Asynchronous Circuit Design. Chris J. Myers

ISBNs: 0-471-41543-X (Hardback); 0-471-22414-6 (Electronic)

2

Communication Channels

> Inform all the troops that communications have completely broken down.
>
> —Ashleigh Brilliant

> This "telephone" has too many shortcomings to be seriously considered as a means of communication. The device is inherently of no value to us.
>
> —Western Union internal memo, 1876

In this chapter we describe a method for the specification of asynchronous systems at a high level through the use of a communication channel model. Any hardware system, synchronous or asynchronous, can be thought of as a set of concurrently operating processes. These processes periodically must communicate data between each other. For example, if one process is a register file and another is an ALU, the register file must communicate operands to the ALU, and the ALU must communicate a result back to the register file. In the *channel model*, this communication takes places when one process attempts to send a message along a *channel* while another is attempting to receive a message along the same channel. We have implemented a package in VHDL to provide a channel abstraction. In this chapter we give a brief overview of VHDL and how to use our package. Since it is beyond the scope of this book to teach VHDL, we primarily provide templates that will allow you to use VHDL to simulate asynchronous systems without needing a thorough understanding of the language.

2.1 BASIC STRUCTURE

Each module that you specify using our channel model will have the same basic structure. A channel is simply a point-to-point means of communication between two concurrently operating processes. One process uses that channel to send data to the other process. As an example, the VHDL model for the wine shop example is given below.

```
----------------------
-- wine_example.vhd
----------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity wine_example is
end wine_example;
architecture behavior of wine_example is
  type wine_list is (cabernet, merlot, zinfandel,
                     chardonnay, sauvignon_blanc,
                     pinot_noir, riesling, bubbly);
  signal wine_drunk:wine_list;
  signal WineryShop:channel:=init_channel;
  signal ShopPatron:channel:=init_channel;
  signal bottle:std_logic_vector(2 downto 0):="000";
  signal shelf:std_logic_vector(2 downto 0);
  signal bag:std_logic_vector(2 downto 0);
begin
winery:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  send(WineryShop,bottle);
end process winery;
shop:process
begin
  receive(WineryShop,shelf);
  send(ShopPatron,shelf);
end process shop;
patron:process
begin
  receive(ShopPatron,bag);
  wine_drunk <= wine_list'val(conv_integer(bag));
end process patron;
end behavior;
```

Let's consider each part of the VHDL model separately. First, there is a comment indicating the name of the file. All comments in VHDL begin with "−−" and continue until the end of the line. The next section of code indicates what other libraries and packages are used by this file. These are like include statements in C or C++. You should always use the first line, which states that you want access to the `ieee` library. This a very useful standard library. For example, the next three lines include packages from this library. You probably also always want to use these three lines, as these packages define and implement the `std_logic` data type and arithmetic operations on them. You will use this data type a lot. The last two lines include the packages `nondeterminism` and `channel`, which can be found in Appendix A. The nondeterminism package defines some functions to generate random delays and random selections for simulation. The channel package includes a definition of the channel data type and operations on it, such as *send* and *receive*.

The next part is called the *entity*. The entity describes the interface of the module being designed. In this case, it defines the *wine_example*. In this example we are designing a *closed system* (i.e., no inputs or outputs), so our entity is very simple. We will introduce the syntax for more complex entity descriptions when we get to structural VHDL below.

The next part is called the *architecture*. This is used to define the behavior of the entity being designed. For each entity, there can be multiple architectures describing either different implementations or the same implementation at different levels of abstraction. The name of the architecture is identified in the first line (i.e., *behavior*) and the entity it corresponds to (i.e., *wine_example*).

The architecture has two parts: the *declaration section* and the *concurrent statement section*. In the declaration section, we can define *types*, *signals*, or, as we will see later, *components*. For example, we have defined an *enumerated type* called *wine_list*, which is a list of the different types of wine produced at the winery. We can then declare a signal *wine_drunk* of this type. A signal represents a wire (or possibly a collection of wires) in a design.

