a new thread creates a new stream object with input stream `S0`, output stream `T0`, and initial state `X0`. The stream object reads messages from the input stream, does internal calculations, and sends messages on the output stream. In general, an object can have any fixed number of input and output streams.

Stream objects can be linked together in a graph, where each object receives messages from one or more other objects and sends messages to one or more other objects. For example, here is a pipeline of three stream objects:

```
declare S0 T0 U0 V0 in
thread {StreamObject S0 0 T0} end
thread {StreamObject T0 0 U0} end
thread {StreamObject U0 0 V0} end
```

The first object receives from `S0` and sends on `T0`, which is received by the second object, and so forth.

4.3.5 Digital logic simulation

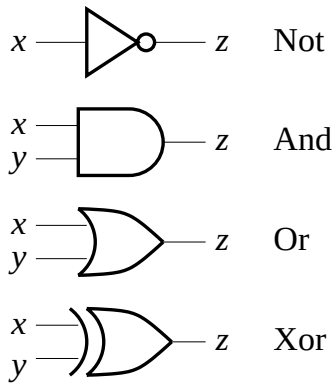
Programming with a directed graph of stream objects is called *synchronous programming*. This is because a stream object can only perform a calculation after it reads one element from each input stream. This implies that all the stream objects in the graph are synchronized with each other. It is possible for a stream object to get ahead of its successors in the graph, but it cannot get ahead of its predecessors. (In Chapter 8 we will see how to build active objects which can run completely independently of each other.)

All the examples of stream communication we have seen so far are very simple kinds of graphs, namely linear chains. Let us now look at an example where the graph is not a linear chain. We will build a digital logic simulator, i.e., a program that faithfully models the execution of electronic circuits consisting of interconnected logic gates. The gates communicate through time-varying signals that can only take discrete values, such as 0 and 1. In *synchronous* digital logic the whole circuit executes in lock step. At each step, each logic gate reads its input wires, calculates the result, and puts it on the output wires. The steps are cadenced by a circuit called a *clock*. Most current digital electronic technology is synchronous. Our simulator will be synchronous as well.

How do we model signals on a wire and circuits that read these signals? In a synchronous circuit, a signal varies only in discrete time steps. So we can model a *signal* as a stream of 0's and 1's. A *logic gate* is then simply a stream object: a recursive procedure, running in its own thread, that reads input streams and calculates output streams. A *clock* is a recursive procedure that produces an initial stream at a fixed rate.

Combinational logic

Let us first see how to build simple logic gates. Figure 4.16 shows some typical gates with their standard pictorial symbols and the boolean functions that define



		z			
x	y	Not	And	Or	Xor
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Figure 4.16: Digital logic gates

them. The exclusive-or gate is usually called `Xor`. Each gate has one or more inputs and an output. The simplest is the `Not` gate, whose output is simply the negation of the input. In terms of streams, we define it as follows:

```
fun {NotGate Xs}
   case Xs of X|Xr then (1-X)|{NotGate Xr} end
end
```

This gate works instantaneously, i.e., the first element of the output stream is calculated from the first element of the input stream. This is a reasonable way to model a real gate if the clock period is much longer than the gate delay. It allows us to model *combinational* logic, i.e., logic circuits that have no internal memory. Their outputs are boolean functions of their inputs, and they are totally dependent on the inputs.

How do we connect several gates together? Connecting streams is easy: the output stream of one gate can be directly connected to the input stream of another. Because all gates can execute simultaneously, each gate needs to execute inside its own thread. This gives the final definition of `NotG`:

```
local
   fun {NotLoop Xs}
      case Xs of X|Xr then (1-X)|{NotLoop Xr} end
   end
in
   fun {NotG Xs}
      thread {NotLoop Xs} end
   end
end
```

Calling `NotG` creates a new `Not` gate in its own thread. We see that a working logic gate is much more than just a boolean function; it is actually a concurrent entity that communicates with other concurrent entities. Let us build other kinds of gates. Here is a generic function that can build any kind of two-input gate:

```
fun {GateMaker F}
```

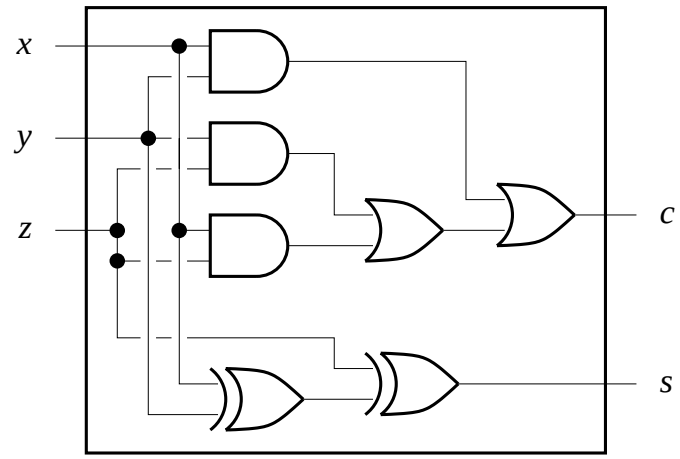


Figure 4.17: A full adder

```
fun {$ Xs Ys}
   fun {GateLoop Xs Ys}
      case Xs#Ys of (X|Xr)#(Y|Yr) then
         {F X Y}|{GateLoop Xr Yr}
      end
   end
in
   thread {GateLoop Xs Ys} end
end
end
```

