Asynchronous Circuit Design. Chris J. Myers
Copyright © 2001 by John Wiley & Sons, Inc.
ISBNs: 0-471-41543-X (Hardback); 0-471-22414-6 (Electronic)

3 Communication Protocols

(The Chief Programmer) needs great talent, ten years experience and considerable systems and applications knowledge, whether in applied mathematics, business data handling, or whatever.
—Fred P. Brooks, *The Mythical Man Month*

Never put off until run time what you can do at compile time.
—David Gries

In the beginning was the word. But by the time the second word was added to it, there was trouble. For with it came syntax ....
—John Simon

In this chapter we describe the *handshake protocols* that are used to implement channel communication and methods of translation from a channel-level specification to a protocol-level description.

3.1 BASIC STRUCTURE

As we saw in Chapter 1, a channel communication can be implemented through a handshake protocol on two or more signal wires. One or more wires are used to *request* the communication; the others are used to *acknowledge* completion of the communication. In this section we describe how handshake protocols can be described in VHDL. Throughout the rest of the chapter, we use similar language constructs to describe alternative protocols. One of the alternative protocols from this chapter is given below. This protocol is for a passive/active shop which communicates one bit of data (i.e., one of two different types of wine) using a *dual-rail protocol.* In this section we describe the basic structure and syntax. The behavior and rationale of this and alternative protocols are described in future sections.

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;
entity shopPA_dualrail is
  port(bottle1:in std_logic;
       bottle0:in std_logic;
       ack_wine:buffer std_logic:='0';
       shelf1:buffer std_logic:='0';
       shelf0:buffer std_logic:='0';
       ack_patron:in std_logic);
end shopPA_dualrail;
architecture hse of shopPA_dualrail is
begin
shopPA_dualrail:process
begin
  guard(ack_patron,'0');
  guard_or(bottle0,'1',bottle1,'1');
  if bottle0 = '1' then assign(shelf0,'1',1,3);
  elsif bottle1 = '1' then assign(shelf1,'1',1,3);
  end if;
  assign(ack_wine,'1',1,3);
  guard(ack_patron,'1');
  vassign(shelf0,'0',1,3,shelf1,'0',1,3);
  guard_and(bottle0,'0',bottle1,'0');
  assign(ack_wine,'0',1,3);
end process;
end hse;
```

The first difference from the channel-level model is inclusion of the *handshake* package instead of the *channel* package. This package can be found in Appendix A, and it includes the definitions of the procedures: *guard*, *guard_or*, *guard_and*, *assign*, and *vassign*. The next change is the replacement of the *WineryShop* port of type channel with ports of type *std_logic*. The first of these new ports are *bottle1* and *bottle0*, which are used to tell the shop to accept a new bottle of wine of type 1 or type 0, respectively. The next port is *ack_wine*, which is used to indicate acknowledgment of the wine delivery to the shop. Although this signal is an *output*, it must be of mode *buffer* because it is also tested within the architecture. This port is initialized to '0'. The *ShopPatron* channel is implemented with the next three ports. The first two ports, *shelf1* and *shelf0*, are output ports used to communicate a single bit of data to the patron (i.e., the type of wine). The last port is *ack_patron*, which is used by the *patron* to acknowledge receipt of the wine.

In this protocol, the first thing the shop does is wait until *ack_patron* is '0'. This is accomplished using the *guard* procedure. The procedure *guard*(s,v) takes a signal, s, and a value, v, and replaces the following code:

```
if (s /= v) then
  wait until s = v;
end if;
```

The reason for the *if* statement in the *guard* procedure is that when a wait statement is encountered, if the expression is *already* true, the process stalls until the expression goes false and becomes true again. In this case, *ack_patron* starts low, so if we did not first test the signal, the process above would stall waiting for an event on *ack_patron*. However, *ack_patron* is already '0', and it does not change again unless this process sets *shelf1* or *shelf0*. This process, however, will not change these signals until it is woken up with an event on *ack_patron*. Therefore, this process is deadlocked. In order to address this problem, each wait must be predicated with a test to make sure that the expression is false before the wait statement is executed.

The next step in the protocol is to wait until either *bottle0* or *bottle1* goes high. This is accomplished using the *guard_or* procedure. The procedure *guard_or*($s1$,$v1$,$s2$,$v2$,...) takes a set of signals and values and stalls a process until some signal s_i has taken value v_i, and is defined as follows:

```
if ((s1 /= v1) and (s2 /= v2) ...  ) then
  wait until (s1 = v1) or (s2 = v2) ...;
end if;
```

After *bottle0* or *bottle1* goes high, the protocol next sets *shelf0* or *shelf1* high, depending on which of the bottle wires goes high. It sets the appropriate shelf signal using the *assign* procedure. The procedure *assign*(s,v,l,u) takes a signal, s, a value, v, a lower bound of delay, l, and an upper bound of delay, u, and it replaces the following code:

```
assert (s /= v)
  report ''Vacuous assignment!''
  severity failure;
s <= v after delay(l,u);
wait until s = v;
```

The *assert statement* causes the assignment to fail if the signal already has the value being assigned. It is often the case that this indicates a error in the specification. The *delay* function in the actual signal assignment is from the *nondeterminism* package (see Appendix A). This function is used to simulate assignments that happen after a random delay. The function delay takes a lower and upper bound and returns a delay between the two. The use of delay makes it easier to debug, as it separates the transitions in time. The wait statement at the end is necessary because in VHDL when signal assignment statements are executed, they schedule events but do not actually change the value of the signal. If we removed the wait statement from the *assign* procedure, the shelf signal could be scheduled to change in two time units. Next, *ack_wine* could be scheduled to change in one time unit. The result would be that *ack_wine* would go high before the shelf signal changes value. Since the desired behavior is to order the two signal assignments, it is necessary to add the wait statement in the *assign* procedure.

The *assign* procedure also allows parallel assignments. For example, the procedure call *assign*($s1$,$v1$,$l1$,$u1$,$s2$,$v2$,$l2$,$u2$) replaces the following code:

```
assert ((s1 /= v1) or (s2 /= v2))
  report ''Vacuous assignment!''
  severity failure;
s1 <= v1 after delay(l1,u1);
s2 <= v2 after delay(l2,u2);
wait until (s1 = v1) and (s2 = v2);
```

After the appropriate shelf signal goes high, *ack_wine* is set high and the shop waits for *ack_patron* to go high. Next, the shop resets the shelf signal. At this point, only one of the two shelf signals is high. Therefore, the assignment will only result in a change in one of the two signals. Therefore, we use the *vacuous assign (vassign)* procedure, as defined below.

```
if (s /= v) then
  s <= v after delay(l,u);
  wait until s = v;
end if;
```

The *vassign* procedure also allows parallel assignments as defined below.