For this example, we have also defined two channels for communication. The *WineryShop* channel is used for delivering bottles of wine to the shop and the *ShopPatron* channel is used for selling bottles of wine to the patron. Both channels are initialized in the declaration section by assignment to the return value of the function *init_channel*.

The next three signals are used by the winery, shop, and patron to keep track of what type of wine they have. These signals are of type *std_logic_vector*. The type *std_logic* is a nine-valued enumerated type where the values represent various states of a signal wire (see Table 2.1). In addition to the strong logic values '0' and '1', there is a strong unknown value 'X', which indicates that multiple drivers are forcing the wire to opposite values. There are also weak values which can be overpowered by strong values. The last three indicate a wire that is uninitialized, 'U'; high impedance,'Z'; or don't care, '−'. The table also shows the resolution of the value if two modules drive the same wire

Table 2.1 Std_logic values.

	Meaning	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'-'
'U'	Uninitialized	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'	'U'
'X'	Forcing unknown	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'
'0'	Forcing 0	'U'	'X'	'0'	'X'	'0'	'0'	'0'	'0'	'X'
'1'	Forcing 1	'U'	'X'	'X'	'1'	'1'	'1'	'1'	'1'	'X'
'Z'	High impedance	'U'	'X'	'0'	'1'	'Z'	'W'	'L'	'H'	'X'
'W'	Weak unknown	'U'	'X'	'0'	'1'	'W'	'W'	'W'	'W'	'X'
'L'	Weak 0	'U'	'X'	'0'	'1'	'L'	'W'	'L'	'W'	'X'
'H'	Weak 1	'U'	'X'	'0'	'1'	'H'	'W'	'W'	'H'	'X'
'-'	Don't care	'U'	'X'	'X'	'X'	'X'	'X'	'X'	'X'	'X'

to different values. A *std_logic_vector* is an array of *std_logic* signals. We are encoding the type of wine using 3-bit-wide *std_logic_vectors*. The bottle going out of the winery is initialized to "000", which corresponds to the 0th type of wine (i.e., a bottle of cabernet).

The concurrent statement section starts with the *begin* statement, and it is where the behavior of the module is defined. The only concurrent statement which we introduce at this time is the *process statement*. In this example, there are three processes which operate concurrently. There is one for the winery, one for the shop, and another for the patron. Each process statement begins with an optional label and the keyword *process*. The statements within the process are executed sequentially.

The behavior of the *winery* begins by randomly selecting a type of wine to produce using the *selection* function. This function is defined in the *nondeterminism* package, and it takes two integer parameters. The first is the number of choices, and the second is the size of the *std_logic_vector* to return. The next step is that the winery waits for some random time between 5 and 10 minutes until it is ready to transmit another bottle of wine. This is accomplished using the *delay* function which is also defined in the *nondeterminism* package. This function takes two integer parameters which set the lower and upper bound for the range of possible delay values. Finally, the winery sends its bottle to the shop with the procedure call *send*. This procedure has two parameters: a channel to communicate on and a *std_logic* or *std_logic_vector* containing the data to be transmitted. The *send* procedure is defined in the channel package. In this example, it will wait until the shop is ready to receive the wine and then delivers it.

The behavior of the *shop* begins by receiving a bottle of wine from the winery with the procedure call *receive*. This procedure also has two parameters: a channel to communicate on and a *std_logic* or *std_logic_vector* where the data is to be copied upon reception. The *receive* procedure is also defined

in the channel package. In this example, it waits until the winery has a bottle of wine to deliver; then the shop accepts delivery of it. After receiving the wine, the shop sends it to the patron over the *ShopPatron* channel.

The behavior of the *patron* begins by receiving a bottle of wine which it then identifies (probably with a small sip as the winery does not label its wine). To do this in VHDL, the *bag*, which is a *std_logic_vector*, must first be converted into an integer using the *conv_integer* function. This integer is used as an index to the function *wine_list*'**val**. As mentioned before, *wine_list* is an enumerated type, and **'val** is an *attribute*. When the **'val** attribute is combined with an enumerated type, it produces a function which takes an integer and returns the value of that position in the enumerated type. For example, 0 would return cabernet and 1 would return merlot.