This function is a good example of higher-order programming: it combines genericity with instantiation. With it we can build many gates:

```
AndG ={GateMaker fun {$ X Y} X*Y end}
OrG  ={GateMaker fun {$ X Y} X+Y-X*Y end}
NandG={GateMaker fun {$ X Y} 1-X*Y end}
NorG ={GateMaker fun {$ X Y} 1-X-Y+X*Y end}
XorG ={GateMaker fun {$ X Y} X+Y-2*X*Y end}
```

Each of these functions creates a gate whenever it is called. The logical operations are implemented as arithmetic operations on the integers 0 and 1.

Now we can build combinational circuits. A typical circuit is a *full adder*, which adds three one-bit numbers, giving a two-bit result. Full adders can be chained together to make adders of any number of bits. A full adder has three inputs, x, y, z, and two outputs c and s. It satisfies the equation $x+y+z = (cs)_2$. For example, if $x = 1$, $y = 1$, and $z = 0$, then the result is $c = 1$ and $s = 0$, which is $(10)_2$ in binary, namely two. Figure 4.17 defines the circuit. Let us see how it works. c is 1 if at least two inputs are 1. There are three ways that this can happen, each of which is covered by an `AndG` call. s is 1 if the number of 1 inputs is odd, which is exactly the definition of exclusive-or. Here is the same

circuit defined in our simulation framework:

```
proc {FullAdder X Y Z ?C ?S}
   K L M
in
   K={AndG X Y}
   L={AndG Y Z}
   M={AndG X Z}
   C={OrG K {OrG L M}}
   S={XorG Z {XorG X Y}}
end
```

We use procedural notation for `FullAdder` because it has two outputs. Here is an example of using the full adder:

```
declare
X=1|1|0|_
Y=0|1|0|_
Z=1|1|1|_ C S in
{FullAdder X Y Z C S}
{Browse inp(X Y Z)#sum(C S)}
```

This adds three sets of input bits.

Sequential logic

Combinational circuits are limited because they cannot store information. Let us be more ambitious in the kinds of circuits we wish to model. Let us model *sequential* circuits, i.e., circuits whose behavior depends on their own past output. This means simply that some outputs are fed back as inputs. Using this idea, we can build *bistable* circuits, i.e., circuits with two stable states. A bistable circuit is a memory cell that can store one bit of information. Bistable circuits are often called flip flops.

We cannot model sequential circuits with the approach of the previous section. What happens if we try? Let us connect an output to an input. To produce an output, the circuit has to read an input. But there is no input, so no output is produced either. In fact, this is a *deadlock* situation since there is a cyclic dependency: output waits for input and input waits for output.

To correctly model sequential circuits, we have to introduce some kind of time delay between the inputs and the outputs. Then the circuit will take its input from the *previous* output. There is no longer a deadlock. We can model the time delay by a *delay gate*, which simply adds one or more elements to the head of the stream:

```
fun {DelayG Xs}
   0|Xs
end
```

For an input $a|b|c|d|\ldots$, `DelayG` outputs `0`$|a|b|c|d|\ldots$, which is just a delayed version of the input. With `DelayG` we can model sequential circuits. Let

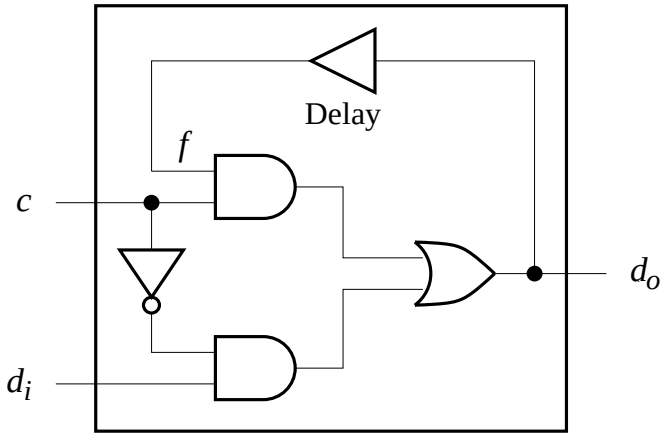


Figure 4.18: A latch

us build a *latch*, which is a simple kind of bistable circuit that can memorize its input. Figure 4.18 defines a simple latch. Here is the program:

```
fun {Latch C DI}
   DO X Y Z F
in
   F={DelayG DO}
   X={AndG F C}
   Z={NotG C}
   Y={AndG Z DI}
   DO={OrG X Y}
   DO
end
```

The latch has two inputs, `C` and `DI`, and one output, `DO`. If `C` is 0, then the output tracks `DI`, i.e., it always has the same value as `DI`. If `C` is 1, then the output is frozen at the last value of `DI`. The latch is bistable since `DO` can be either 0 or 1. The latch works because of the delayed feedback from `DO` to `F`.

