

*Digital Signal Processing: A Computer Science Perspective*

Jonathan Y. Stein

Copyright © 2000 John Wiley & Sons, Inc.

Print ISBN 0-471-29546-9 Online ISBN 0-471-20059-X

## Part III

# Architectures and Algorithms

## Graphical Techniques

*Digital* signal processing means *algorithmic* processing, representing signals as streams of numbers that can be manipulated by a programmable computer. Since DSP algorithms are programmed, standard computer languages may be used in principle for their implementation. In particular, block diagrams, that are conventionally used to help one grasp the essential elements of complex conventional computer programs, may be useful as DSP description and specification tools as well.

It is difficult for people to capture and comprehend the structure of large pieces of algorithmic code, with the difficulty increasing rapidly with the length of uninterrupted code, the number of conditionals and branches, and the inherent complexity of the algorithm. In block diagrams, rectangles represent calculations the program may perform, straight lines represent possible paths between the calculations, and there are also special symbols for control structures. The proponents of block diagrams claim that by looking at a skillfully prepared block diagram the program structure becomes clear. Detractors say that these diagrams are useful only for a certain paradigm of programming that went out with the 'goto'; and that they only describe the control structures and not the data structures. Both sides agree that they are essentially a second language (in addition to the language in which the program is coded) to describe the same functionality, and as such the task of keeping them up to date and accurate is arduous.

In computer science the use of block diagrams was once pervasive but has gone out of style. In DSP *flow graphs*, which are similar to block diagrams, are still very popular. This is not because DSP is old-fashioned or less developed than computer science. This is not because DSP lacks other formalisms and tools to describe signals and systems. It is simply because the block diagram is a much more useful tool in DSP than it ever was in programming. DSP flow graphs graphically depict a DSP system's signal structure; rectangles and circles represent systems and directed lines represent signals. We thus capture the dual nature of systems and signals in one graphic portrait.

In addition, many common DSP tasks are highly structured in time and/or frequency; this results in DSP block diagrams that have easily perceived geometric structure. Finally, an algebra of graphic transformations can be developed that allow one to simplify DSP block diagrams. Such transformations often result in reductions in computational complexity that would be hard to derive without the graphical manipulations.

In this chapter we will consider DSP graphical techniques. The word *graphical* is not used here as in ‘computer graphics’ (although we advocate the use of graphical displays for depicting DSP graphs), rather as in *graph theory*. The term *graph* refers to a collection of points and lines between these points. We start with a historical introduction to graph theory. Thereafter we learn about DSP flow graphs and how to manipulate them. RAX is a visual programming block diagram system. We describe the operation and internals of RAX in order to acquaint the reader with this important class of DSP tools.

## 12.1 Graph Theory

Graphic representations have doubtless been used in science and technology for as long as humankind has pursued these subjects. The earliest uses were probably simple geometric constructions; it is easy to envision chief engineers in primitive civilizations making rough drawings before embarking on major projects; we can imagine sages in ancient civilizations studying figures and charts and then surprising kings with their predictions. We know that thousands of years ago diagrams were used for engineering and education. What the ancients grasped was that one can capture the essential elements of a complex problem using simple graphical representations.

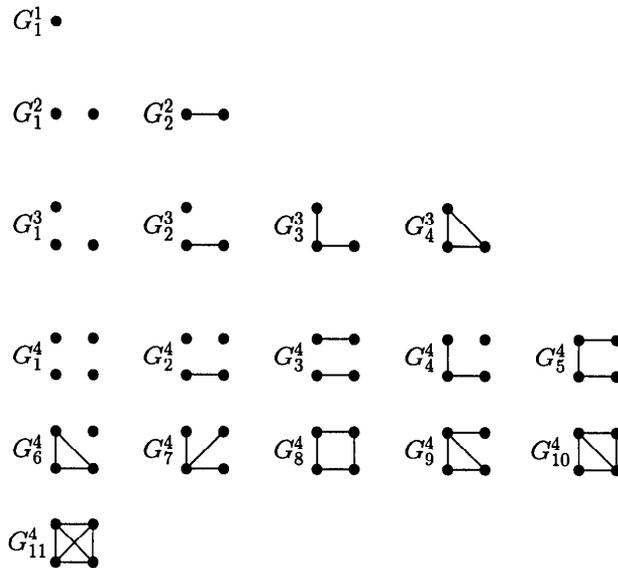
A diagram obviously does not capture all the features of the original. A map of a city is not of the original size nor does it reveal the wonders of architecture, the smells of the restaurants, the sounds of honking horns, etc. Still the map is extremely useful when navigating around town, even if it omits which streets are one-way and which tend to have traffic jams. Maps of the entire world are even more abstract representations since the world is spherical and the map is flat. Yet maps can be designed to correctly portray distances between cities, or bearing from one spot to another (but not both). As long as one realizes that a diagram can only capture certain elements of the original, and selects a diagrammatic method that captures the elements needed to solve the problem at hand, diagrams can be helpful.

In Euclidean geometry we consider two triangles to be equivalent if one could slide one on top of the other and they would coincide. The color or line width of such triangles is not taken into account, and neither is their orientation or position on the page. The transformations that are considered unimportant include arbitrary translations and rotations. When two triangles are related by such a transformation they are considered to be the same triangle. Much of high school geometry deals with methods to show two triangles are equivalent in this sense. A simple extension would be to allow transformations that include a change of scale. This would make a triangle on a map equivalent to the triangle on the ground. In this type of geometry any two triangles are considered equivalent if all of their angles are the same. In affine geometry even more general transformations are allowed, namely those which scale the  $x$  axis and  $y$  axis differently. In affine geometry all triangles are the same, but they are different from all the rectangles (which are all equivalent to each other).

Topology is even more general than affine geometry. It allows completely arbitrary transformations as long as they do not rip apart the plane or glue it different points together. You can think of this as drawing the figure on a sheet of rubber and stretching it however you want—as long as it doesn't rip or stick to another part of itself. In topology a triangle is equivalent to a rectangle or a circle, but different from a figure-eight. Graph theory is the study of points and the lines between them in topological space. In graph theory almost all the original geometry is thrown away, and we are left with a single abstraction, the *graph*.

The word *graph* as used in graph theory means a collection of points and lines that connect these points. In the mathematical terminology the points are called *vertices* and the lines *edges*; in computer science the designations *nodes* and *arcs* are more common. We shall require arcs to connect distinct nodes (no arc loops back to the same node) and rule out multiple arcs between identical nodes. The distances between nodes, the lengths or thicknesses of the arcs, and the geometric orientations are meaningless in graph theory. All that counts is which nodes are connected to which.

In Figure 12.1 we see all possible types of graphs with up to four nodes. Two nodes are said to be 'adjacent' if they are connected by an arc. A 'path' is a disjoint collection of arcs that leads from one node to another. For example, in  $G_4^4$  there is a path of length 2 from the top-left node to the bottom-right, but no path to the top-right node. A 'cycle' is a path that leads from a node to itself. In  $G_6^4$  there is a cycle, but not in  $G_7^4$ . The number of arcs emanating from a given node is called its *degree*; there are always an even number of nodes of odd degree.



**Figure 12.1:** All graphs with up to four nodes. We only allow arcs connecting distinct nodes and disallow self connections.

Many of the most interesting problems in graph theory involve the number of graphs of a certain kind. A graph in which there is a path from any node to every other node (e.g.,  $G_1^1, G_2^2, G_3^3, G_4^4, G_7^4, G_8^4, G_9^4, G_{10}^4, G_{11}^4$ ) is called 'connected', while one that has all nodes connected to all others (e.g.,  $G_1^1, G_2^2, G_4^4, G_{11}^4$ ) is called complete.

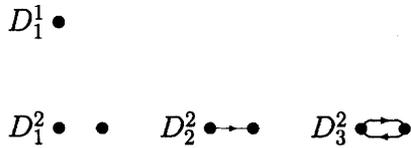
The beginnings of graph theory are usually traced back to 1736. In that year the famous mathematician, Leonhard Euler, considered the father of analysis, published his solution to a puzzle that he had been working on. Euler, who was born in Switzerland, was professor of mathematics at the academy of St. Petersburg (founded in 1725 by Catherine, the wife of Peter the Great). The cold weather so adversely affected his eyesight that in 1736 we find him living in the capital of East Prussia, Königsberg (German for 'the Kings city'). This city, founded in 1255 by Teutonic knights, was the seat of the dukes of Prussia from 1525 through 1618. After World War II the city was annexed to the USSR and renamed Kaliningrad (Russian for 'Kalinin's city') after the Soviet leader M.I. Kalinin. Today it is the capital of the Kaliningrad Oblast and is Russia's sole port that does not freeze-over in winter.