2.2 STRUCTURAL MODELING IN VHDL

Whereas we could model any system using a single entity/architecture pair, it would get quite cumbersome for large designs. This is especially true when many copies of the same process are needed. In this section we introduce a way of specifying separate processes within different entity/architecture pairs and then composing them together using a top-level structural architecture. Let's begin by looking at the entity/architecture pair which specifies just the behavior of the shop.

```
-- shop.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;
entity shop is
  port(wine_delivery:inout channel:=init_channel;
       wine_selling:inout channel:=init_channel);
end shop;
architecture behavior of shop is
  signal shelf:std_logic_vector(2 downto 0);
begin
shop:process
begin
  receive(wine_delivery,shelf);
  send(wine_selling,shelf);
end process shop;
end behavior;
```

The key difference between this model and the previous one is seen in the *port declarations* in the entity. As mentioned before, the entity is used to specify the interface of a module. Considering the shop separately, there are now two channels at the interface which are declared as ports in the

entity. Each port declaration is given a unique name. In this example, the ports are *wine_delivery* and *wine_selling*. Each port is also given a *mode* setting the direction of the data flow. The modes can be *in*, *out*, or *inout*. In this example, the channels are given a mode of *inout*. Although it may appear that *wine_delivery* should be *in* and *wine_selling* should be *out*, they are actually *inout* because while data flows only one way, the control for the communication flows in both directions. The next part of the port declaration is the port types. In our example, both ports are of type *channel*. Finally, each port declaration is allowed an optional initialization. In this case, both channels are initialized with calls to the function *init_channel*. The only other differences are that the architecture is linked to the entity *shop* and includes only the signal and process for the *shop*. The entity/architecture pairs for the *winery*, the *patron*, and the composition with the shop are given below.

```
-- winery.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;
entity winery is
  port(wine_shipping:inout channel:=init_channel);
end winery;
architecture behavior of winery is
  signal bottle:std_logic_vector(2 downto 0):="000";
begin
winery:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  send(wine_shipping,bottle);
end process winery;
end behavior;

-- patron.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity patron is
  port(wine_buying:inout channel:=init_channel);
end patron;
architecture behavior of patron is
  type wine_list is (cabernet, merlot, zinfandel,
                     chardonnay, sauvignon_blanc,
                     pinot_noir, riesling, bubbly);
  signal wine_drunk:wine_list;
  signal bag:std_logic_vector(2 downto 0);
```

```
begin
patron:process
begin
  receive(wine_buying,bag);
  wine_drunk <= wine_list'val(conv_integer(bag));
end process patron;
end behavior;

-- wine_example2.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;
entity wine_example is
end wine_example;
architecture structure of wine_example is
  component winery
    port(wine_shipping:inout channel);
  end component;
  component shop
    port(wine_delivery:inout channel;
         wine_selling:inout channel);
  end component;
  component patron
    port(wine_buying:inout channel);
  end component;
  signal WineryShop:channel:=init_channel;
  signal ShopPatron:channel:=init_channel;
begin
  THE_WINERY:winery
    port map(wine_shipping => WineryShop);
  THE_SHOP:shop
    port map(wine_delivery => WineryShop,
             wine_selling => ShopPatron);
  THE_PATRON:patron
    port map(wine_buying => ShopPatron);
end structure;
```

The descriptions of the *winery* and *patron* are similar to the one for the *shop*. The last entity/architecture pair represents an alternative structural architecture for the *wine_example*. Within the structural architecture, declarations are given for each of the components. Component declarations are forward declarations of existing (but defined elsewhere) entities to define the ports to be used in the instantiations. These can be thought of as being like function prototypes in C or C++. Next, two channels are declared to connect the *winery* to the *shop* and from the *shop* to the *patron*.