Clocking

Assume we have modeled a complex circuit. To simulate its execution, we have to create an initial input stream of values that are discretized over time. One way to do it is by defining a *clock*, which is a timed source of periodic signals. Here is a simple clock:

```
fun {Clock}
   fun {Loop B}
      B|{Loop B}
   end
in
   thread {Loop 1} end
```

```
proc {Gate X1 X2 ... Xn Y1 Y2 ... Ym}
   proc {P S1 S2 ... Sn U1 U2 ... Um}
      case S1#S2#...#Sn
      of (X1|T1)#(X2|T2)#...#(Xn|Tn) then
         Y1 Y2 ... Ym
         V1 V2 ... Vm
      in
         {GateStep X1 X2 ... Xn Y1 Y2 ... Ym}
         U1=Y1|V1
         U2=Y2|V2
         ...
         Um=Ym|Vm
         {P T1 T2 ... Tn V1 V2 ... Vm}
      end
   end
in
   thread {P X1 X2 ... Xn Y1 Y2 ... Ym} end
end
```

Figure 4.19: A linguistic abstraction for logic gates

```
end
```

Calling `{Clock}` creates a stream that grows very quickly, which makes the simulation go at the maximum rate of the Mozart implementation. We can slow down the simulation to a human time scale by adding a delay to the clock:

```
fun {Clock}
   fun {Loop B}
      {Delay 1000} B|{Loop B}
   end
in
   thread {Loop 1} end
end
```

The call `{Delay N}` causes its thread to suspend for `N` milliseconds and then to become running again.

A linguistic abstraction for logic gates

In most of the above examples, logic gates are programmed with a construction that always has the same shape. The construction defines a procedure with stream arguments and at its heart there is a procedure with boolean arguments. Figure 4.19 shows how to make this construction systematic. Given a procedure `GateStep`, it defines another procedure `Gate`. The arguments of `GateStep` are booleans (or integers) and the arguments of `Gate` are streams. We distinguish the gate's inputs and outputs. The arguments `X1`, `X2`, ..., `Xn` are the gate's inputs. The arguments `Y1`, `Y2`, ..., `Ym` are the gate's outputs. `GateStep` defines

the instantaneous behavior of the gate, i.e., it calculates the boolean outputs of the gate from its boolean inputs at a given instant. `Gate` defines the behavior in terms of streams. We can say that the construction *lifts* a calculation with booleans to become a calculation with streams. We could define an abstraction that implements this construction. This gives the function `GateMaker` we defined before. But we can go further and define a linguistic abstraction, the **`gate`** statement:

`gate input` ⟨x⟩$_1$ ⋯ ⟨x⟩$_n$ **`output`** ⟨y⟩$_1$ ⋯ ⟨y⟩$_m$ **`then`** ⟨s⟩ **`end`**

This statement translates into the construction of Figure 4.19. The body ⟨s⟩ corresponds to the definition of `GateStep`: it does a boolean calculation with inputs ⟨x⟩$_1$ ⋯ ⟨x⟩$_n$ and outputs ⟨y⟩$_1$ ⋯ ⟨y⟩$_m$. With the **`gate`** statement, we can define an `And` gate as follows:

```
proc {AndG X1 X2 ?X3}
   gate input X1 X2 output X3 then X3=X1*X2 end
end
```

The identifiers `X1`, `X2`, and `X3` refer to different variables inside and outside the statement. Inside they refer to booleans and outside to streams. We can embed **`gate`** statements in procedures and use them to build large circuits.

We could implement the **`gate`** statement using Mozart's parser-generator tool `gump`. Many symbolic languages, notably Haskell and Prolog, have the ability to extend their syntax, which makes this kind of addition easy. This is often convenient for special-purpose applications.

4.4 Using the declarative concurrent model directly

Stream communication is not the only way to program in the declarative concurrent model. This section explores some other techniques. These techniques use the declarative concurrent model directly, without taking advantage of an abstraction such as stream objects.

4.4.1 Order-determining concurrency

"In whichever order these twenty-four cards are laid side by side, the result will be a perfectly harmonious landscape."
– From *"The Endless Landscape": 24-piece Myriorama*, Leipzig (1830s).

A simple use of concurrency in a declarative program is to *find the order* of calculations. That is, we know which calculations have to be done, but because of data dependencies, we do not know their order. What's more, the order may depend on the values of the data, i.e., there is no one static order that is always right. In this case, we can use dataflow concurrency to find the order automatically.