Königsberg's topography is even more interesting than its history. The Pregel river (Pregolya in Russian) flows through the city from east to west, on its way to the Frisches Haff (German for 'freshwater bay', called Wislany Zalew in Polish, and Vistula in Lithuanian), a lagoon of the Baltic Sea. Not only does the river divide the city in two, but the river itself splits into northern and southern branches that later reconverge, forming an island in the center of town. The island is connected to the other parts of town by seven bridges. The question that puzzled Euler was this. Is it possible to leave your home for a walk, cross all the bridges exactly once, and return home? In terms of graph theory the question is, 'Is it possible to start at some node, traverse all the arcs exactly once, returning to the initial node?' If possible, such a path is called an *Euler cycle*. Euler recognized the puzzle as being a specific example of a general question—does a given graph have an Euler cycle or not? Today we call such graphs *Eulerian*.

It is obvious that the distance between the nodes of the graph do not affect the answer to this question; this is truly a topological problem. What Euler discovered is that *degree is important*. A connected graph is 'Eulerian' if and only if every node has even degree. Only Eulerian graphs have paths of the desired type. With this insight Euler simultaneously founded the disciplines of graph theory and topology.

We pick up our story once again 120 years later in Great Britain. Sir William Rowan Hamilton, well known for his extensive contributions to physics (the Hamiltonian function, the Hamilton-Jacobi equation) and mathematics (complex numbers, vector algebra, group theory, 'quaternions'), was also studying a puzzle. This puzzle involved the nodes of the regular dodecahedron (a solid with twelve regular pentagonal faces). Unlike Euler's problem where each *arc* of a graph must be traversed exactly once, in Hamilton's puzzle one is required to visit each *node* exactly once. A graph is 'Hamiltonian' if and only if it contains a Hamiltonian cycle, that is, a cycle that contains each node exactly once. What Hamilton discovered is that the dodecahedron is Hamiltonian, that is, there is a path from node to node that returns to the starting point after visiting each node exactly once. Hamilton considered finding this path so challenging that he attempted to market the puzzle. The determination of necessary and sufficient conditions for a graph to be Hamiltonian turned out to be even more challenging; it remains one of the major unsolved problems in graph theory.

'Directed graphs' or 'digraphs' are just like graphs, only the arcs have an associated direction (which we depict with an arrow). In Figure 12.2 we see all possible types of digraphs with one or two nodes. Nodes in digraphs have in-degree and out-degree (also called fan-in and fan-out), and two nodes may



**Figure 12.2:** All digraphs with up to two nodes.

be connected by two directed arcs in opposite directions. Paths and cycles must traverse arcs in the direction of the arrows.

A digraph is a ‘forest’ if it does not contain a cycle. For forests the direction of the arrow dictates a certain priority, so rather than saying that connecting nodes are adjacent, we speak of parent and child nodes. The pre-arc node is the parent while the post-arc one is the child. All nodes that can be reached from a given node are called descendants and the original node the ancestor. Since forests do not have cycles we never have to worry about a node being its own ancestor. A forest that has a single ancestor node (the ‘root’) from which all other nodes descend is called a tree. Actually it is easier to think of the tree as being the basic graph and the forest as being a collection of trees.

Digraphs are the basis of a computational model used extensively in DSP called the flow graph (or ‘flow diagram’, ‘dataflow network’, ‘DSP block diagram’, ‘graphical flow programming’, ‘visual programming language’, etc.). The directed arcs of the digraph represent signals while the nodes stand for processing subsystems. If the digraph is a forest we say that the system is a *feedforward system*, while digraphs with cycles are called *feedback systems*. The study of flow graphs will be the subject of the next section.

## EXERCISES

- 12.1.1 Special types of graphs are used in electronics (schematic diagrams), physics (Feynman diagrams), computer science (search trees), and many other fields. Research and explain at least three such uses.
- 12.1.2 Why isn’t  $\bowtie$  on the list of graphs with four nodes? What about  $\boxtimes$ ?
- 12.1.3 How many different kinds of graphs are there for five nodes?
- 12.1.4 Draw all digraphs for 3 nodes.
- 12.1.5 Explain Euler’s rule intuitively.
- 12.1.6 A graph is called nonplanar if it cannot be drawn on a piece of paper without arcs crossing each other. Draw a nonplanar graph.

- 12.1.7 An Euler path is similar to an Euler cycle except that one needn't return to the same node. Similarly we can define a Hamiltonian path. Draw Euler and Hamilton path for points on a two-dimensional grid. Which paths are cycles? Find Euler paths that are not Hamilton paths and vice versa.
- 12.1.8 A *trellis* is a digraph created by mapping possible *transitions* between  $N$  states as a function of time. Conventionally the time axis is from left to right and the  $N$  states are drawn vertically. There is an arc between each state at time  $t$  and a several possible states at time  $t + 1$ . Assume a trellis with four states (called 0, 1, 2 and 3) with states 0 and 1 at time  $n$  being able to transition to even states at time  $n + 1$ , while 2 and 3 can only transition to odd states. Draw this trellis from time  $n = 0$  through 4. How many different trellises of length  $L$  are there? How may a trellis be stored in a file? What data structure may be used in a program?

## 12.2 DSP Flow Graphs

Superficially DSP flow graphs look similar to the block diagrams used to describe algorithms in computer science. In computer science the arcs indicate control paths, and computation is performed or decisions taken at nodes. Depending on decisions taken the processing will continue down different arcs to different computational nodes. Thus by the use of this graphical technique we can capture the control structure of computer programs. Other aspects of the program, such as data types and memory requirements, are not captured in these diagrams, and must be documented in some other way.

The metaphor behind the use of DSP flow graphs is that of signals 'flowing' between processing subsystems. At each of the nodes input signals are processed to produce output signals that are passed along arcs to the following nodes. Thus DSP flow graphs capture both the signal and processing system aspects of a problem. Since the vertices contain processing elements that we must identify, we will have to enrich the graphic notation of the previous section.

Let's see how to make DSP flow graphs. When a signal  $x$  is the input to some system we will depict this



and similarly we depict  $y$  as the output of a system in the following way.

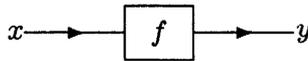


Thus the identity system, which leaves a signal  $x$  unchanged, is depicted

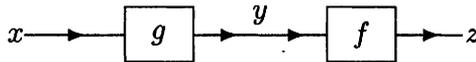


which means precisely  $y = x$ . A *hidden* signal, that is, a signal that is neither a system input nor output, can be named by placing a symbol near the corresponding arc. All the above are standard digraph figures.

As we mentioned before, nodes correspond to processing, which must be identified. We do this by drawing circles (for simple common processes) or squares (for more general processes). For example,  $y = f(x)$  is depicted



and  $z = g(y) = g(f(x))$  is shown



where the hidden signal  $y$  has been identified.

A very common operation is to multiply a signal by a real number. The standard digraph would have a multiplication node perform this function, i.e., we would expect  $y = Gx$  to be depicted something like

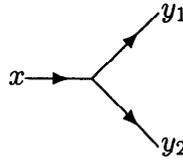


but since the operation is so common, we introduce a short-hand notation.



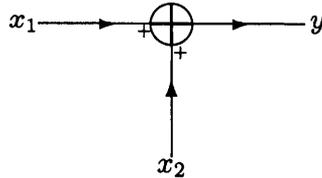
Whenever a symbol appears near an arrow we understand an implicit multiplication node  $y = Gx$ . Do not confuse this with the symbol representing a hidden signal that is placed close to an arc but not near an arrow. In such cases no multiplication is intended, and eliminating this symbol would not change the system's operation at all.

Many times we wish to have the signal  $x$  reach more than one processing system. In regular digraphs each arc connects a single pre-arc node with a unique post-arc node; in DSP we allow the connection of a single pre-arc node to multiple post-arc nodes with a single arc. The point where the signal splits into two is called a *branch point* or a *tee connector*.

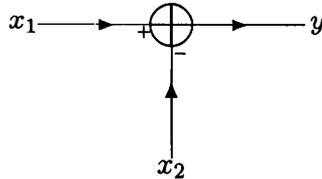


This means that the same signal is delivered to both nodes,  $y_1 = y_2 = x$ .