```
if (s1 /= v1) then
  s1 <= v1 after delay(l1,u1);
end if;
if (s2 /= v2) then
  s2 <= v2 after delay(l2,u2);
end if;
if (s1 /= v1) or (s2 /= v2) then
  wait until s1 = v1 and s2 = v2;
end if;
```

After the shelf signals are reset, the shop waits until both bottle signals are low. This is done using the *guard_and* procedure. The procedure *guard_and*($s1$,$v1$,$s2$,$v2$,...) takes a set of signals (i.e., $s1$, $s2$, ...) and a set of values (i.e., $v1$, $v2$, ...), and it stalls a process until each signal s_i has taken value v_i. In other words, it replaces the following code:

```
if ((s1 /= v1) or (s2 /= v2) ...  ) then
  wait until s1 = v1 and s2 = v2 ...;
end if;
```

Finally, the shop protocol assigns *ack_wine* to '0' and loops back to the beginning.

3.2 ACTIVE AND PASSIVE PORTS

A channel connects two processes for communication. The point of connection to each process is called a *port.* For each channel, one port must be *active* while the other is *passive.* The process connected to the *active* port initiates each communication on the channel. The process connected to the *passive* port must patiently wait for communication to be initiated. The first step in translation from a channel-level specification to a handshaking-level specification is to assign the *active* and *passive* side of each channel. The decision can be annotated by replacing the function call *init_channel* with the call *active* or *passive* within the entity. Although this does not change the simulation behavior, it documents the decision. If we have decided to make the connection with the winery passive and the connection with the patron active, we would change the entity for the *shop* as follows:

```
entity shopPA is
  port(wine_delivery:inout channel:=passive;
       wine_selling:inout channel:=active);
end shopPA;
```

One must always assign exactly one side to be passive and the other side to be active. If the *probe* function is not used by either process that communicates on the channel, the choice of which port to make passive is arbitrary. If, on the other hand, the *probe* function is used in one of the processes, the process using the *probe* function must be connected to the passive port. It is illegal for the *probe* function to be used on both sides of a channel, as this would imply that both sides of the channel wait passively for communication.

3.3 HANDSHAKING EXPANSION

The next step of the compilation process is to introduce signal wires that are to be used to implement the channel communication. We first consider simple *bundled data* communication and later introduce methods of encoding the data. Synchronization of a bundled data communication is typically achieved using two wires: one for *requests* and another for *acknowledgments.* The *request* wire is controlled by the *active* side of the communication, and the *acknowledge* wire is controlled by the *passive* side.

For the passive/active *shop*, we would introduce the wires *req_wine* and *ack_wine* between the winery and the shop. We would also introduce the wires *req_patron* and *ack_patron* between the shop and the patron. There is also data that must be transmitted between the processes, namely the bottle of wine which is transmitted on the *bottle* and *shelf* wires. The bottle of wine can be one of eight different types, which means that it would require a minimum of 3 bits to encode the type. The bundled data method can use this minimum encoding. The channels after handshaking expansion are shown

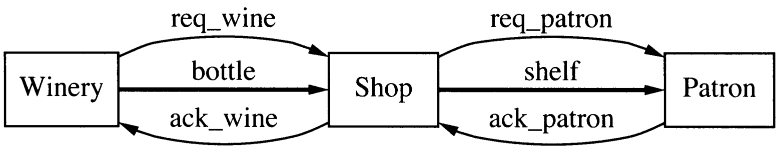

Fig. 3.1 Handshaking expansion for passive/active *shop* using bundled-data.

in Figure 3.1. The *bottle* is transmitted using a 3-bit *std_logic_vector* in the bundled data protocol. One possible description of the active *winery* process after handshaking expansion is shown below.

```
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.handshake.all;
entity winery_bundled is
  port(req_wine:buffer std_logic:='0';
       ack_wine:in std_logic;
       bottle:buffer std_logic_vector(2 downto 0):="000");
end winery_bundled;
architecture two_phase of winery_bundled is
begin
winery_bundled_2phase:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  assign(req_wine,not req_wine,1,3); -- call shop
  guard(ack_wine,req_wine);          -- wine delivered
end process;
end two_phase;
```

The channel in the entity has been replaced by a request, *req_wine*, and acknowledge, *ack_wine*, wire of type *std_logic*, and a 3-bit *std_logic_vector*, *bottle*, to carry the type of wine. As in the channel-level specification, the first two lines of the process randomly select a type of wine and wait a random delay of 5 to 10 minutes before delivering it to the shop. With bundled data, it is necessary that the data be applied to the data lines before the request signal is asserted. In other words, the *shop* needs to be able to safely assume that when it is requested to take that bottle of wine, there is valid data sitting at its inputs when it sees *req_wine* change. This approach requires a timing assumption called a *bundling constraint*. This assumption basically says that the data must be applied to the data lines before the request is asserted. The bundling constraint is simulated in the model using a wait statement.

The next two lines of the process implement the *send* procedure call from the channel-level model. We initially assume that we have a new bottle of wine to transmit. To indicate this to the shop, the winery changes *req_wine* from '0' to '1'. This is accomplished by the call to *assign*, which sets *req_wine* to its complement. The winery then waits for acknowledgment that the wine has been delivered. This is indicated by observing *ack_wine* change from '0' to '1'. This is accomplished by the call to *guard*, which stalls until *ack_wine* equals *req_wine*. These two signal changes complete the transaction of delivery of wine from the winery to the shop.

The model now loops back around and selects a new type of wine. The second time around, the process changes *req_wine* from '1' to '0', and it then waits for *ack_wine* to go to '0'. These two transitions complete another transaction. Since two transitions are required per transaction, this protocol is known as *two-phase handshaking* or *two-cycle signaling*. This protocol is also called *transition signaling*, since communication transactions happen on each transition of the control signals. Next, let us consider the two-phase handshaking expansion of the passive *patron* shown below.

```
patronP_bundled_2phase:process
begin
  guard(req_patron,not ack_patron);      -- shop calls
  bag <= shelf after delay(2,4);
  wait for delay(5,10);
  assign(ack_patron,not ack_patron,1,3); -- buys wine
end process;
```

The handshaking expansion above implements the *receive* procedure call from the channel-level model. It begins by waiting for a request to receive a new bottle of wine. This is indicated when *req_patron* differs from *ack_patron*. After *req_patron* changes to '1', the patron knows there is stable data on the *shelf* data lines which it can copy into the *bag*. It then must wait for a delay equal to the bundling constraint. At this point, it can acknowledge acceptance of the wine. This is accomplished by toggling the *ack_patron* line. The shop does a *receive* followed by a *send*, and it is expanded in a similar fashion, as shown below.

```
shop_bundled_2phase:process
begin
  guard(req_wine,not ack_wine);          -- winery calls
  shelf <= bottle after delay(2,4);
  wait for delay(5,10);
  assign(ack_wine,not ack_wine,1,3);     -- wine arrives
  assign(req_patron,not req_patron,1,3);-- call patron
  guard(ack_patron,req_patron);          -- patron buys wine
end process;
```

Alternatively, we could have decided to use *four-phase handshaking*, also known as *four-cycle signaling*. The entity remains the same, only the protocol in the architecture changes, as shown below.

```
winery_bundled_4phase:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  assign(req_wine,'1',1,3);    -- call shop
  guard(ack_wine,'1');         -- wine delivered
  assign(req_wine,'0',1,3);    -- reset req_wine
  guard(ack_wine,'0');         -- ack_wine resets
end process;
```

Again, the protocol starts with *req_wine* being changed from '0' to '1', and it waits for *ack_wine* to change from '0' to '1'. These two signal changes again complete the transaction of delivery of wine from the winery to the shop. However, at this point, the *req_wine* and *ack_wine* signals are reset before there can be a new transaction. This protocol is also called *level signaling*, since communication happens only when the request signal is '1'. The four-phase bundled data specification for the passive *patron* is below.

```
patronP_bundled_4phase:process
begin
  guard(req_patron,'1');        -- shop calls
  bag <= shelf after delay(2,4);
  wait for delay(5,10);
  assign(ack_patron,'1',1,3);  -- patron buys wine
  guard(req_patron,'0');        -- req_patron resets
  assign(ack_patron,'0',1,3);  -- reset ack_patron
end process;
```

The *patron* again waits for *req_patron* to go to '1'. This indicates that it is safe to copy the data from the *shelf* into the *bag*. After waiting 5 ns to satisfy the bundling constraint, it then sets *ack_patron* to '1' to acknowledge that it has latched the wine. Then it waits for *req_patron* to reset to '0', and it resets *ack_patron* to '0'. The *shop* process is similar, and it is shown below.

