resources to use, say, a 64-bit ALU to process a single 8-bit number.

To support such data types, some processors support **subword parallelism**, where a wide ALU is divided into narrower slices enabling simultaneous arithmetic or logical operations on smaller words.

**Example 8.9:** Intel introduced subword parallelism into the widely used general purpose Pentium processor and called the technology MMX (Eden and Kagan, 1997). MMX instructions divide the 64-bit datapath into slices as small as 8 bits, supporting simultaneous identical operations on multiple bytes of image pixel data. The technology has been used to enhance the performance of image manipulation applications as well as applications supporting video streaming. Similar techniques were introduced by Sun Microsystems for Sparc<sup>TM</sup> processors (Tremblay et al., 1996) and by Hewlett Packard for the PA RISC processor (Lee, 1996). Many processor architectures designed for embedded applications, including many DSP processors, also support subword parallelism.

A **vector processor** is one where the instruction set includes operations on multiple data elements simultaneously. Subword parallelism is a particular form of vector processing.

## Superscalar

**Superscalar** processors use fairly conventional sequential instruction sets, but the hardware can simultaneously dispatch multiple instructions to distinct hardware units when it detects that such simultaneous dispatch will not change the behavior of the program. That is, the execution of the program is identical to what it would have been if it had been executed in sequence. Such processors even support **out-of-order execution**, where instructions later in the stream are executed before earlier instructions. Superscalar processors have a significant disadvantage for embedded systems, which is that execution times may be extremely difficult to predict, and in the context of multitasking (interrupts and threads), may not even be repeatable. The execution times may be very sensitive to the exact timing of interrupts, in that small variations in such timing may have big effects on the execution times of programs.

## VLIW

Processors intended for embedded applications often use VLIW architectures instead of superscalar in order to get more repeatable and predictable timing. **VLIW (very large in-**

**struction word**) processors include multiple function units, like superscalar processors, but instead of dynamically determining which instructions can be executed simultaneously, each instruction specifies what each function unit should do in a particular cycle. That is, a VLIW instruction set combines multiple independent operations into a single instruction. Like superscalar architectures, these multiple operations are executed simultaneously on distinct hardware. Unlike superscalar, however, the order and simultaneity of the execution is fixed in the program rather than being decided on-the-fly. It is up to the programmer (working at assembly language level) or the compiler to ensure that the simultaneous operations are indeed independent. In exchange for this additional complexity in programming, execution times become repeatable and (often) predictable.

**Example 8.10:** In Example 8.7, we saw the specialized instruction `FIRS` of the c54x architecture that specifies operations for two ALUs and one multiplier. This can be thought of as a primitive form of VLIW, but subsequent generations of processors are much more explicit about their VLIW nature. The Texas Instruments TMS320c55x, the next generation beyond the c54x, includes two multiply-accumulate units, and can support instructions that look like this:

```
1  MAC          *AR2+, *CDP+, AC0
2  :: MAC      *AR3+, *CDP+, AC1
```

Here, `AC0` and `AC1` are two accumulator registers and `CDP` is a specialized register for pointing to filter coefficients. The notation `::` means that these two instructions should be issued and executed in the same cycle. It is up to the programmer or compiler to determine whether these instructions can in fact be executed simultaneously. Assuming the memory addresses are such that the fetches can occur simultaneously, these two `MAC` instructions execute in a single cycle, effectively dividing in half the time required to execute an FIR filter.

For applications demanding higher performance still, VLIW architectures can get quite elaborate.

**Example 8.11:** The Texas Instruments c6000 family of processors have a VLIW instruction set. Included in this family are three subfamilies of processors, the



c62x and c64x fixed-point processors and the c67x floating-point processors. These processors are designed for use in wireless infrastructure (such as cellular base stations and adaptive antennas), telecommunications infrastructure (such as voice over IP and video conferencing), and imaging applications (such as medical imaging, surveillance, machine vision or inspection, and radar).

**Example 8.12:** The **TriMedia** processor family, from NXP, is aimed at digital television, and can perform operations like that in (8.2) very efficiently. NXP Semiconductors used to be part of Philips, a diversified consumer electronics company that, among many other products, makes flat-screen TVs. The strategy in the TriMedia architecture is to make it easier for a compiler to generate efficient code, reducing the need for assembly-level programming (though it includes specialized **CISC** instructions that are difficult for a compiler to exploit). It makes things easier for the compiler by having a larger register set than is typical (128 registers), a **RISC**-like instruction set, where several instructions can be issued simultaneously, and hardware supporting **IEEE 754** floating point operations.