Of course it is meaningless to connect more than one pre-arc node together; but we *can* add two signals  $x_1$  and  $x_2$  to get  $y = x_1 + x_2$ . This is depicted in the standard notation using an addition node.

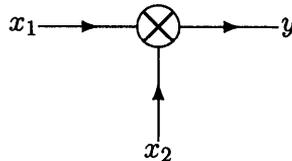


The small + signs mean that addition is to be performed. Subtraction  $y = x_1 - x_2$  is depicted

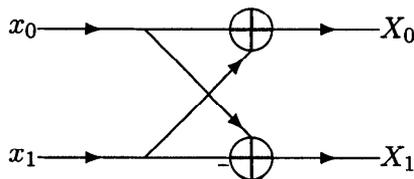


and other combinations of signs are possible.

Value-by-value multiplication  $y = x_1 x_2$  is depicted as you would expect.

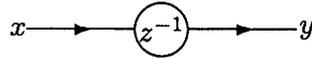


We can combine these basic elements in many ways. For example, the basic  $N = 2$  DFT of equation (4.33) is depicted



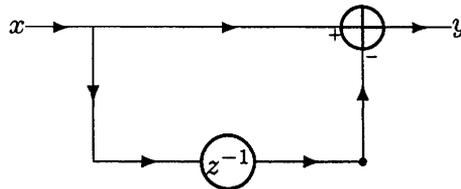
and signifies  $X_0 = x_0 + x_1$  and  $X_1 = x_0 - x_1$ .

One of the most important processing nodes is the unit delay, which is depicted by  $z^{-1}$  inscribed in a circle.

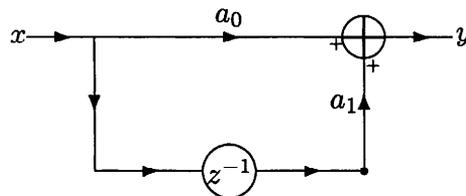


This diagram means that the *signal*  $y$  is the same as the *signal*  $x$  delayed one digital unit of time, that is,  $y_n = x_{n-1}$  for all  $n$ . Often we loosely think of the signal value at some time  $n$  as entering the delay, and its previous value exiting. Since we only represent time-invariant systems with flow graphs this interpretation is acceptable, as long as it is remembered that this same operation is performed for every unit of time. Also note that we shall never see a  $z$  in a signal flow diagram. We represent only causal, realizable systems.

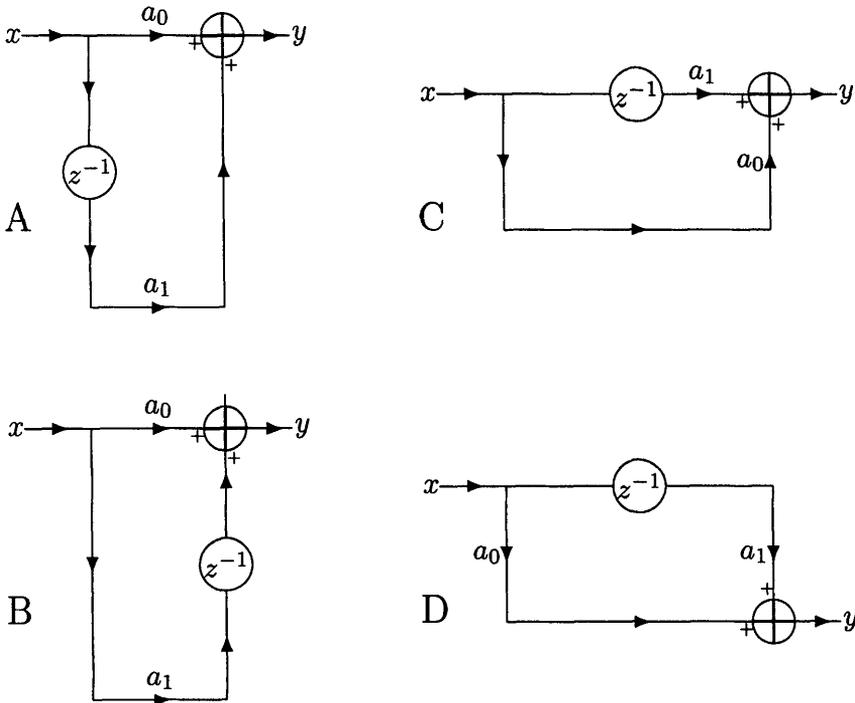
Using the unit delay we can easily represent the simple difference approximation to the derivative  $y_n = \Delta x_n = x_n - x_{n-1}$



or a general single delay convolution  $y_n = a_0 x_n + a_1 x_{n-1}$ .



You will notice that we have drawn a small filled circle in each of these diagrams. This circle does not represent a processing system, rather it is a reminder that a memory location must be set aside to store a signal value. In order to continuously calculate the simple difference or single delay convolution, we must store  $x_n$  at every time  $n$  so that at the next time step it will be available as  $x_{n-1}$ . We do not usually explicitly mark these memory locations, only stressing them when the additional emphasis is desired.



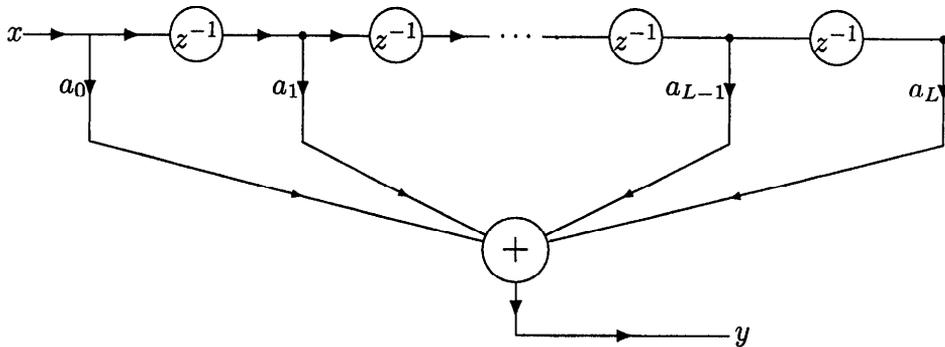
**Figure 12.3:** Four other ways of drawing a basic MA (FIR) block as a DSP flow graph.

The beauty of using graphs is that we can redraw them any way we consider æsthetic or useful, as long as the topology stays the same. Thus, the basic single delay FIR block also can be drawn in the ways depicted in Figure 12.3 and in many other ways.

All the different DSP graphs that depict the same DSP process are called *implementations* of this process. Note that the implementation of Figure 12.3.A is topologically identical to the previous graph, but has the gains appearing more symmetrically. In Figure 12.3.B we interchanged the order of the gain and the delay. Thus this implementation is not identical from the pure graph-theoretic point of view, but is an identical DSP process since the gain and delay operators commute. Figure 12.3.C looks different but one can easily convince oneself that it too represents the same block. Figure 12.3.D is similar to Figure 12.3.C but with the gains positioned symmetrically.

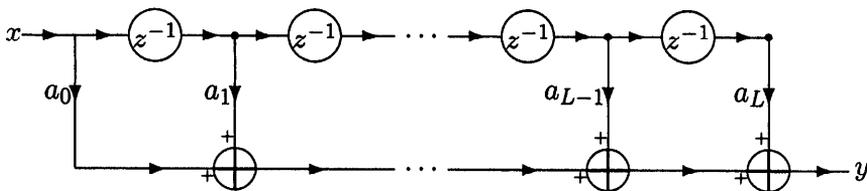
How can we implement the more general convolution?

$$y_n = \sum_{l=0}^L a_l x_{n-l}$$



**Figure 12.4:** A straightforward implementation of the FIR filter using an adder of in-degree  $L + 1$ .

In Figure 12.4 we see a straightforward implementation, where the large node at the bottom is an adder of in-degree  $L + 1$ . Such an adder is not always available, and is not really required, since we can also implement the FIR filter using the standard two-input adder, as in Figure 12.5.