```
shop_bundled_4phase:process
begin
  guard(req_wine,'1');         -- winery calls
  shelf <= bottle after delay(2,4);
  wait for delay(5,10);
  assign(ack_wine,'1',1,3);    -- shop receives wine
  guard(req_wine,'0');         -- req_wine resets
  assign(ack_wine,'0',1,3);    -- reset ack_wine
  assign(req_patron,'1',1,3); -- call patron
  guard(ack_patron,'1');       -- patron buys wine
  assign(req_patron,'0',1,3); -- reset req_patron
  guard(ack_patron,'0');       -- ack_patron resets
end process;
```

Although this protocol may appear to be more complex in that it requires twice as many transitions of signal wires, it often leads to simpler circuitry. This means that when operator delays dominate communication costs, four-phase is better. However, if transmission delays dominate communication costs, two-phase is better.

3.4 RESHUFFLING

Consider the *shop_bundled_4phase* process above. It is not possible to implement this specification directly as a circuit. Initially, all the signal wires are '0', and the circuit is supposed to wait for a call from the winery (i.e, wait for *req_wine* to change to '1'). After the wine has arrived and the signal *req_wine* and *ack_wine* have been reset, the value of the signal wires are again all '0'. At this point, however, the circuit for the shop must call the patron (i.e., *req_patron* is enabled to change to '1'). In other words, when all signal wires are '0', the circuit is in a state of confusion. At this point, should the shop wait for a call from the winery or call the patron?

One way to fix this problem is to *reshuffle* the order of the assignments and guards. Since in the four-phase protocol the setting of the request and acknowledgment signals to '0' are used only to reset the variables, they can be asserted anywhere as long as cyclic order is maintained. In other words, we could move up calling the patron to immediately after the wine arrives. This would change the specification of the shop protocol as follows:

```
Shop_PA_reshuffled:process
begin
  guard(req_wine,'1');          -- winery calls
  shelf <= bottle after delay(2,4);
  wait for delay(5,10);
  assign(ack_wine,'1',1,3);     -- shop receives wine
  assign(req_patron,'1',1,3);   -- call patron
  guard(req_wine,'0');          -- req_wine resets
  assign(ack_wine,'0',1,3);     -- reset ack_wine
  guard(ack_patron,'1');        -- patron buys wine
  assign(req_patron,'0',1,3);   -- reset req_patron
  guard(ack_patron,'0');        -- ack_patron resets
end process;
```

Another interesting reshuffling is to delay the wait on the reset of the acknowledgment for an active communication. Consider the guard on the signal *ack_patron* going to '0'. It is not actually necessary to wait for it to reset until it is necessary to set *req_patron* to '1' again. Therefore, it can be delayed until just before this point, as shown below. The first time the guard on *ack_patron* is encountered, it is already '0', so the guard does nothing. For this reason, this guard is called *vacuous*. This protocol is termed *lazy-active*.

```
Shop_PA_lazy_active:process
begin
  guard(req_wine,'1');          -- winery calls
  shelf <= bottle after delay(2,4);
  wait for delay(5,10);
  assign(ack_wine,'1',1,3);     -- shop receives wine
  guard(ack_patron,'0');        -- ack_patron resets
  assign(req_patron,'1',1,3);   -- call patron
  guard(req_wine,'0');          -- req_wine resets
  assign(ack_wine,'0',1,3);     -- reset ack_wine
  guard(ack_patron,'1');        -- patron buys wine
  assign(req_patron,'0',1,3);   -- reset req_patron
end process;
```

Care must, however, be taken when reshuffling as it can introduce *deadlock*. Consider what would happen if the patron moved in at the winery and the handshaking is reshuffled as shown below.

```
Winery_Patron:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  assign(req_wine,'1',1,3);     -- call shop
  guard(ack_wine,'1');          -- wine delivered
  guard(req_patron,'1');        -- shop calls patron
  bag <= shelf after delay(2,4);
  wait for delay(5,10);
  assign(ack_patron,'1',1,3);   -- patron buys wine
  guard(req_patron,'0');        -- req_patron resets
  assign(ack_patron,'0',1,3);   -- reset ack_patron
  assign(req_wine,'0',1,3);     -- reset req_wine
  guard(ack_wine,'0');          -- ack_wine resets
end process;
```

The following sequence of events would cause the system to deadlock:

$$req\_wine+,\ ack\_wine+,\ req\_patron+,\ ack\_patron+$$

The *shop* is waiting for *req_wine* to go to '0' while the combined *winery/patron* would be waiting for *req_patron* to go to '0'.

3.5 STATE VARIABLE INSERTION

As we saw in Chapter 1, another alternative to solving the state coding problem is to add a *state variable.* Consider again the passive/active shop before reshuffling. Another way to solve the state coding problem is to add a state variable as shown below.

```
Shop_PA_SV:process
begin
  guard(req_wine,'1');          -- winery calls
  shelf <= bottle after delay(2,4);
  wait for delay(5,10);
  assign(ack_wine,'1',1,3);     -- shop receives wine
  assign(x,'1',1,3);            -- set x
  guard(req_wine,'0');          -- req_wine resets
  assign(ack_wine,'0',1,3);     -- reset ack_wine
  assign(req_patron,'1',1,3);   -- call patron
  guard(ack_patron,'1');        -- patron buys wine
  assign(x,'0',1,3);            -- reset x
  assign(req_patron,'0',1,3);   -- reset req_patron
  guard(ack_patron,'0');        -- ack_patron resets
end process;
```

We will see later how to find a good insertion point for state variables.

3.6 DATA ENCODING

The advantages of the bundled data approach described above are that it is simple, it allows the use of standard combinational datapath components, and it still allows adaptivity to changes in operating conditions. The disadvantage is that it does not allow any speedup due to data dependencies. A second approach encodes each data line using two wires, and it is thus called *dual-rail.* One wire is set to '1' to indicate that the data line is '0' while another is set to '1' to indicate that the data line is '1'. If both wires are '0', this indicates there is no valid data on the data lines. Both wires are not allowed to be '1' at the same time. This scheme can require up to twice as many wires (8 wires per channel for the wine shop example as opposed to 5 for bundled data). However, it does allow for data-dependent delay variations.

To simplify the discussion, we first consider encoding a single-bit of data, and later we show how to expand it to multibit channels. Figure 3.2 shows the wine shop example using dual-rail encoded data. Observe that there is no longer any need for explicit request wires. For example, the data wires, *bottle0* and *bottle1*, replace the *req_wine* signal. The *winery* process using a dual-rail data encoding is shown below.

```
winery_dual_rail:process
  variable z:integer;
begin
  z:=selection(2);
  case z is
    when 1 =>
      assign(bottle1,'1',1,3);
    when others =>
      assign(bottle0,'1',1,3);
```

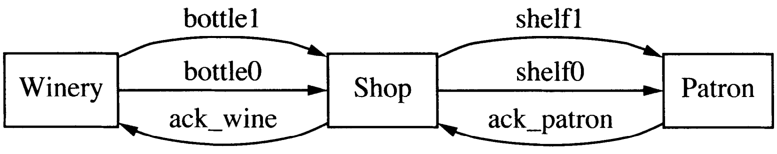

Fig. 3.2 Handshaking expansion for passive/active *wine_shop* example using dual-rail data for a single bit.