The concurrent statement part includes three component instantiations. Each begins with a label for that instance of the component. In this example, the label for the winery is *THE_WINERY*. Following the label is the name

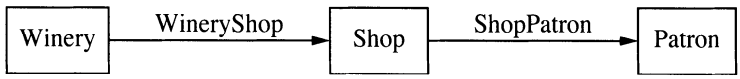

Fig. 2.1 Block diagram for *wine_example*.

of the entity for the component being instantiated. The last part is the *port map*. The port map is used to indicate which wires within a component are connected to which wires at the top level. For example, the *wine_shipping* port within the winery is connected to the *WineryShop* port at the top level. In the instantiation for the *shop*, *wine_delivery* is also connected to the *WineryShop* port, which in effect connects the *winery* to the *shop*. The rest of the instantiations for the *shop* and the *patron* are similar. The block diagram showing the connections between the components is shown in Figure 2.1.

As another example, consider a new shop opening closer to the patron. The patron now buys his wine from the new shop. Due to contracts with the winery, the new shop must buy its wine from the original shop. The new block diagram is shown in Figure 2.2, and the new architecture is shown below.

```
architecture new_structure of wine_example is
  component winery
    port(wine_shipping:inout channel);
  end component;
  component shop
    port(wine_delivery:inout channel;
         wine_selling:inout channel);
  end component;
  component patron
    port(wine_buying:inout channel);
  end component;
  signal WineryShop:channel:=init_channel;
  signal ShopNewShop:channel:=init_channel;
  signal NewShopPatron:channel:=init_channel;
begin
  THE_WINERY:winery
    port map(wine_shipping => WineryShop);
  OLD_SHOP:shop
    port map(wine_delivery => WineryShop,
             wine_selling => ShopNewShop);
  NEW_SHOP:shop
    port map(wine_delivery => ShopNewShop,
             wine_selling => NewShopPatron);
  THE_PATRON:patron
    port map(wine_buying => NewShopPatron);
end new_structure;
```

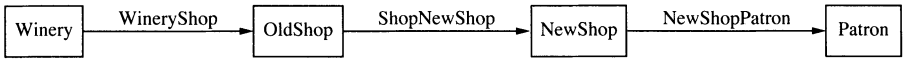

Fig. 2.2 Block diagram including the new shop.

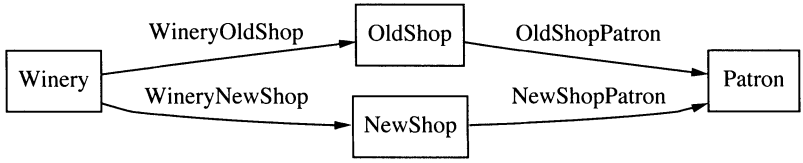

Fig. 2.3 Direct communication between winery and the new shop.

2.3 CONTROL STRUCTURES

So far, the flow of control in all our examples has been very simple in that no choices are being made at any point. In this section we introduce control structures for *selection* and *repetition*. We use as an example the scenario where the winery and patron can deal with either shop. The communication channels are depicted in Figure 2.3.

2.3.1 Selection

There are two ways in VHDL to model selection: **if-then-else** statements and **case** statements. As an example, let's assume that the winery has entered into an agreement to sell its merlot to the new shop. All other types of wine will still go to the old shop. This can be written in two ways:

```
winery2:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  if (wine_list'val(conv_integer(bottle)) = merlot) then
    send(WineryNewShop,bottle);
  else
    send(WineryOldShop,bottle);
  end if;
end process winery2;

winery3:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  case (wine_list'val(conv_integer(bottle))) is
  when merlot =>
    send(WineryNewShop,bottle);
  when others =>
```

```
      send(WineryOldShop,bottle);
    end case;
  end process winery3;
```

Note that in the case statement, **others** is a keyword to indicate any other type of wine.

The previous selection mechanism is *deterministic* in that the action is completely determined by the type of wine being produced. It is often also useful to be able to model a *nondeterministic*, or random, selection. For example, if the winery has decided to choose randomly which shop to sell to each time, it would be written in one of these two ways:

```
winery4:process
variable z:integer;
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  z:=selection(2);
  if (z = 1) then
    send(WineryNewShop,bottle);
  else
    send(WineryOldShop,bottle);
  end if;
end process winery4;

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;
```

In the examples above, the function *selection* is overloaded in the *nondeterminism* package, to take one parameter, a constant indicating the range desired, and returns a random integer result. Since the selection statement is sent the constant 2, it returns either a 1 or a 2. Note that z is an integer variable that is local only to this process.