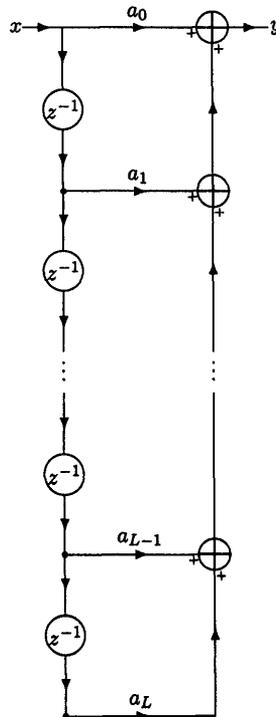


**Figure 12.5:** A straightforward implementation of the FIR filter using standard two-input adders.

This figure is worth studying. To assure yourself that you understand it completely, mark all the vertical arcs, especially those marked with filled circles. For example, the arc that descends after the first delay and then is summed with  $a_0 x_n$  splits should be marked  $x_{n-1}$ . By this we mean the signal that for all  $n$  is equal to the incoming signal  $x$  delayed by one time unit. When you consider a complex DSP graph of this type it is worthwhile interpreting it in two stages. First think of analog signals flowing through the graph. In order to assist in this interpretation assume that every processing node corresponds to a separate hardware component. Ignore any filled circles and treat  $z^{-1}$  nodes as time delays that happen to correspond to the sampling interval  $t_s$ .

Once you understand the graph at this level you can return to the world of DSP programming. The delays are now single sample delays, the processing nodes are computations that may all be carried out by the same processor, and the filled circles are memory locations. When thinking in this mode we often think of ‘typical values’, such as  $x_n$  and  $x_{n-1}$ , rather than entire signals such as  $x$ . We implicitly allow the same computation to be carried out over and over by a single processor. The basic computation to be performed repeatedly consists of multiplication of a delayed input value by a filter coefficient and adding; this combination is called a **Multiply-And-Accumulate (MAC)** operation.

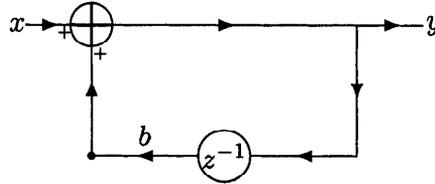
Looking closely at Figure 12.5 we see that this FIR implementation is based on the block from Figure 12.3.D. Several of these blocks are concatenated in order to form the entire convolution. This is a widely used technique—after perfecting an implementation we replicate it and use it again and again. You can think of an implementation as being similar in this regard to a subroutine in standard procedural programming languages.



**Figure 12.6:** An alternative way to depict the all-zero (MA) filter.

Of course this is not the only way to draw an FIR filter. A particularly interesting way to depict the same graph is shown in Figure 12.6. In this implementation we replicate the FIR block of Figure 12.3.A. It is easy to see that this graph is topologically identical to the previous one.

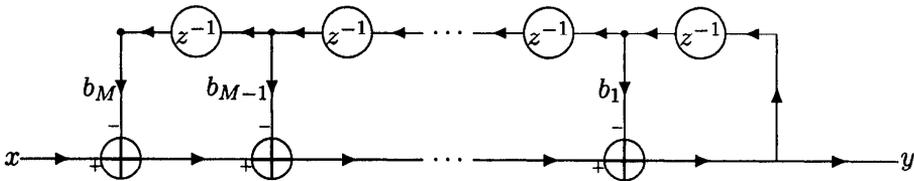
Up to now we have only seen graphs without cycles, graphs that correspond to feedforward systems. The simple feedback system,  $y_n = by_{n-1} + x_n$ , is depicted as



while a full all-pole system

$$y_n = x_n - \sum_{m=1}^M b_m y_{n-m}$$

can be depicted as in Figure 12.7.

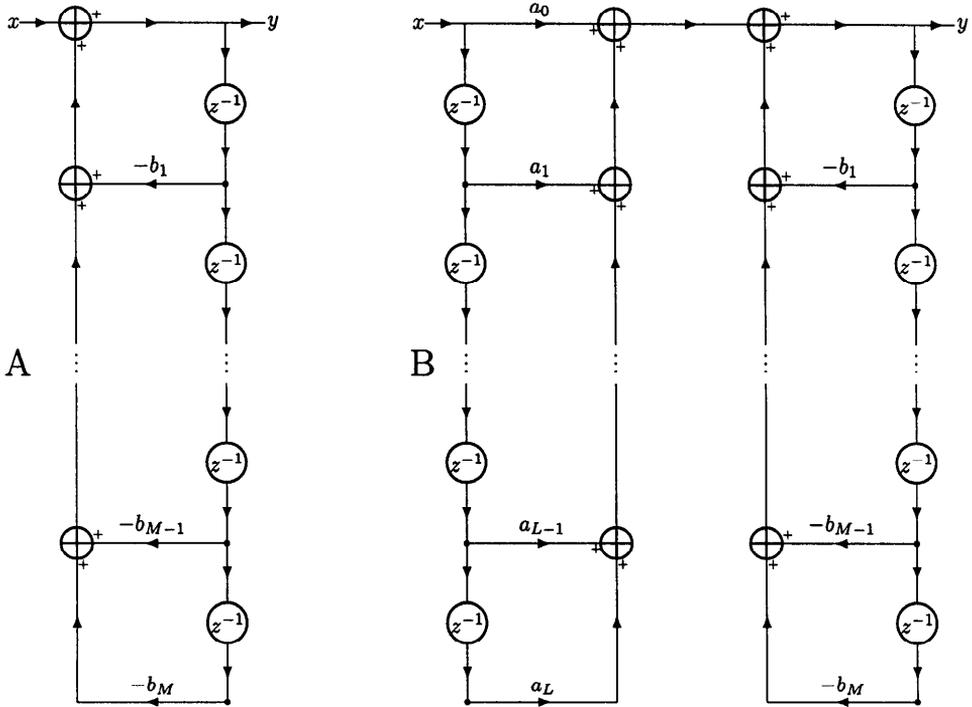


**Figure 12.7:** A full all-pole filter implemented using MAC operations.

Once again it is worthwhile to carefully mark all the arcs to be sure that you understand how this implementation works. Don't be concerned that signal values are transported *backward* in time and then influence their own values like characters in science fiction time-travel stories. This is precisely the purpose of using feedback (remember Section 7.4).

Of course this is not the only way to draw this AR filter. A particularly interesting implementation is depicted in Figure 12.8.A. We purposely made this implementation a mirror reflection of the FIR implementation of Figure 12.6. Now by concatenating the MA and AR portions we can at last implement

$$y_n = \sum_{l=0}^L a_l x_{n-l} - \sum_{m=1}^M b_m y_{n-m}$$



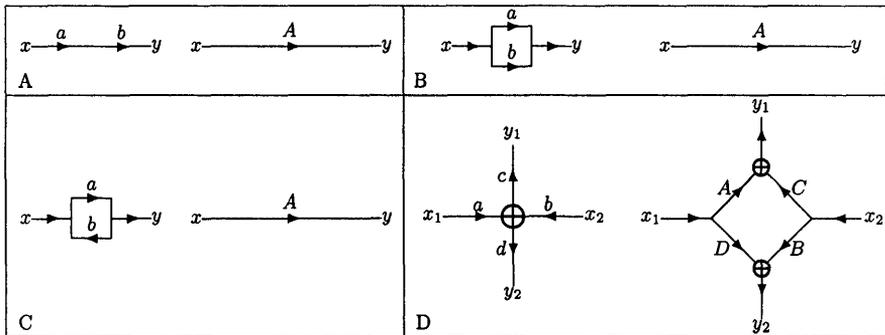
**Figure 12.8:** In (A) we present an alternative graphical representation of the all-pole (AR) filter. (B) is an implementation of the full ARMA filter.

the full ARMA filter. We do this in Figure 12.8.B.

Take a few moments to appreciate this diagram. First, by its very construction, it graphically demonstrates how ARMA filters can be decomposed into separate MA and AR subsystems. These FIR and all-pole systems are seen to be quite different in character. Of course the order of the AR and MA subsystems are not necessarily equal; they only look the same here since an ellipsis can hide different heights. Second, it's pleasingly symmetric; there *are* computational commonalities between the subsystems. Thus computational hardware or software designed for one may be modified to compute the other as well. Finally, this diagram gives us a novel way of understanding filters. Looking at the analog level we cannot avoid imagining the signal flowing in on the left, traveling down, splitting up, and recombining with differently weighted and delayed versions of itself. It then enters the feedback portion where it loops around endlessly, each time delayed, weighted, and combined with itself, until it finally exits at the right.

## EXERCISES

- 12.2.1 The following examples demonstrate simplification of diagrams with gains. In all cases identify the gain(s) that appear in the right diagram in terms of those that appear in the left diagram.