```
  end case;
  guard(ack_wine,'1');
  vassign(bottle1,'0',1,3,bottle0,'0',1,3);
  guard(ack_wine,'0');
end process;
```

Initially, both *bottle0* and *bottle1* are low. This indicates that there is no bottle of wine on the channel. The *selection* procedure is used to randomly select between two different types of wine. Depending on the result, either *bottle0* or *bottle1* is set high. The winery then waits for *ack_wine* to go to '1' to indicate that the bottle of wine has been accepted by the shop. It then resets its data wires and waits for *ack_wine* to go to '0'. The processes describing the dual rail protocols for the *shop* and the *patron* are shown below.

```
shopPA_dual_rail:process
begin
  guard(ack_patron,'0');
  guard_or(bottle0,'1',bottle1,'1');
  if bottle0 = '1' then assign(shelf0,'1',1,3);
  elsif bottle1 = '1' then assign(shelf1,'1',1,3);
  end if;
  assign(ack_wine,'1',1,3);
  guard(ack_patron,'1');
  vassign(shelf0,'0',1,3,shelf1,'0',1,3);
  guard_and(bottle0,'0',bottle1,'0');
  assign(ack_wine,'0',1,3);
end process;

patronP_dualrail:process
begin
  guard_or(shelf1,'1',shelf0,'1');
  assign(ack_patron,'1',1,3);
  guard_and(shelf1,'0',shelf0,'0');
  assign(ack_patron,'0',1,3);
end process;
```

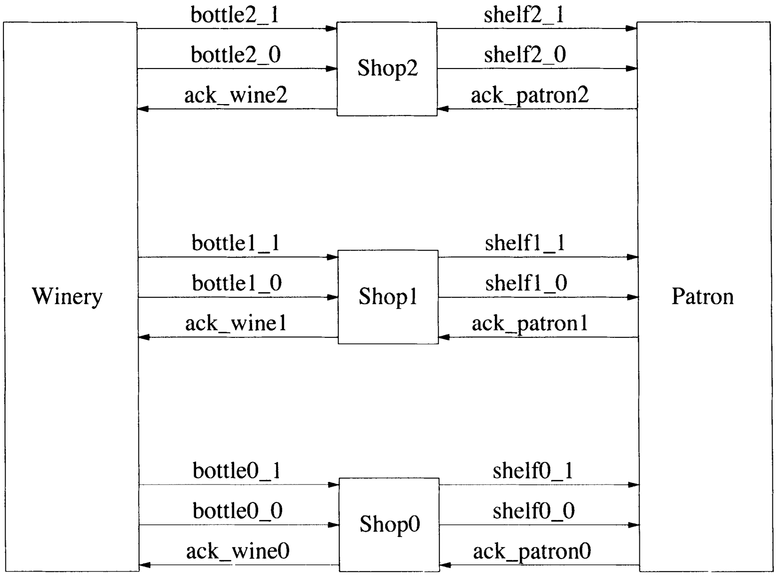

Fig. 3.3 Handshaking expansion for passive/active *wine_shop* example using dual-rail data for 3 bits.

The protocol used for the shop has been reshuffled to produce a better circuit. It starts by making sure that the *ack_patron* signal is '0'. This indicates that the *patron* is currently ready to accept a new bottle of wine. It then waits until the winery sends a bottle of wine (i.e., *bottle0* or *bottle1* is '1'). Once there is a new bottle of wine, it determines which of the two data wires went to '1' and sets the corresponding output data wire to '1' (i.e., if *bottle0* is '1', then *shelf0* should be set to '1'). It then acknowledges that it received the wine from the winery by setting *ack_wine* to '1'. It then waits to receive an acknowledgment that the patron received the wine (i.e., *ack_patron* has gone to '1'). It then resets its data wires. Finally, it waits for the winery to reset its data wires, and it resets the *ack_wine* signal. The patron waits for a *shelf* wire to go high, sets *ack_patron* high, and resets the handshake.

In order to transmit 3 bits of data, you can simply instantiate three copies of the *shop* process and connect them as shown in Figure 3.3. The winery needs to be changed as shown below to send out the three pairs of dual-rail encoded bits. It also must collect three acknowledgments before continuing.

```
winery_dual_rail:process
  variable z:integer;
begin
  z:=selection(8);
  case z is
    when 1 =>
      assign(bottle2_0,'1',1,3,bottle1_0,'1',1,3,
              bottle0_0,'1',1,3);
      when 2 =>
        assign(bottle2_0,'1',1,3,bottle1_0,'1',1,3,
              bottle0_1,'1',1,3);
      when 3 =>
        assign(bottle2_0,'1',1,3,bottle1_1,'1',1,3,
              bottle0_0,'1',1,3);
      when 4 =>
        assign(bottle2_0,'1',1,3,bottle1_1,'1',1,3,
              bottle0_1,'1',1,3);
      when 5 =>
        assign(bottle2_1,'1',1,3,bottle1_0,'1',1,3,
              bottle0_0,'1',1,3);
      when 6 =>
        assign(bottle2_1,'1',1,3,bottle1_0,'1',1,3,
              bottle0_1,'1',1,3);
      when 7 =>
        assign(bottle2_1,'1',1,3,bottle1_1,'1',1,3,
              bottle0_0,'1',1,3);
      when others =>
        assign(bottle2_1,'1',1,3,bottle1_1,'1',1,3,
              bottle0_1,'1',1,3);
    end case;
    guard_and(ack_wine2,'1',ack_wine1,'1',ack_wine0,'1');
    vassign(bottle2_0,'0',1,3,bottle1_0,'0',1,3,
          bottle0_0,'0',1,3);
    vassign(bottle2_1,'0',1,3,bottle1_1,'0',1,3,
          bottle0_1,'0',1,3);
    guard_and(ack_wine2,'0',ack_wine1,'0',ack_wine0,'0');
  end process;
```

The patron must collect data from the three *shop* processes. It also must send out three separate acknowledgments as shown below.

```
  patronP_dualrail:process
  begin
    guard_or(shelf2_1,'1',shelf2_0,'1');
    guard_or(shelf1_1,'1',shelf1_0,'1');
    guard_or(shelf0_1,'1',shelf0_0,'1');
    assign(ack_patron2,'1',1,3,ack_patron1,'1',1,3,
          ack_patron0,'1',1,3);
    guard_and(shelf2_1,'0',shelf2_0,'0');
    guard_and(shelf1_1,'0',shelf1_0,'0');
    guard_and(shelf0_1,'0',shelf0_0,'0');
    assign(ack_patron2,'0',1,3,ack_patron1,'0',1,3,
          ack_patron0,'0',1,3);
  end process;
```

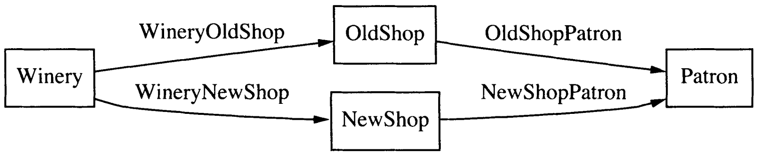

Fig. 3.4 Example with two wine shops.

3.7 EXAMPLE: TWO WINE SHOPS

As another example, let us consider the handshaking expansion of the example from Chapter 2 with two wine shops depicted in Figure 3.4. In this example, the winery randomly decides which shop to deliver its wine to, and the patron probes each shop before deciding which shop to buy wine from. The processes for the *winery*, *shop*, and *patron* are shown below.