2.3.2 Repetition

Every process repeats forever, so it implies a loop. If you want to specify a loop internal to a process, there are three different types of looping constructs.

The first type of loop construct that we consider is the **for** loop. Just as in conventional programming languages, the **for** loop allows you to loop a fixed number of times. For example, if the winery decided to send the first four bottles of wine to the old shop and the next three to the new shop, it would be specified like this (note that the loop index variable i is declared implicitly):

```
winery6:process
begin
  for i in 1 to 4 loop
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryOldShop,bottle);
  end loop;
  for i in 1 to 3 loop
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryNewShop,bottle);
  end loop;
end process winery6;
```

The next type of loop construct is the **while** loop. Again, its usage should be familiar. The **while** loop allows you to specify behavior that loops until a boolean condition is satisfied. For example, if the winery decided to send out bottles of wine to the old shop until it had a bottle of merlot to send, and then it would send the merlot to the new shop, it could be specified like this:

```
winery7:process
begin
  while (wine_list'val(conv_integer(bottle)) /= merlot)
  loop
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryOldShop,bottle);
  end loop;
  bottle <= selection(8,3);
  wait for delay(5,10);
  send(WineryNewShop,bottle);
end process winery7;
```

The last kind of looping construct is the infinite **loop**. For example, consider the case where the winery has decided to stop doing business with the old shop and deal only with the new shop, but due to existing contract obligations it still owed one final bottle to the old shop. This could be specified as follows:

```
winery8:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  send(WineryOldShop,bottle);
  loop
```

```
    bottle <= selection(8,3);
    wait for delay(5,10);
    send(WineryNewShop,bottle);
  end loop;
end process winery8;
```

2.4 DEADLOCK

An additional complication that arises with asynchronous design is the potential of introducing a *deadlock* into the system. Deadlock is the state in which a system can no longer make progress toward a required goal. For example, consider the two processes below:

```
producer:process
begin
  send(X,x);
  send(Y,y);
end process producer;

consumer:process
begin
  receive(Y,a);
  receive(X,b);
end process consumer;
```

The producer tries to send x out on channel X while the consumer is trying to receive data on channel Y. They both sit waiting for the other to respond and no progress can be made. In other words, the system is deadlocked. Although this is obvious in this example, such situations are difficult to observe in larger examples.

Let us again consider the wine shop example. In the preceding section we described several different ways the winery can decide to which shop to send the wine. The patron now could go to two different shops to get his wine. The problem that the patron faces now is how to decide which shop has wine. The simplest approach would be to patronize just one shop. This approach, however, can cause the system to deadlock. Consider the case where the winery decides to send its next two bottles of wine to the new shop and the patron patronizes only the old shop. In this case, the winery delivers the first bottle to the new shop, where it is put on their shelf. When they are ready to deliver the second bottle, the new shop is unable to accept it because it still has the first bottle on its shelf. Assuming that the winery is persistent, no more wine will be produced. Another idea might be for the patron to alternate between the two shops. Assuming that the patron begins by going to the old shop, and the winery sends the first two bottles to the new shop, the same problem arises.

2.5 PROBE

To solve this deadlock problem, the patron needs to know who has got wine to sell before he commits to shopping at one particular shop. This is accomplished by adding *probe* to the channel package. The *probe* function takes a channel as a parameter and returns true if there is a pending communication on that channel, and otherwise, returns false. Using this function, the patron can first check if the old shop is trying to sell him wine, and if so, receive it. If not, he can check the new shop. If neither has wine to sell, he will wait for 5 to 10 minutes, and start checking again (he's not very patient). An important subtle note is that the wait statement is essential; otherwise, the simulator will go into an infinite loop. Within every process in VHDL, there must be either a *wait statement* or a *sensitivity list.* Without either one, the process will execute forever and starve all other processes. Note that there is an implicit wait within a *send* or *receive* procedure call. If neither condition is true, a *wait* statement must be inserted to cause the simulation to stop working on this process and give another one a chance to make progress.

```
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;
```