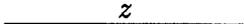
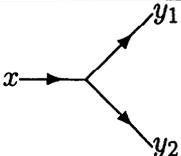
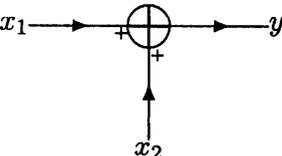
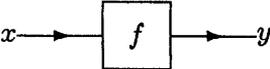


- 12.2.2 Draw an ARMA filter with MA order 3 and AR order 4. How many memory locations are required? Label all the signals.
- 12.2.3 A filter implementation with minimal number of memory allocations is called *canonical*. What is the number of memory locations in the canonical ARMA filter with MA order  $p$  and AR order  $q$ ?
- 12.2.4 The *transposition theorem* states that reversing all the arc directions, changing adders to tee connections and vice-versa, and interchanging the input and output, does not alter the system's transfer function. Prove this theorem for the simple case of  $y_n = x_n + by_{n-1}$ .

## 12.3 DSP Graph Manipulation

Let's summarize all that we have learned so far. Flow graphs are used to represent realizable, time-invariant signal processing systems. The most important graphic elements are depicted in Figure 12.9. Combining these basic elements in various ways we can depict many different systems. Every DSP flow graph corresponds to a unique signal processing system, but every system can be implemented by many seemingly different graphs.

Different implementations may have somewhat different characteristics, and may correspond to hardware implementations of differing cost and software implementations of varying complexity. It is thus useful to learn ways of manipulating flow graphs, that is, to change the graph without changing

	$x$ is the input to a calculation
	$y$ is the output from a calculation
	$y = x$
	the hidden signal $z$
	a memory location
	splitting a signal $y_1 = x$ and $y_2 = x$
	a gain $y = Gx$
	adding signals $y = x_1 + x_2$
	an arbitrary system $y = f(x)$

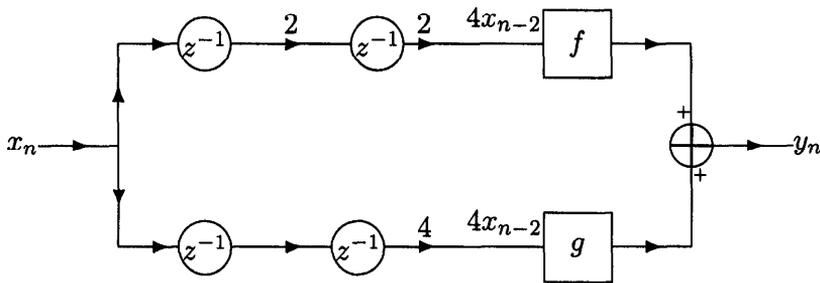
**Figure 12.9:** The most important DSP graph elements.

the system implemented. Remember that a system is defined only by the outputs generated for all inputs. As long as these remain unchanged the system is unchanged, no matter what the flow diagram looks like. We will often call graph operations that leave the system unchanged 'symmetries'.

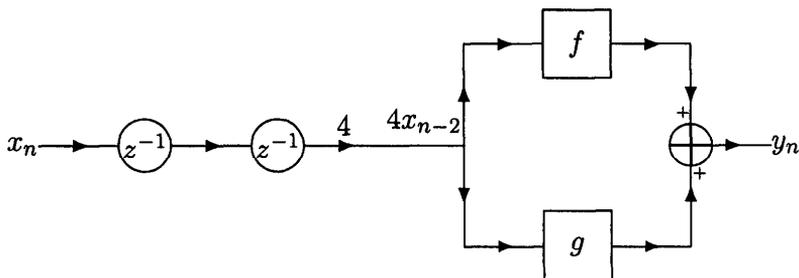
The first symmetry, which we have already stressed, is that graphs are to be understood topologically. Geometric quantities such as arc length, angles, and such are irrelevant. We can even perform mirror reflections drawing the whole picture backward as long as all the arrows are reversed (but please print the alphanumeric characters in the conventional orientation).

Since the topology remains unchanged one can always move a gain along an arc to a convenient place. You will doubtless recall that we did this when redrawing the basic FIR block. More generally, you can move a gain along an arc until the first addition or nonlinear system as long as you replicate it at tee connectors. You can also combine consecutive gains into a single gain or split a single gain into two in series.

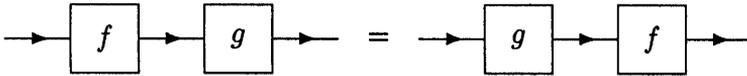
These operations are special cases of the 'like signal merging' symmetry. Whenever we find identical hidden signals in two different places, we can consolidate the graph, eliminating extraneous arcs and nodes, as long as no input or output signals are affected. For example, consider the graph



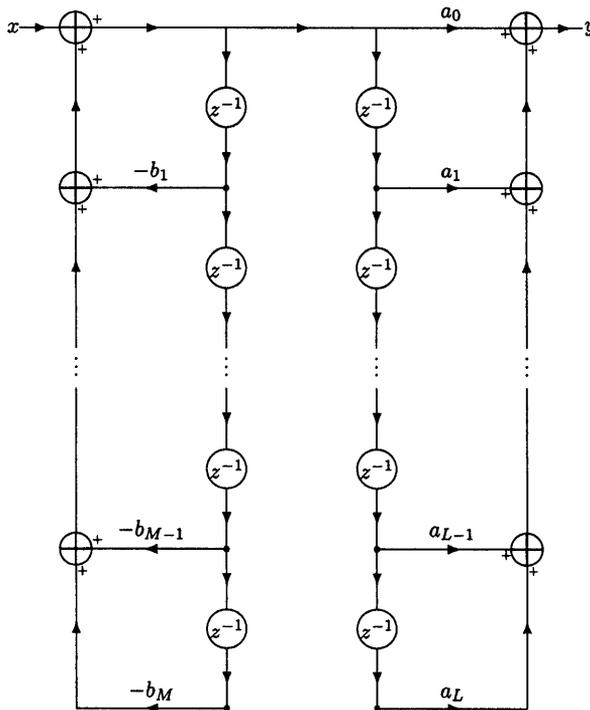
where we have identified the signal  $4x_{n-1}$  on two different arcs. We can consolidate everything between the input and these two signals.



Two signal processing systems  $f$  and  $g$  are said to ‘commute’ if one can interchange their order without changing the overall system.



In particular any two linear systems, and thus any two filters commute. We often exploit commutation of filters to simplify DSP flow graphs.



**Figure 12.10:** An alternative way to depict the ARMA filter. Here we perform the autoregressive (all-pole) filter before the moving average (FIR) filter.

As an example, let’s simplify Figure 12.8.B for the full ARMA filter. Note that there we actually performed the MA portion before the AR, which would make this an MAAR filter (were this possible to pronounce). Since the MA and AR subsystems are filters and thus commute we can place the AR portion before the MA part, and obtain Figure 12.10.

This diagram is just as symmetric as the first but seems to portray a different story. The signal first enters the infinite loop, cycling around and around inside the left subsystem, and only the signal that manages to leak



The grouping into higher-level subsystems results in a simplification similar to that of using subroutines in programming languages. Of course these new black boxes can be used in turn to recursively build up yet more complex subsystems. Such a system, built up from various levels of subsystems that can be graphically decomposed into simpler subsystems, is called a hierarchical flow graph. Recursively applying the ungrouping symmetries reduces hierarchical graphs to graphs composed solely of primitive processes.

## EXERCISES

- 12.3.1 Draw the basic all-pole block in four different ways.
- 12.3.2 Recall that the main entity of the state-space description is the system's internal state. One way to encode the internal state involves specifying hidden signals. Which hidden signals are required to be identified? How does the state thus specified evolve with time?
- 12.3.3 Give an example of two systems that do *not* commute.
- 12.3.4 Draw high-level hierarchical flow graphs for the following systems, and then decompose the high-level description into primitives.
  - a filter composed of several copies of the same FIR filter in series
  - a band-pass filter that mixes a signal down to zero frequency and then low-pass filters it
  - a circuit that computes the instantaneous frequency

## 12.4 RAX Externals

In the early eighties the author was working on signal analysis in a sophisticated signal processing lab. This lab, like most at that time, was composed largely of complex analog signal processing equipment mounted vertically in 19-inch racks. Each rack would typically house between five and ten different pieces of equipment, including function generators, amplifiers, filters, precision synthesizers, and oscilloscopes. Each piece of equipment conventionally had buttons and knobs on its front panel, and input and output connections on its back. These back panel connections would be routed to *patch panels* where the users could rapidly connect them up.