```
winery5:process
variable z:integer;
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  z:=selection(2);
  case z is
  when 1 =>
    send(WineryNewShop,bottle);
  when others =>
    send(WineryOldShop,bottle);
  end case;
end process winery5;

shop:process
begin
  receive(WineryShop,shelf);
  send(ShopPatron,shelf);
end process shop;

patron2:process
begin
  if (probe(OldShopPatron)) then
    receive(OldShopPatron,bag);
    wine_drunk <= wine_list'val(conv_integer(bag));
  elsif (probe(NewShopPatron)) then
    receive(NewShopPatron,bag);
    wine_drunk <= wine_list'val(conv_integer(bag));
  end if;
  wait for delay(5,10);
end process patron2;
```

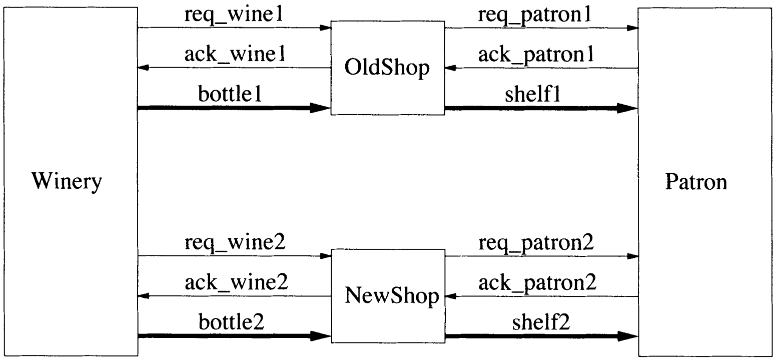

Fig. 3.5 Handshaking expansion with two wine shops.

The first thing to notice is that the patron probes its two channels. Therefore, the patron must be connected to the passive ports of the *OldShopPatron* and *NewShopPatron* channels. This implies that the *shop* must be connected to the active ports. For the *WineryOldShop* and *WineryNewShop* channels, the choice of active and passive ports is arbitrary since neither process probes these channels. Let us connect the winery to the active ports of the *WineryOldShop* port and the *WineryNewShop* port. Assuming bundled data, the block diagram after handshaking expansion is shown in Figure 3.5. The *winery* and *patron* processes after handshaking expansion are shown below. The *shop* process is identical to the one we saw earlier. One key thing to notice is in the *patron* process, the *probe* function calls turn into tests on *req_patron1* and *req_patron2*.

```
winery:process
variable z :  integer;
begin
  z := selection(2);
  bottle <= selection(8,3);
  wait for delay(5,10);
  case z is
    when 1 =>
      bottle1 <= bottle after delay(2,4);
      wait for 5 ns;
      assign(req_wine1,'1',1,3);    -- call winery
      guard(ack_wine1,'1');         -- wine delivered
      assign(req_wine1,'0',1,3);    -- reset req_wine
      guard(ack_wine1,'0');         -- ack_wine resets
    when others =>
      bottle2 <= bottle after delay(2,4);
      wait for 5 ns;
      assign(req_wine2,'1',1,3);    -- call winery
      guard(ack_wine2,'1');         -- wine delivered
      assign(req_wine2,'0',1,3);    -- reset req_wine
      guard(ack_wine2,'0');         -- ack_wine resets
  end case;
end process;

patronP:process
begin
  if (req_patron1 = '1') then
    bag <= shelf1 after delay(2,4);
    wait for delay(5,10);
    assign(ack_patron1,'1',1,3); -- patron buys wine
    guard(req_patron1,'0');      -- req_patron resets
    assign(ack_patron1,'0',1,3); -- reset ack_patron
    wine_drunk <= wine_list'val(conv_integer(bag));
  elsif (req_patron2 = '1') then
    bag <= shelf2 after delay(2,4);
    wait for delay(5,10);
    assign(ack_patron2,'1',1,3); -- patron buys wine
    guard(req_patron2,'0');      -- req_patron resets
    assign(ack_patron2,'0',1,3); -- reset ack_patron
    wine_drunk <= wine_list'val(conv_integer(bag));
  end if;
  wait for delay(1,2);
end process;
```

3.8 SYNTAX-DIRECTED TRANSLATION

We conclude this chapter by describing an asynchronous design procedure where the handshaking expansion step directly determines the circuit implementation. These procedures are based upon the software compiler idea of *syntax-directed translation.* In these procedures, each language construct in the high-level channel-based specification corresponds to a particular circuit structure. These circuit structures are then composed together as dictated by the structure of the program.

Consider the *shop* process, repeated for convenience below.