## 8.2.4 Multicore Architectures

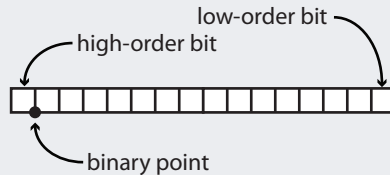
A **multicore** machine is a combination of several processors on a single chip. Although multicore machines have existed since the early 1990s, they have only recently penetrated into general-purpose computing. This penetration accounts for much of the interest in them today. **Heterogeneous multicore** machines combine a variety of processor types on a single chip, vs. multiple instances of the same processor type.

**Example 8.13:** Texas Instruments **OMAP** (open multimedia application platform) architectures are widely used in cell phones, which normally combine one or more **DSP** processors with one or more processors that are closer in style to

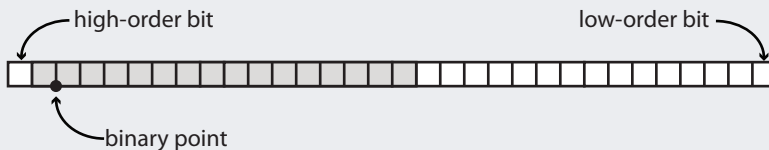
general-purpose processors. The DSP processors handle the radio, speech, and media processing (audio, images, and video). The other processors handle the user interface, database functions, networking, and downloadable applications. Specifically, the OMAP4440 includes a 1 GHz dual-core ARM Cortex processor, a c64x DSP, a GPU, and an image signal processor.

### Fixed-Point Numbers

Many embedded processors provide hardware for integer arithmetic only. Integer arithmetic, however, can be used for non-whole numbers, with some care. Given, say, a 16-bit integer, a programmer can *imagine* a **binary point**, which is like a decimal point, except that it separates bits rather than digits of the number. For example, a 16-bit integer can be used to represent numbers in the range  $-1.0$  to  $1.0$  (roughly) by placing a (conceptual) binary point just below the high-order bit of the number, as shown below:



Without the binary point, a number represented by the 16 bits is a whole number  $x \in \{-2^{15}, \dots, 2^{15} - 1\}$  (assuming the two-complement binary representation, which has become nearly universal for signed integers). With the binary point, we *interpret* the 16 bits to represent a number  $y = x/2^{15}$ . Hence,  $y$  ranges from  $-1$  to  $1 - 2^{-15}$ . This is known as a **fixed-point number**. The format of this fixed-point number can be written 1.15, indicating that there is one bit to the left of the binary point and 15 to the right. When two such numbers are multiplied at full precision, the result is a 32-bit number. The binary point is located as follows:



... Continued on page 235.

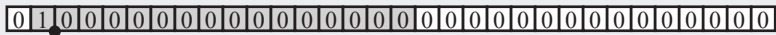
### Fixed-Point Numbers (continued)

The location of the binary point follows from the **law of conservation of bits**. When multiplying two numbers with formats  $n.m$  and  $p.q$ , the result has format  $(n+p).(m+q)$ . Processors often support such full-precision multiplications, where the result goes into an accumulator register that has at least twice as many bits as the ordinary data registers. To write the result back to a data register, however, we have to extract 16 bits from the 32 bit result. If we extract the shaded bits on page 235, then we preserve the position of the binary point, and the result still represents a number roughly in the range  $-1$  to  $1$ .

There is a loss of information, however, when we extract 16 bits from a 32-bit result. First, there is a possibility of **overflow**, because we are discarding the high-order bit. Suppose the two numbers being multiplied are both  $-1$ , which has binary representation in twos complement as follows:




When these two number are multiplied, the result has the following bit pattern:



which in twos complement, represents  $1$ , the correct result. However, when we extract the shaded 16 bits, the result is now  $-1$ ! Indeed,  $1$  is not representable in the fixed-point format  $1.15$ , so overflow has occurred. Programmers must guard against this, for example by ensuring that all numbers are strictly less than  $1$  in magnitude, prohibiting  $-1$ .

A second problem is that when we extract the shaded 16 bits from a 32-bit result, we discard 15 low-order bits. There is a loss of information here. If we simply discard the low-order 15 bits, the strategy is known as **truncation**. If instead we first add the following bit pattern the 32-bit result, then the result is known as **rounding**:



Rounding chooses the result that is closest to the full-precision result, while truncation chooses the closest result that is smaller in magnitude.

DSP processors typically perform the above extraction with either rounding or truncation in hardware when data is moved from an accumulator to a general-purpose register or to memory.