The lab had several analysis stations, and a typical station consisted of two or three racks full of complex and expensive equipment. Each individual

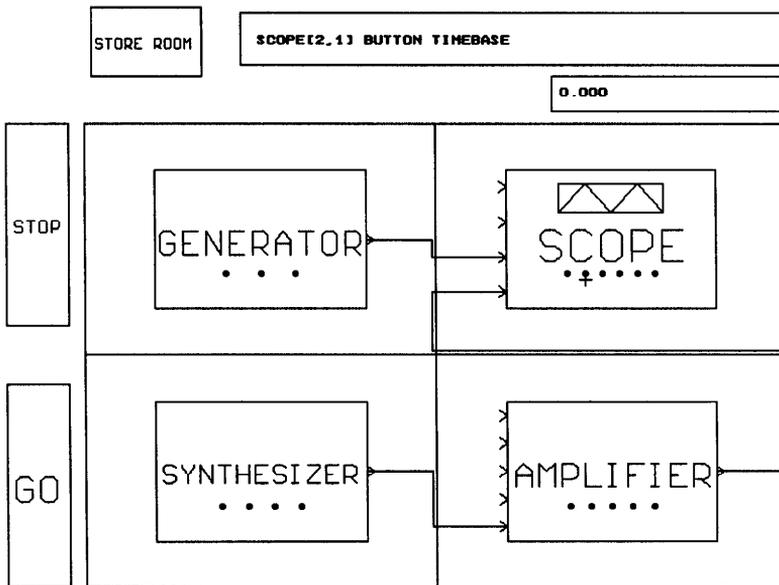
piece of equipment could cost tens to hundreds of thousands of dollars, would have to be calibrated and serviced regularly, and would usually take about two to three days to master. Just mounting a new box would take several hours, including placing it onto slides, screwing the slides into the racks, routing all the cables from its back panel to the patch panels, testing these cables (which would always seem to fail), and properly labeling the patch panel connectors. While the veteran lab staff could set up quite complex signal processing functions in minutes, someone new to the lab would go through a learning process of several months before feeling confident enough to work alone.

This lab was considered both modern and efficient. Outdated equipment was continually replaced with the most modern and sophisticated available; the lab staff was the most competent that could be found. However, trouble was definitely on the horizon. Maintenance costs were skyrocketing, the training of new lab staff was getting harder and lengthier, and even the most sophisticated equipment was not always sufficient for all the new challenges the lab faced. For these reasons we embarked on the development of an experimental software system.

The system was originally called *RACKS*, supposedly an acronym for **R**eplace **A**nalog **C**omponents with **K**nowledge-based **S**oftware, but actually referring to the *racks* of equipment the system emulated. The name was later shortened to RAX as an acronym for **R**eally **A**wesome **b**o**X**es. RAX was a visual programming environment that simulated the operation of an analysis station. Using a pointing device (originally a joystick, but you can think of it as a mouse if you prefer), equipment could be instantly taken out of a virtual store room, placed into virtual racks, connected by virtual cables, and operated. The operation of RAX was not always real-time, but it enabled useful analyses to be easily performed.

RAX was never commercially available, and is hardly state-of-the-art, not having been updated since its initial development. It was quite limited, for example, not allowing hierarchical definition of blocks. It was also not very run-time efficient, generally passing single samples between blocks. The host computer and DSP cards used as the platform for RAX are by now museum pieces. However, I have several reasons for expounding on it here. First and foremost is my own familiarity with its internals; many of the issues that we will examine are quite general, and I can discuss them with maximum knowledge regarding RAX. Second, RAX is relatively simple and thus easy to grasp, but at the same time general and easily extensible. Third, I promised my coworkers that one day I would finish the documentation, and better fifteen years late than never.

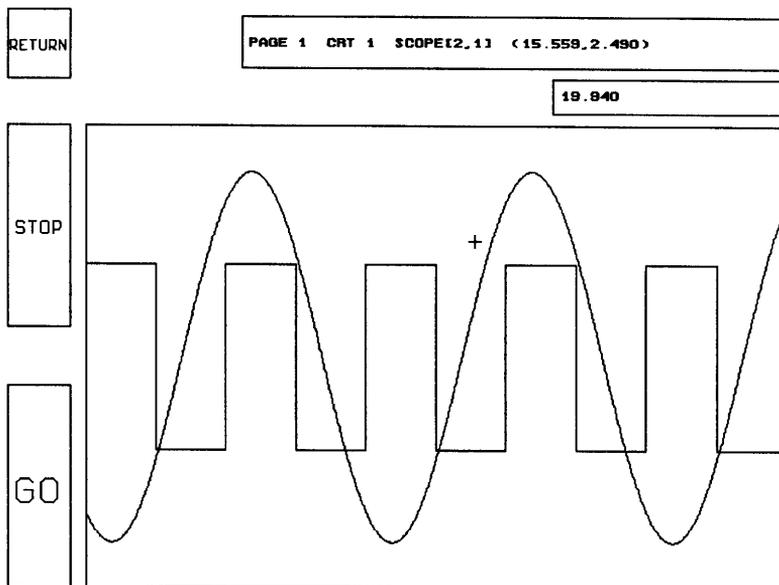
The RAX model of the world is that of *racks* of equipment, which are vertical rectangular arrays. These vertical arrays are called *racks*; racks are made up of *slots*, and each slot can hold a single piece of *equipment*. The piece of equipment in a specific rack and slot position is called a *box*, while what a type of equipment does is called its *function*. Each box has *input connectors*, *output connectors*, and *buttons*. *Cables* can be connected between a single output connector and any number of input connectors. Buttons are used to set internal parameters of the different boxes (e.g., input filenames and amplification gains).



**Figure 12.12:** The graphics screen of RAX for a simple setup. Note that this figure's resolution and fonts are representative of 1983 computer graphic displays (the original had 16 colors).

An example of a working RAX system is depicted in Figure 12.12. The resolution and fonts are representative of the technology of graphic displays circa 1985; the original screens had up to 16 colors, which are not observable here. In the figure we see a function generator, a synthesizer, an amplifier, and a scope connected in a frivolous way. The function generator is a box with no inputs and a single output. This output can be a square, triangular, or sawtooth wave, with buttons to select the signal type and control the amplitude and frequency. The synthesizer generates (real or complex) sinusoids of given amplitude, frequency, and phase. The amplifier (which would

never actually be used in this way since the synthesizer has adjustable amplitude) can take up to five inputs. Its output is the sum of these inputs, each multiplied by its own gain (selected by appropriate buttons). The scope has input channels for one or two signals to be displayed. Its other inputs are for external trigger and clock. The scope has no output, but buttons that set the volts per division, timebase, trigger mode and level, clock rate, number of sweeps. The scope also has a display, called CRT for **Cathode Ray Tube**. The output of the function generator is connected to channel B of the scope, while the synthesizer feeds the amplifier which in turn is connected to channel A of the scope. A sample scope display from this setup is depicted in Figure 12.13.



**Figure 12.13:** The graphics screen of the scope for the simple RAX setup depicted in the previous figure. The small + represents the position of the pointing device, and this position is indicated in the message window.

Although completely general, for reasons of efficiency RAX boxes are usually relatively high-level functions. For example, an FIR filter would be hand coded and called as a box, and not built up from individual multipliers and adders. It is relatively easy to add new functions as required; one need only code them using special conventions and link them into RAX. Computationally demanding functions may actually run on DSP processors, if they have been coded to exploit such processors and the processors are available.

RAX can be integrated into the real world in two ways. For non-real-time use there are ‘input file’ and ‘output file’ boxes. These boxes are stream-oriented, reading and writing as required. The input file box has one output and a button with which the user specifies the name of the file to read. Similarly, the output file box has a single input and a button to specify the file to write. For simple processing, or when DSP processors are used, there are also A/D and D/A boxes that stream to and from the true devices.

In RAX the same piece of equipment can be placed into many different rack-slot positions, which is interpreted as different boxes that happen to have the function. Boxes are identified by giving the function, the rack number, and the slot number (e.g., `SCOPE[2,1]`). Connectors and buttons have their own notations. When the pointing device enters a rack-slot that houses a piece of equipment its identifier is displayed in the message area at the upper right. When the pointer is close enough to a button or connector, its identifier is displayed as well. Pressing the pointer’s actuator (similar to clicking a mouse) over a button causes a pop-up menu to appear where the user can edit the corresponding parameter. Pressing the actuator near a connector causes a ‘rubber band line’ to be drawn from that connector to the pointer, which can then be placed near another connector and pressed again. If the connection is valid the rubber band line disappears and in its place a connection route is drawn. Valid connections connect a single output to any number of inputs. The connection route is drawn to avoid existing routes, and is color coded for optimal distinguishability.