```
shop:process
begin
  receive(WineryShop,shelf);
  send(ShopPatron,shelf);
end process shop;
```

First, the process statement implies a loop. A loop of the form *while(cond) loop S; end loop* can be translated into the circuit shown in Figure 3.6(a). This circuit operates under the two-phase model. In this circuit, the *req* wire feeds a *merge gate*, *M*. Typically, the merge gate is implemented with an XOR. Assuming that the input coming from the process is initially 0, a change on the *req* wire to 1 causes the input to the *selector module*, SEL, to go to 1. The

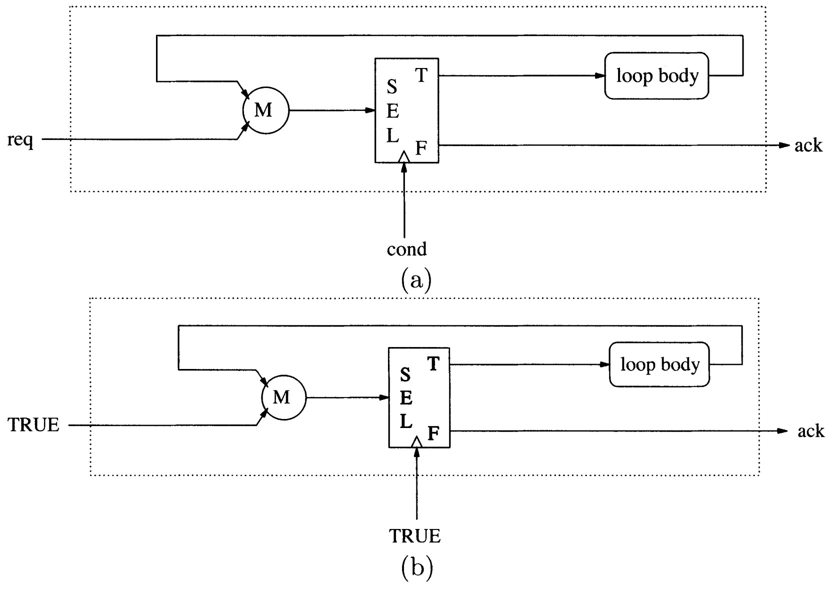

Fig. 3.6 (a) Circuit for looping constructs. (b) Circuit for the *shop* process statement.

SEL module then samples the *cond* wire. If the wire is *true*, it requests the loop to execute. When the loop is done, it acknowledges to the merge gate, which causes another request to the selector. If the condition is false, the entire block acknowledges to the environment that it is done looping. For the process statement, the *req* is *true*, which starts the *shop* operating. A process never terminates, so the condition, *cond*, is set to *TRUE*. The result is shown in Figure 3.6(b).

The next part to consider is the signal *shelf*. The *receive* statement assigns data from the channel *WineryShop* to the signal *shelf*. The circuit to implement the *signal assignment* is shown in Figure 3.7. In this figure, thin lines represent handshake signals and thick lines represent data. When this circuit gets a request, the *enable module* (EN) allows the data on the input lines, *In*, to drive the input bus to the register, *datain*. The EN module then issues a request to the *CALL module*, which makes a request to the register to latch its input data, which will appear on the signal *shelf* when the register latches. The purpose of the CALL module is to coordinate multiple assignments to a single signal. For example, if a signal has two assignments to it in the specification, it can be implemented as shown in Figure 3.8.

To complete implementation of the *receive* statement, we must now construct a circuit to handle synchronization on the channel. Such a circuit is shown in Figure 3.9. This circuit starts with a request to the CALL module. The CALL module forwards the request to the C-element. When the *winery*

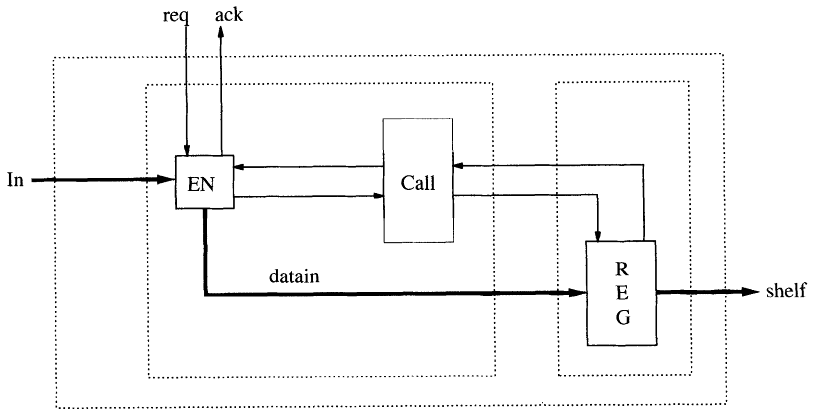

Fig. 3.7 Circuit for assignment to *shelf*.

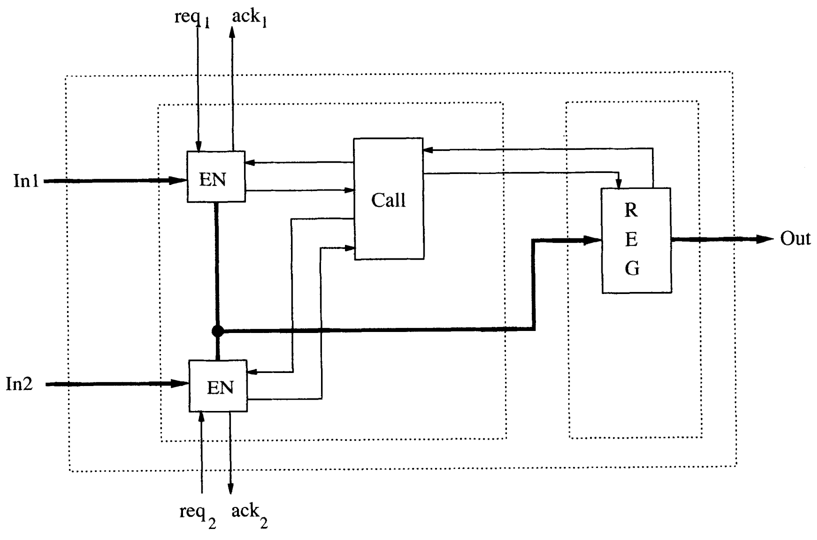

Fig. 3.8 Circuit to implement assignment to a signal from two locations.

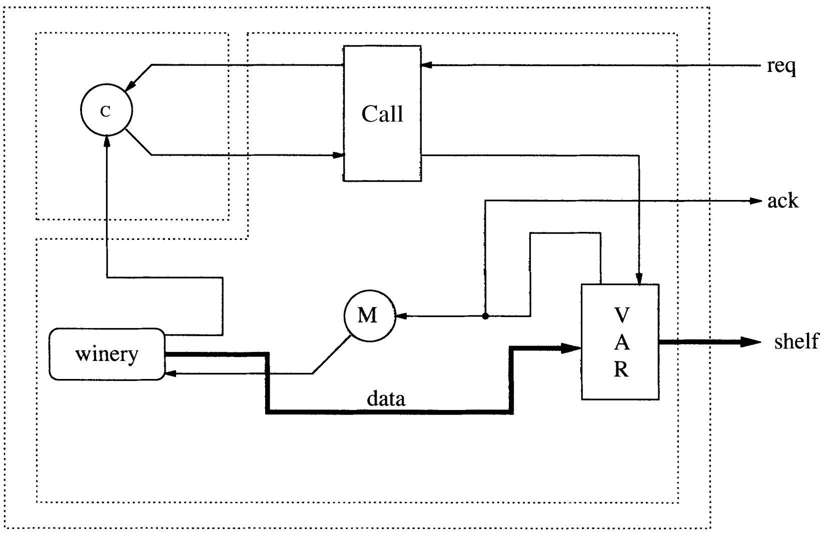

Fig. 3.9 Circuit for a receive statement.

is ready to send data, the other input to the C-element is also asserted. This causes the C-element to generate a transition to the CALL module, which then issues a request for the data sitting on the channel to be latched into the variable (VAR). Note that VAR is a symbol for the circuit shown in Figure 3.7. Once the data has been latched, it is output on the signal *shelf* and the VAR circuit sends an acknowledgment to the unit that requested the communication and also to the *winery* to tell it that the data sent has been latched. Note that the CALL module and the Merge (M) are included because there may be multiple places in the specification where a receive on a given channel may get executed.

The last circuit module needed is one to implement the *send* statement, which is shown in Figure 3.10. This circuit begins with a request to the EN module. This causes the data sitting on the signal *shelf* to be copied to the data wires connected to the *patron*. The request is also forwarded to the CALL module, which forwards it the patron. When the patron acknowledges that the data has been received, the CALL module forwards this acknowledgment back to the EN module. This completes the send, so the block acknowledges its requester.

Although not needed for this circuit, another interesting circuit construct is for selections of the form

```
if (cond1) then
  S1;
elsif (cond2) then
  S2;
```

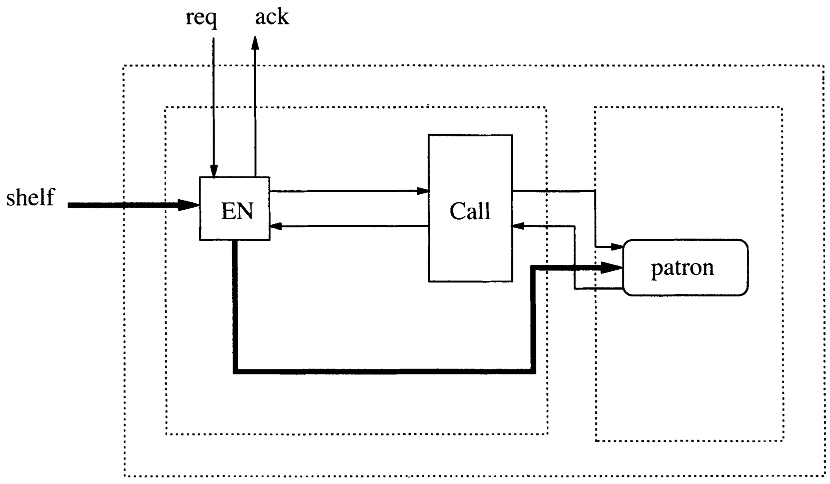

Fig. 3.10 Circuit for a send statement.

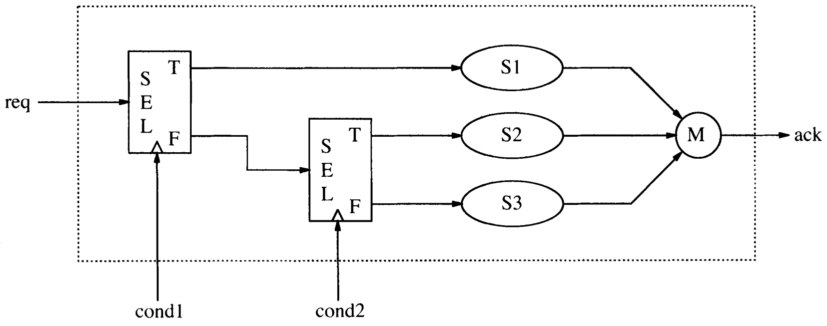

Fig. 3.11 Circuit for a selection statement.