For embedded applications, multicore architectures have a significant potential advantage over single-core architectures because **real-time** and safety-critical tasks can have a dedicated processor. This is the reason for the heterogeneous architectures used for cell phones, since the radio and speech processing functions are hard real-time functions with considerable computational load. In such architectures, user applications cannot interfere with real-time functions.

This lack of interference is more problematic in general-purpose multicore architectures. It is common, for example, to use multi-level **caches**, where the second or higher level cache is shared among the cores. Unfortunately, such sharing makes it very difficult to isolate the real-time behavior of the programs on separate cores, since each program can trigger cache misses in another core. Such multi-level caches are not suitable for real-time applications.

A very different type of multicore architecture that is sometimes used in embedded applications uses one or more **soft cores** together with custom hardware on a **field-programmable gate array (FPGA)**. FPGAs are chips whose hardware function is programmable using hardware design tools. Soft cores are processors implemented on FPGAs. The advantage of soft cores is that they can be tightly coupled to custom hardware more easily than off-the-shelf processors.

## 8.3 Summary

The choice of processor architecture for an embedded system has important consequences for the programmer. Programmers may need to use assembly language to take advantage of esoteric architectural features. For applications that require precise timing, it may be difficult to control the timing of a program because of techniques in the hardware for dealing with pipeline hazards and parallel resources.

## Fixed-Point Arithmetic in C

Most C programmers will use `float` or `double` data types when performing arithmetic on non-whole numbers. However, many embedded processors lack hardware for floating-point arithmetic. Thus, C programs that use the `float` or `double` data types often result in unacceptably slow execution, since floating point must be emulated in software. Programmers are forced to use integer arithmetic to implement operations on numbers that are not whole numbers. How can they do that?

First, a programmer can *interpret* a 32-bit `int` differently from the standard representation, using the notion of a **binary point**, explained in the boxes on pages 234 and 235. However, when a C program specifies that two `ints` be multiplied, the result is an `int`, not the full precision 64-bit result that we need. In fact, the strategy outlined on page 234, of putting one bit to the left of the binary point and extracting the shaded bits from the result, will not work, because most of the shaded bits will be missing from the result. For example, suppose we want to multiply 0.5 by 0.5. This number can be represented in 32-bit `ints` as follows:

0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Without the binary point (which is invisible to C and to the hardware, residing only in the programmer's mind), this bit pattern represents the integer  $2^{30}$ , a large number indeed. When multiplying these two numbers, the result is  $2^{60}$ , which is not representable in an `int`. Typical processors will set an overflow bit in the processor status register (which the programmer must check) and deliver as a result the number 0, which is the low-order 32 bits of the product. To guard against this, a programmer can shift each 32 bit integer to the right by 16 bits before multiplying. In that case, the result of the multiply  $0.5 \times 0.5$  is the following bit pattern:

0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

With the binary point as shown, this result is interpreted as 0.25, the correct answer. Of course, shifting data to the right by 16 bits discards the 16 low-order bits in the `int`. There is a loss of precision that amounts to **truncation**. The programmer may wish to round instead, adding the `int`  $2^{15}$  to the numbers before shifting to the right 16 times. Floating-point data types make things easier. The hardware (or software) keeps track of the amount of shifting required and preserves precision when possible. However, not all embedded processors with floating-point hardware conform with the **IEEE 754** standard. This can complicate the design process for the programmer, because numerical results will not match those produced by a desktop computer.

## Exercises

1. Consider the reservation table in Figure 8.4. Suppose that the processor includes forwarding logic that is able to tell that instruction *A* is writing to the same register that instruction *B* is reading from, and that therefore the result written by *A* can be forwarded directly to the ALU before the write is done. Assume the forwarding logic itself takes no time. Give the revised reservation table. How many cycles are lost to the pipeline bubble?
2. Consider the following instruction, discussed in Example 8.6:

```
1 MAC *AR2+, *AR3+, A
```

Suppose the processor has three ALUs, one for each arithmetic operation on the addresses contained in registers AR2 and AR3 and one to perform the addition in the MAC multiply-accumulate instruction. Assume these ALUs each require one clock cycle to execute. Assume that a multiplier also requires one clock cycle to execute. Assume further that the register bank supports two reads and two writes per cycle, and that the accumulator register A can be written separately and takes no time to write. Give a reservation table showing the execution of a sequence of such instructions.

3. Assuming fixed-point numbers with format 1.15 as described in the boxes on pages 234 and 235, show that the *only* two numbers that cause overflow when multiplied are  $-1$  and  $-1$ . That is, if either number is anything other than  $-1$  in the 1.15 format, then extracting the 16 shaded bits in the boxes does not result in overflow.