After bringing up the application, the user specifies the number of racks and the number of slots per rack. These numbers can be changed at any time, with the restriction that no mounted equipment should fall onto the floor. Next the user opens the store room and drags pieces of equipment from there, placing them into rack-slots. The user can then connect output connectors to input connectors and set parameters using the buttons. When satisfied the user points and depresses the run button. At that point time starts to run and the signal processing begins. The user may at any time select any display (e.g., from a scope, spectrum analyzer, or voltmeter) and view the graphic results. When such a display is active, the message area continuously displays the pointer’s coordinates, for example, volts and time for the scope. To return to the racks display the user can then press the return button, and equipment buttons can be adjusted while time is running. To stop time from running there is a stop button. If the user considers the setup to be useful it can be saved to a *netlist* file, from which it can be loaded at some later date. The netlist for the simple setup of Figure 12.12 is printed out as Figure 12.14.

```

SAMPLING FREQUENCY = 100.00000
RACK 1
  SLOT 1 GENERATOR
                                FUNC=SQUARE
                                FREQ=1
                                VOLT=1
                                OUT>SCOPE[2,1].CHANB
  SLOT 2 SYNTHESIZER
                                FREQ=1
                                VOLT=1
                                PHASE=1
                                COMPL=REAL
                                OUT>AMPLIFIER[2,2].IN
RACK 2
  SLOT 1 SCOPE
                                VOLTDIV=1
                                TIMEBASE=1
                                TRIGMODE=FREE
                                CLOCK=0
                                SWEEPS=1
                                TRIGLEV=0
                                CHANA<AMPLIFIER[2,2].OUT
                                CHANB<GENERATOR[1,1].OUT
  SLOT 2 AMPLIFIER
                                GAIN=1
                                GAIN2=0
                                GAIN3=0
                                GAIN4=0
                                GAIN5=0
                                IN<SYNTHESIZER[1,2].OUT
                                OUT>SCOPE[2,1].CHANA

```

**Figure 12.14:** The *netlist* of the simple example.

## EXERCISES

12.4.1 RAX is a 'clock-driven' system, meaning that some external concept of time causes the scheduler to operate. Alternatives include 'data-driven' and 'control-driven' systems. In the former, external inputs are the trigger for everything to happen; each input is followed through causing box after box to operate in turn. In the latter, whenever a box cannot run due to an input not being ready, the box connected to it is run in order to generate that input (and recursively all boxes before it). Discuss the advantages and disadvantages of these three techniques.

- 12.4.2 Observe in Figure 12.12 that different cables never overlap, at most they cross at a point. Give a simple algorithm for accomplishing this.
- 12.4.3 Find some visual programming language to experiment with. How is it similar to and how is it different from RAX?
- 12.4.4 What are RAX's main functional deficiencies?

## 12.5 RAX Internals

Now let's start to peek behind the scenes to see how RAX accomplishes its magic. The first thing we must explain is that RAX is an interpreter rather than a compiler. The entire RAX run-time system must be present for anything to happen and the GUI, IO, signal display, task scheduling, and processing are all supplied by RAX itself. Modern systems will usually allow the user to operate in interpreted mode in order to debug the system and then to compile to some standalone language such as C or DSP assembly language. This point understood, let us proceed to the program's structure.

The main program looks like this in pseudocode:

```
initializations
main loop
    handle user events
    update graphics
    if RunMode
        schedule tasks
        increment time
finalizations
```

where initializations and finalizations refer to the opening and closing of files, allocation and deallocation of memory, starting and terminating the graphics environment, and other mundane computer tasks. Handling user events refers to checking for motion of the pointing device and updating the message area accordingly; and checking for pointer device button presses or keyboard entry and the corresponding changing of parameters and program modes. RunMode is true whenever the user has pressed the START button, and stays true until the STOP button is pressed.

In the analog world every box is operating all the time. This has to be emulated in RAX since the main and all the boxes run on a single CPU.

This is the responsibility of the *task scheduler*. The scheduler runs through every piece of equipment in the racks, and decides whether it is ready to run. This decision is based on continuity of time and an assumption of causality. Each box remembers when it last ran, and each input to each box contains a time-stamped value. A box can run only when the present time is strictly after the time it last ran, and only when the time is not before the time of its inputs. Assuming a box can run, the scheduler is responsible for loading the box state, connecting input and output cables, calling the proper task (perhaps running on a DSP), and storing the new state. These duties determine the *context switch time*, the minimum time it takes to switch from running one box to another. RAX was somewhat wasteful in this regard, having originally been designed for simulation purposes and only later being retrofitted with DSP boards for real-time use. An alternative strategy (one that was employed for the DSP code) is for each box to keep its state information internally. This cannot be done using static arrays for host code since one equipment type can be used multiple times in a single setup.

Finally, when all boxes have run for the specified time, the time is incremented according to the present sampling rate. One of the major limitations of RAX is the use of a single sampling rate. Although the sampling rate can be changed, there cannot be simultaneously more than one rate, and all boxes must use the same clock. This is both an efficiency problem (some processes might only need updating very infrequently, but must be scheduled every time) and a real constraint (resampling processes, such as those required for modems, cannot be implemented). This problem could be fixed by simulating real time using a common multiple of all desired sampling rates and dividing as required. Hardware systems commonly implement this same policy by using a high-frequency crystal oscillator and various frequency dividers.

Behind the simple description of the handling of user events and scheduler are a plethora of infrastructure functions. For example, there is a function that given the rack and slot numbers determines whether the rack-slot is occupied, and if so retrieves the type of equipment. Another, given a type of equipment, finds the meaning of the inputs or buttons. Given a cable identifier, the output number, origin rack-slot that feeds it, and all inputs it feeds can be found. Given an entire configuration, the minimum number of racks and slots may be calculated. Of course there are functions to place a piece of equipment in a given position, to remove a piece of equipment, to connect a cable between connectors, etc.

Behind every graphics display (scope, spectrum analyzer, etc.) there is a 'display list' that contains all lines drawn to that display. Whenever the display is to be shown the display list lines are translated into screen coördinates and plotted. Every motion of the pointer device in such a screen requires translation from screen coördinates back to world coördinates.

Now let's discuss how rack-slots are populated and how equipment is described. Every box placed in a rack-slot position is assigned a unique identification number, starting from 1 and reaching the total number of occupied positions. This identifier is used by the scheduler, which loops from 1 to the total number of boxes in its main loop. There is a vector of this length that holds the position and an equipment pointer, this pointer in turn bringing us to the equipment type, state, parameters, cabling, time last run, processor (0 for host, otherwise DSP number), and equipment function.

How is this function defined? When we wish to add a new function we must define a constructor that returns the allocated and initialized state, buttons, inputs, and outputs; and a destructor that undoes all of the above. Then we write the *run* routine. This routine takes the state, buttons, and inputs, and returns the updated state, and outputs (and possibly updates the graphic display). All run routines tend to have the same form. First the buttons are read. Then the inputs are checked for correctness and type (since boxes react differently depending on type, for example, amplification of a complex input returns a complex output, while a real returns a real). One time step is then performed, generating values that are placed into the appropriate outputs. If there is a graphics display, its display list is updated. Finally, the state and time variables are updated, and control is returned to the scheduler.

Functions that run on DSP processors are built slightly differently. These functions store their state locally and are not as tightly controlled as their native counterparts. For them the constructor consists of downloading the object code to the appropriate processor and noting this fact. The run routine on the host simply passes the inputs to the processor through shared memory (SHAM) and collects the outputs. The DSP code is written as an infinite loop that checks for the appearance of new inputs in the SHAM, processes, and copies the outputs to the SHAM.

How are all the disjoint entities coupled to make a single coherent system? This is a general problem in systems with many functional parts, and there are in general four possibilities. These possibilities are usually known as compile-time, link-time, download-time, and run-time bonding. The simplest method is to gather all the program code for all the different kinds of equipment together into a single file, and compile this file together. In

such an implementation there can be one global constructor and destructor procedure, which allocates and deallocates all memory required for all boxes. Such compile-time bonding results in very fast run-time code, but is extremely inefficient from the memory utilization point of view. The object code for every sort of equipment is present even if we need only a few boxes, although in modern paging virtual memory systems this may not actually impact performance. A more serious design flaw is the complexity and lack of flexibility of the code.