```
else
  S3;
end if;
```

A circuit implementing this type of construct is shown in Figure 3.11. This circuit is started by a transition on the *req* signal. It then samples the *cond1* signal. If this signal is *true*, it starts the operations labeled $S1$. Otherwise, it requests the next *SEL* module to check *cond2*. If this signal is *true*, it starts $S2$. Otherwise, it starts $S3$. As soon as either $S1$, $S2$, or $S3$ is complete, this circuit acknowledges that the selection has completed.

Using these circuit modules, we can now stitch together a complete circuit for the *shop*. If two statements are executed in sequence, they can be connected by taking the acknowledgment from one and using it as the request

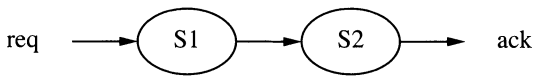

Fig. 3.12 Circuit for sequential composition of two statements.

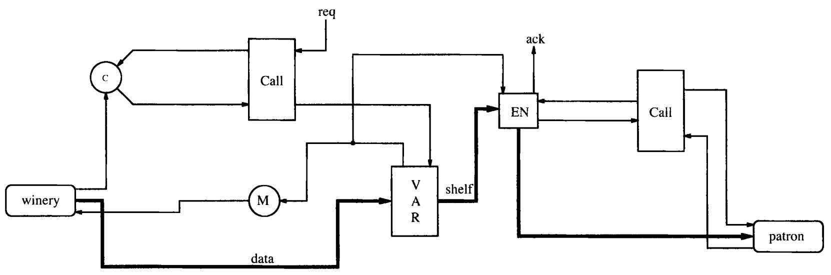

Fig. 3.13 Circuit for a receive followed by a send.

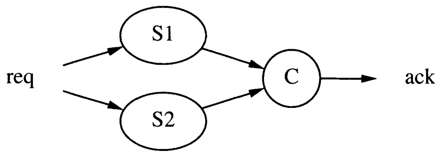

Fig. 3.14 Circuit for parallel composition of two statements.

to the next, as shown in Figure 3.12. For example, the composition of the *receive* and *send* statements from the *shop* yield the circuit shown in Figure 3.13. Although again not needed for this circuit, another useful construct is for parallel composition, and it is shown in Figure 3.14.

Composing the circuit shown in Figure 3.13 with the loop construct and expanding the VAR block yields the complete circuit shown in Figure 3.15. This circuit is clearly more complex than it needs to be. There are several *peephole optimizations* that can be performed to improve this circuit. For example, any *CALL* module that has only a single input request can be removed. After this optimization, the circuit becomes the one shown in Figure 3.16. Another optimization is to eliminate any select modules where the condition is a constant true or false. Also, if a merge only has a single input, it can be removed. The result is shown in Figure 3.17. When a data bus has only a single driver, the EN module can be removed. Finally, since one input of the merge gate is always true, we can replace the merge gate with an inverter. The final simple circuit is shown in Figure 3.18.

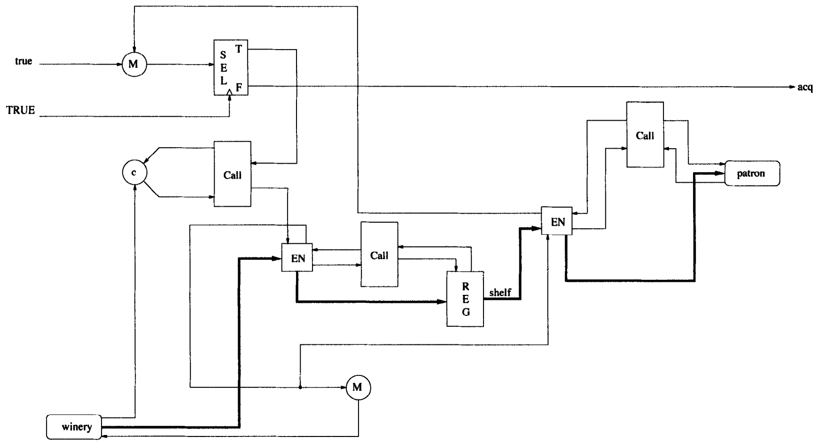

Fig. 3.15 Unoptimized circuit for the *shop*.

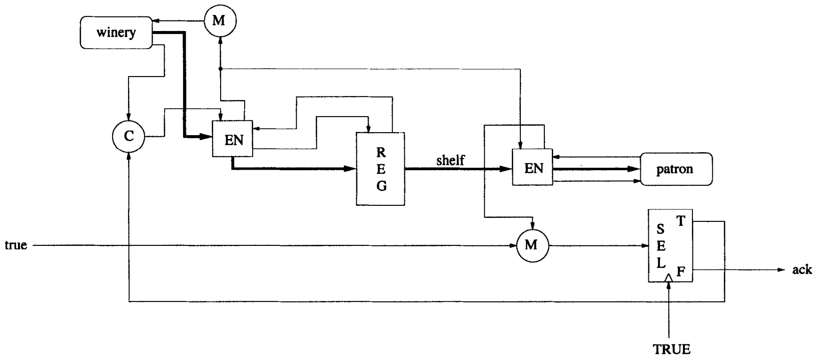

Fig. 3.16 *Shop* circuit after CALL module optimization.

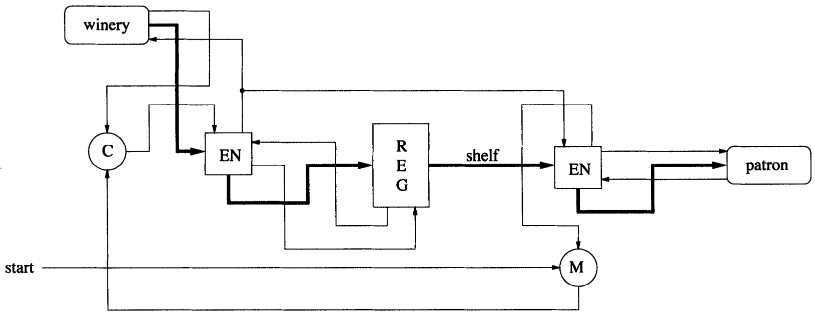

Fig. 3.17 *Shop* circuit after SEL and merge module optimizations.

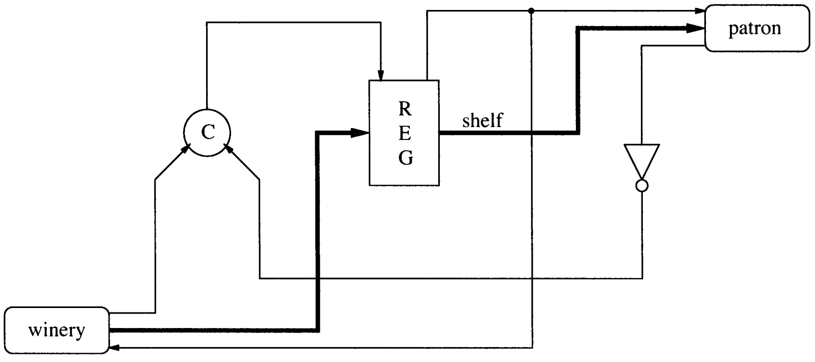

Fig. 3.18 Final circuit for the *shop*.

3.9 SOURCES

The translation process described in this chapter from communication channels to handshaking expansions follows Martin's method [254].

Some of the earliest bundled data asynchronous designs were constructed in the macromodule project at the Washington University in St. Louis. In this project, asynchronous design methods were used to build large computation systems out of macromodules [87, 88, 302, 369]. These macromodules operated using the bundled data design approach, where data wires carried with them control signals to indicate data validity. One of the most famous more recent bundled data design methods is *micropipelines*, which was proposed by Sutherland in his Turing Award paper [370]. Improvements in the latch controller for micropipelines have been proposed by several researchers to reduce the control overhead [104, 133, 275, 375]. A semicustom controllable delay element for micropipeline design is proposed by De Gloria and Olivieri [147]. In the STRiP microprocessor architecture, Dean et al. demonstrated that by using different bundling delays for each type of instruction a significant performance gain could be achieved [106]. An extension of this idea to a finer level of granularity was proposed by Nowick et al. in which based upon a quick analysis of the data, a bundled data constraint is chosen [297, 300]. This technique allows for some advantage due to data dependency even in a bundled data design.

The idea of dual-rail encoding dates back to Gilchrist et al.'s "fast" adder design (40-digit addition in $0.21\mu s$ using vacuum tube logic) [139]. Waite generalized these results to arbitrary iterative networks [396]. Armstrong et al. extended the coding scheme to *m/n codes*, where *m* is the *weight* of the code word (number of 1's in it) and *n* is the length of the code word [12]. Blaum and Bruck introduced *skew-tolerent codes*, which permit bits to be transmitted without waiting for acknowledgments, allowing for pipelined communication [43, 44]. They also allow skew in which parts of a second message can be received before the entire first message has arrived. One of the most exciting asynchronous designs which made use of dual-rail encoding is Williams and Horowitz's self-timed floating-point divider [401, 404]. It achieved faster operation than that of a comparable synchronous design since intermediate results did not need to pass through clocked registers. This divider was later used in a commercial SPARC microprocessor [400]. Recently, Singh and Nowick developed a scheme capable of doubling the throughput over Williams pipeline designs by developing a better protocol and achieving faster completion detection [354].

Dean et al. proposed an alternative data encoding scheme called *level encoded two-phase dual rail* (LEDR), which like two-phase does not return data inputs to zero, but like four-phase it uses the signal levels to encode the data values [105].

The TRIMOSBUS proposed by Sutherland et al. detected that data had been transferred to the bus simply by driving the data onto it, checking that the bus had the value it transmitted, and releasing its drive, allowing the staticizer to hold the state [371]. The TRIMOSBUS also used a three-wire protocol which allowed for a one-to-many communication to be completed without risky timing assumptions.

A completely different approach to detecting completion is to monitor current draw from the power supply [107]. When a circuit ceases to draw current, it has completed its operation. This technique has been explored further by other researchers [137, 153, 154, 218, 355].

The syntax-directed translation methods described in Section 3.8 follows the methods proposed by [52, 53]. A similarity can be seen with this work and the macromodule work described earlier in the types of modules used and the way the systems are interconnected. A peephole optimizer for macromodule-based syntax-directed translation appears in [152]. In [273] and [237], translation methods from Petri nets are developed. Nordmann proposed a method that translates from a type of flowchart, and this method was used in the design of the ILLIAC III [296]. Hirayama developed a silicon compiler for asynchronous architectures [165]. Burns and Martin proposed a translation method from CSP [64, 67, 68]. Another technique was proposed by van Berkel and Rem to automatically compile a circuit described using a language called *Tangram* to a *handshake circuit* implementation [36, 37, 40]. Akella and Gopalakrishnan developed a syntax-directed method using a new channel language called *hopCP* [3, 4, 5]. Snepscheut developed a technique to derive circuits from a trace-based language [360]. Ebergen also translated a trace-based language to DI circuits [119, 121]. A similar approach was proposed by Leung and Li [229]. Keller showed in [190] that a library composed of a merge, select, fork, and an arbitrating test-and-set module are sufficient to design any circuit.

Problems

3.1 The patron has decided that he is going to choose which of the two shops to patronize, but first he will give that shop a call to tell them that he is coming. The chosen shop must then call the winery to request a bottle of wine to be delivered. The configuration is again as shown in Figure 3.4 using bundled data. The channel-level models of the *winery*, *shop*, and *patron* are shown below. Derive the handshaking expansion, and simulate until you are convinced that it is correct.