2.6 PARALLEL COMMUNICATION

So far all communication actions have been initiated sequentially. To improve speed, it is often beneficial to launch several communication actions in parallel. Our channel package supports this by allowing you to pass multiple channel/data pairs to the *send* or *receive* procedures. As an example, consider the case where the winery makes two bottles of wine at a time and sends them out concurrently to the two shops. This is described as follows:

```
winery9:process
begin
  bottle1 <= selection(8,3);
  bottle2 <= selection(8,3);
  wait for delay(5,10);
  send(WineryOldShop,bottle1,WineryNewShop,bottle2);
end process winery9;
```

We could leave the patron process as described in the preceding section. However, to improve performance, we may want to allow the patron to receive the bottles in parallel (perhaps, he hired someone to run to the other shop for him). This can be described as follows:

```
patron3:process
begin
  receive(OldShopPatron,bag1,NewShopPatron,bag2);
  wine_drunk1 <= wine_list'val(conv_integer(bag1));
  wine_drunk2 <= wine_list'val(conv_integer(bag2));
end process patron3;
```

2.7 EXAMPLE: MINIMIPS

Translating a word description for a design into a good formal specification can be more of an art form than a science. Since word descriptions are, by their very nature, imprecise and unstructured, there can be no set process to a formal specification. Therefore, the best way to learn how to do this is through an example. This section shows the derivation of a specification from a word description for a simple MiniMIPS microprocessor and some initial steps toward its optimization.

The MiniMIPS is a simple microprocessor described in [164]. The MiniMIPS has a simple reduced instruction set computer (RISC)-style architecture and datapath. The design presented here is indeed simple in that it only supports the eight different types of instructions shown in Table 2.2. All instructions are 32 bits wide and come in one of three formats, shown in Figure 2.4.

To specify the MiniMIPS, we have decided to decompose the design into five communicating processes, as shown in Figure 2.5. Two describe the environment: an instruction memory (*imem*) and a data memory (*dmem*). The other three describe the circuit to be designed. Consider arithmetic and logic operations. The first stage fetches instructions from the instruction memory (*fetch*). The second stage takes these instructions, decodes them, and fetches values from the appropriate registers (*decode*). The third stage takes these register values, executes the appropriate function on them, and returns the result to the register file in the decode block (*execute*). If the instruction is a *load*, the *execute* block would generate an address for the data memory which would send the result back to the decode block for storage. If the instruction is a *store*, the *decode* block would send the data to be stored to the data memory while the execute block sends the address. If the instruction is a *branch*, the *execute* block is responsible for sending the result of the comparison back to the *fetch* block, which then uses this to determine the next instruction to fetch. Finally, if the instruction is an unconditional *jump*, the *fetch* block simply adjusts the program counter (PC).

Table 2.2 MiniMIPS instruction set.

Instruction	Opcode	Function	Operation	Example
add	0	32	rd := rs + rt	add r1, r2, r3
sub	0	34	rd := rs − rt	sub r1, r2, r3
and	0	36	rd := rs & rt	and r1, r2, r3
or	0	37	rd := rs | rt	or r1, r2, r3
lw	35	n/a	rt := mem[rs + offset]	lw r1, (32)r2
sw	43	n/a	mem[rs + offset] := rt	sw r1, (32)r2
beq	4	n/a	if (rs==rt) then PC := PC + offset	beq r1, r2, Loop
j	6	n/a	PC := address	j Loop

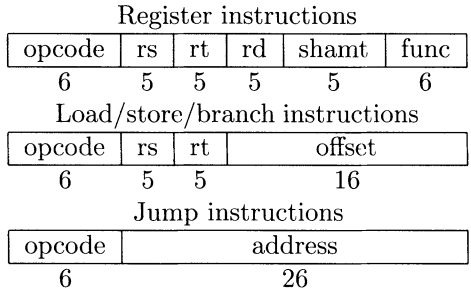

Fig. 2.4 Instruction formats for MiniMIPS.

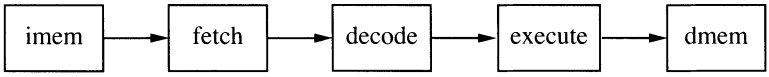

Fig. 2.5 Block diagram for MiniMIPS.