The next possibility is link-time bonding. Here each function is defined in a separate file that is separately compiled into an object file. The linker is responsible for bonding all the object files together. Simple link-time bonding may still waste memory for unused functions, but with proper operating system support the unused functions may take up executable file size but not actually sit in run-time memory. Also, the reduction of the interdependence of the different functions reduces system complexity by forcing object-oriented techniques.

Download-time bonding involves compiling and linking each function into a separately executable program. When the system is run the control system selects which function programs need to be downloaded and launched, and then either the control system, the operating system, or hardware are responsible for moving data between these programs. For example, the data may be passed between these programs using 'interprocess communications' or 'sockets' or 'pipes', or each program may run on separate DSP processors with hardware communications links between them.

The most complex and most memory-efficient form of bonding is run-time, also known as dynamic allocation. Like download-time bonding, each function is a separate program unit. However, functions may be loaded and launched during the running of the system. This is a particularly useful feature when the functioning of the system depends on the input signal. For example, consider a voicemail system that must decode DTMF tones, compress and store speech, and demodulate, decode, and store facsimile transmissions. Since DTMF tones are used to control the system and may be used at any time, the DTMF decoder must be continuously available. When a session commences the speech function may be loaded by default, but upon detection of fax tones the facsimile function must take its place.

In the original implementation of RAX the host code for each piece of equipment was compiled separately, but the entire program was linked together before execution. This link-time bonding was chosen since the programming system used did not support dynamic allocation for user routines. The DSP code, however, was compiled into individually executable pro-

grams, and downloaded upon pressing START. This download-time bonding could be efficiently performed since it only required loading the appropriate DSP code and data, and releasing the processor.

The full algorithm for bonding went something like this. Each function for which there was DSP code was assigned a complexity number between 1 and 10; functions with no DSP code received a zero. For example, the  $N$  sample delay had complexity 0, the amplifier was given a 1, while the spectrum analyzer (which had to window, perform FFT, square, average, take logarithms, perform graphics, etc.) rated a 10. In no case did the functions exceed the capabilities of a single DSP processor. Whenever the configuration changed and START was pressed, the functions with nonzero complexity were sorted in descending order of complexity. Assuming there were  $P$  available processors, the first  $P$  functions would be downloaded to DSPs, while the rest of the functions would run on the host processor.

We have yet to fully explain the cable mechanism. Cables are internally arranged in a array, every cable having one source and any number of sinks. Each element in the cable array consists of three parts, namely a time, a type (boolean, integer, real, or complex), and a value. When a box computes an output value, the time, this value and its type are written into the cable array where they can be read by all those boxes that require it. In RAX only one value can be placed into a given cable at a time; once a value is placed in the cable it remains there until overwritten. As mentioned before, boxes test the time and value on the cable before using the value.

This implementation of cables, which we call *overwrite*, allows multiple boxes to receive a single cable as input, but assumes that boxes always take values they will need, even if they are not yet ready to use them. For example, think of a Fourier analysis box that collects an entire buffer of signal values before calculating an entire spectrum. This box must be activated each time an input value appears, just in order to store away this value into its state. This wastes context switches, potentially slowing the system down. There are several alternative strategies that can be used.

An alternative, called *buffered write*, would be for the cable to collect past values into an ordered list, and for the Fourier analysis box to be called only when the desired number of inputs are all ready. In such an alternative representation we must consider what is to be done about clearing past values that are no longer needed. One possibility is for cables to implement a fixed-length buffer, always holding some number of past values. This is easy to implement, for example, by a circular buffer, but has two problems. First it limits the generality of what a box can do, since no box can use more history than what the cable stores. Second many boxes might only require

the present value or a few past ones, and we might be wasting memory by storing the maximum number. Of course we *could* configure each cable differently, choosing the number of values to store according to the maximum needed by boxes to which the cable is connected.

Another possibility is to allow only one box to be fed by a given cable and to make it that box's responsibility to clear unneeded values from the cable. In such an implementation the precable box *writes* to the cable, and the post-cable box *reads* from it deleting the value; thus we call this method 'write-read'. This is similar to the mechanisms of 'pipes' and 'sockets' that are provided by many operating systems, and these mechanisms can be exploited in implementing such cables. In a system with cables that are written and read we could allow several boxes to write to a single cable, but only one box can read a cable since once a value has been read it is no longer available for other boxes to use. This is not really an insurmountable limitation since we could easily create a tee connector equipment type, which takes in a value from one input and makes it available on two or more outputs.

We have still not completely specified what happens in systems with cables of the latter type. One possibility is for the read attempt to fail if the desired values are not ready, or equivalently, to give the reading box the ability to test for readiness before attempting to read from a cable. Such 'write-test-read' systems can seem to act nondeterministically, even without explicit randomness built in. For example, consider a piece of equipment built to merge two inputs into a single output. This equipment checks its inputs until it finds one which is ready and writes its value to the output. It then reads and discards all other inputs that might be simultaneously available. Even if it always sweeps through its inputs in the same order, its output depends on the detailed operation of the scheduler, and thus seems unpredictable. A second possibility is for the reading box to become blocked until the desired input value is ready. Indeed the blocking mechanism can be used as the heart of the system instead of an explicitly encoded scheduler. All boxes in a 'blocked-read' system simply run in parallel, with the unblocked boxes preparing outputs that in time unblock other processes. Finally, the writing process may become blocked until the cable is ready to receive the value to be sent. Although this 'blocked-write' method seems strange at first, it shares with blocked-read the advantage of automatically synchronizing truly parallel processes.

Since RAX was an 'overwrite' system, DSP processors could only be employed for their relative speed as compared with the host processor available at that time. The potential for parallel processing could not be exploited

since the scheduler was responsible for sending data to each processor, waiting for it to complete its computation, and then collecting its output. In order to allow the processors to truly run in parallel some method of synchronization, either that inherent in blocked-read and blocked-write or an explicit interprocess communications method, must be employed. One model that has been exploited for the parallelization of DSP tasks is Hoare's communicating sequential processes. In this model a collection of computational processes, each of which separately runs sequentially, run truly in parallel and communicate via unidirectional blocked-write channels.

This completes our description of RAX internals. While some of the details are specific to this system, many of the concepts are applicable to any visual programming system. When using such a system for simple tasks the analogy with analog equipment is enough to get you started, but for more complex problems a basic understanding of the internals may mean the difference between success and frustration.

## EXERCISES

- 12.5.1 Write a package to implement graphics display lists as singly linked lists of commands. At minimum there must be `MOVE (x,y)` and `DRAW (x,y)` commands, while more elaborate implementations will have other opcodes such as `SETCOLOR c`, `DRAWRECTANGLE (left, right, bottom, top)`, and `CIRCLE (x,y,r)`. Remember to include routines to construct a new display list, clear the display list (as, for example, when a scope retriggers), add a command to the display list (take into account that the display may or not be currently showing), show the display on screen, translate between real world coordinates and integer screen coordinates, and free the list.
- 12.5.2 Write a package to handle netlist files. You will need at least one routine to read a netlist file and translate it into an internal representation of boxes and parameters and one routine to write a netlist file.
- 12.5.3 Write a RAX-like GUI.
- 12.5.4 Write a RAX-like scheduler.
- 12.5.5 Implement a basic RAX system for functions that all run on the host. You should supply at least a sine wave generator, a filter, and a scope.

## Bibliographical Notes

A good general reference on graph theory is [91], which is a newer version of a classic text.

The present author has not found any mention of the use of flow graphs for signal processing before the 1953 article of Mason [160, 161].

One of the earliest uses of visual programming in DSP was BLODI (the **B**LOCK **D**IAGRAM compiler), which was developed Bell Labs in late 1960. Although without a true graphic interface, it was said to be easier to learn than FORTRAN, and at times easier to use even for the experienced programmer. BLODI had blocks for IO, signal and noise generation, arithmetic operations between signals, delay and FIR filtering, sampling and quantization, and even a flip-flop. Other documented flow languages for signal processing include SIGNAL [88] and LUSTRE [89]. Probably the most popular visual programming environment for signal processing is what was once called BOSS (**B**lock-**O**riented **S**ystem **S**imulator) but was later renamed SPW (**S**ignal **P**rocessing **W**orkSystem).

Hoare's communicating sequential processes, presented in [103], also motivated several DSP systems. For a good discussion of implementational issues for data-flow-oriented languages consult [2].