```
winery:process
begin
  if (probe(wine_to_new_shop)) then
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(wine_to_new_shop,bottle);
  elsif (probe(wine_to_old_shop)) then
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(wine_to_old_shop,bottle);
  else
    wait for delay(5,10);
  end if;
end process winery;

shop:process
begin
  if (probe(wine_selling)) then
     receive(wine_delivery,shelf);
     send(wine_selling,shelf);
  end if;
  wait for delay(1,2);
end process shop;

patron:process
variable z:integer;
begin
  z := selection(2);
  if (z = 1) then
    receive(old_shop_to_patron,bag);
    wine_drunk<=wine_type'val(conv_integer(bag));
  else
    receive(new_shop_to_patron,bag);
    wine_drunk<=wine_type'val(conv_integer(bag));
  end if;
  wait for delay(1,2);
end process patron;
```

3.2 Find two legal reshufflings for the handshaking expansion from Problem 3.1. Simulate until you are convinced that they are correct.

3.3 Perform handshaking expansion on the simple fetch process shown below using bundled data.

```
process
begin
  if (probe(increment)) then
    send(pcadd,'1');
    receive(pcadd_result,pc);
    send(imem_address,pc);
    receive(imem_data,instr);
    send(instr_decode,instr);
    receive(increment);
  elsif (probe(jump)) then
    send(pcadd,offset);
    receive(pcadd_result,pc);
    send(imem_address,pc);
    receive(imem_data,instr);
    send(instr_decode,instr);
    receive(jump);
  end if;
end process;
```

3.4 Find four legal reshufflings of the handshaking expansion from Problem 3.3.

3.5 Perform handshaking expansion on your 4-bit adder design from Problem 2.1.1 using bundled data. Also, perform handshaking expansion on your environment processes. Simulate your design with its environment.

3.6 Perform handshaking expansion on your 1-bit adder design from Problem 2.1.2. Use dual-rail for data encoding. Also, perform handshaking expansion on your environment processes. Simulate your design. Using these single-bit blocks, build a 4-bit adder with its environment and simulate.

3.7 Perform handshaking expansion on your stack design from Problem 2.2 using bundled data. Create an environment and simulate a four-stage stack.

3.8 Perform handshaking expansion on your stack design from Problem 2.2 assuming only 1 bit of data encoded using dual-rail. Create an environment and simulate a four-stage stack.

3.9 Perform handshaking expansion on your shifter design from Problem 2.3 using bundled data. Create an environment and simulate a four-element shifter.

3.10 Perform handshaking expansion on your shifter design from Problem 2.3 using dual-rail data encoding. Create an environment and simulate a four-element shifter.

3.11 Perform handshaking expansion on your entropy decoder design from Problem 2.4.1 using bundled data. Create an environment and simulate.

3.12 Perform handshaking expansion on your entropy decoder design from Problem 2.4.2 using dual-rail data encoding. Create an environment and simulate a 4-bit decoder.

3.13 Using syntax-directed translation, find a circuit to implement Euclid's greatest common divisor (gcd) algorithm given below.

```
receive(A,a,B,b);
while (a != b) loop
  if (a > b) then
    a <= a - b;
    wait for 5 ns;
  else
    b <= b - a;
    wait for 5 ns;
  end if;
end loop;
end if;
send(C,a);
```

You may assume that the user is not allowed to send in operands *a* or *b* equal to 0. You can also assume the existence of a comparator which has two input buses, and after a request sets a condition which is true when operand 1 is greater than operand 2. After the condition is stable, an acknowledgment is provided. You may also assume the existence of a subtractor which has two input buses and one output bus. After getting a request, it returns an acknowledgment to indicate a stable result on the output bus.