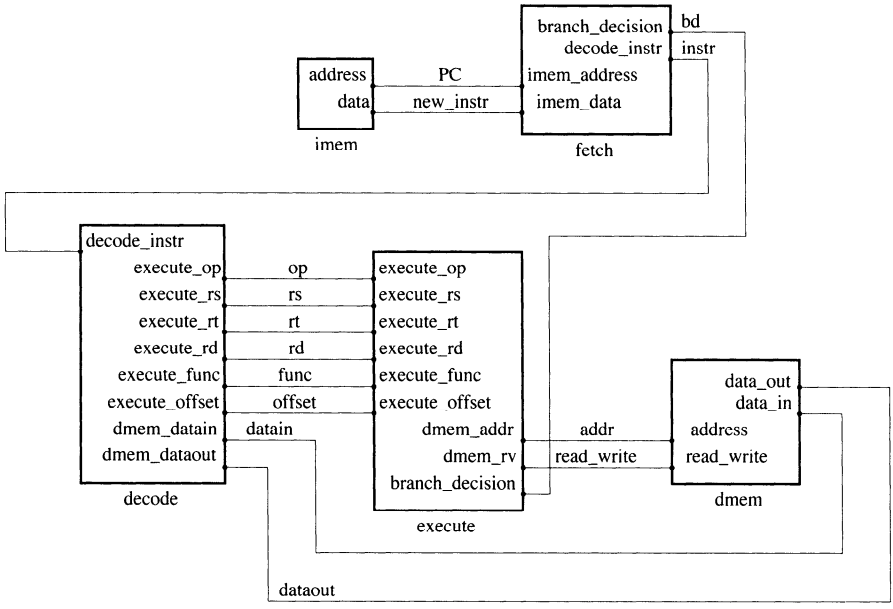

Fig. 2.6 Channel-level diagram for MiniMIPS.

2.7.1 VHDL Specification

The first step in the design process is to determine the needed communication channels between the blocks and draw a block diagram. The result for the MiniMIPS is shown in Figure 2.6. Beginning with the communications between the *imem* and *fetch* blocks, the *fetch* block needs the channel *PC* to send the PC to the *imem* block, and the *imem* block needs the channel *new_instr* to send the new instruction back to the *fetch* block. The *fetch* block also needs the channel *instr* to send the new instruction to the *decode* block, and the channel *bd* from the *execute* block to notify it whether or not a branch is to be taken. The *decode* block parses the instruction and needs many channels to send the *opcode* (*op* channel), ALU function code (*func* channel), source operands (*rs* and *rt* channels), and offset (*offset* channel) to the *execute* block. Note that not all channels are used for every instruction. The *decode* block also needs the channel *rd* from the *execute* block to get the result of arithmetic/logic operations, and the channels *datain* and *dataout* from and to the *dmem* block for receiving and sending data for loads and stores. Finally, the *execute* block needs the channels *addr* and *read_write* to the *dmem* block to send the address and read/write signals. The structural VHDL code connecting the blocks together is shown below.

```
--------------------
-- minimips.vhd
--------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.channel.all;
entity minimips is
end minimips;
architecture structure of minimips is
  component imem
    port(data:inout channel;
         address:inout channel);
  end component;
  component fetch
    port(branch_decision:inout channel;
         decode_instr:inout channel;
         imem_data:inout channel;
         imem_address:inout channel);
  end component;
  component decode
    port(dmem_dataout:inout channel;
         dmem_datain:inout channel;
         execute_offset:inout channel;
         execute_func:inout channel;
         execute_rd:inout channel;
         execute_rt:inout channel;
         execute_rs:inout channel;
         execute_op:inout channel;
         decode_instr:inout channel);
  end component;
  component execute
    port(branch_decision:inout channel;
         dmem_rw:inout channel;
         dmem_addr:inout channel;
         execute_offset:inout channel;
         execute_func:inout channel;
         execute_rd:inout channel;
         execute_rt:inout channel;
         execute_rs:inout channel;
         execute_op:inout channel);
  end component;
  component dmem
    port(read_write:inout channel;
         data_out:inout channel;
         data_in:inout channel;
         address:inout channel);
  end component;
  signal addr:channel;
  signal bd:channel;
```

```
  signal datain:channel;
  signal dataout:channel;
  signal func:channel;
  signal instr:channel;
  signal new_instr:channel;
  signal offset:channel;
  signal op:channel;
  signal PC:channel;
  signal rd:channel;
  signal read_write:channel;
  signal rs:channel;
  signal rt:channel;
begin
  imem1:imem
    port map(data => new_instr,
             address => PC);
  fetch1:fetch
    port map(branch_decision => bd,
             decode_instr => instr,
             imem_data => new_instr,
             imem_address => PC);
  decode1:decode
    port map(dmem_dataout => dataout,
             dmem_datain => datain,
             execute_offset => offset,
             execute_func => func,
             execute_rd => rd,
             execute_rt => rt,
             execute_rs => rs,
             execute_op => op,
             decode_instr => instr);
  execute1:execute
    port map(branch_decision => bd,
             dmem_rw => read_write,
             dmem_addr => addr,
             execute_offset => offset,
             execute_func => func,
             execute_rd => rd,
             execute_rt => rt,
             execute_rs => rs,
             execute_op => op);
  dmem1:dmem
    port map(read_write => read_write,
             data_out => dataout,
             data_in => datain,
             address => addr);
end structure;
```

Now let's consider specification of each of the individual blocks using our channel model. First, the *imem* block models a small (eight-word) memory using an array to store the instructions. The array is hardcoded to store a particular program. Therefore, to simulate a different program, it is necessary to edit the code. The *imem* block receives an address from the *fetch* block. This address is a *std_logic_vector*, so it is converted to an integer and then used to index into the memory array to find the instruction requested. The memory than waits for the specified amount of time to simulate the delay of the memory (this is also necessary to make sure that *instr* has taken its new value). Then it sends the fetched instruction to the *fetch* block. The VHDL code is shown below.

```
--------------------
-- imem.vhd
--------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity imem is
  port(address:inout channel:=init_channel;
       data:inout channel:=init_channel);
end imem;
architecture behavior of imem is
  type memory is array (0 to 7) of
    std_logic_vector(31 downto 0);
  signal addr:std_logic_vector(31 downto 0);
  signal instr:std_logic_vector(31 downto 0);
begin
process
  variable imem:memory:=(
    X"8c220000", -- L: lw r2,0(r1)
    X"8c640000", --    lw r4,0(r3)
    X"00a42020", --    add r4,r5,r4
    X"00642824", --    and r5,r3,r4
    X"ac640000", --    sw r4,0(r3)
    X"00822022", -- M: sub r4,r4,r2
    X"1080fff9", --    beq r4,r0,L
    X"18000005");--    j M
begin
  receive(address,addr);
  instr <= imem(conv_integer(addr(2 downto 0)));
  wait for delay(5,10);
  send(data,instr);
end process;
end behavior;
```

The *fetch* block fetches instructions and determines the PC for the next instruction. The behavior of the *fetch* block is that it sends the current PC to the *imem* block, it waits to receive a new instruction, and it increments the PC. It then partially decodes the instruction to determine if it is a branch or jump. If it is a branch, it forwards the instruction to the *decode* block and waits to hear from the *execute* block whether or not the branch is to be taken. If it is taken, it determines the new PC by adding a sign-extended version of the *offset* to the current PC. Otherwise, no action is necessary since the PC has already been incremented. If the new instruction is a jump, the *fetch* block can simply insert the address into the lower 26 bits of the current PC. All other types of instructions are simply forwarded to the decode block. The VHDL code is shown below. Note the use of *aliases* for the opcode, offset, and address. These allow you to associate names for partial ranges of an array.

```
--------------------
-- fetch.vhd
--------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity fetch is
  port(imem_address:inout channel:=init_channel;
       imem_data:inout channel:=init_channel;
       decode_instr:inout channel:=init_channel;
       branch_decision:inout channel:=init_channel);
end fetch;
architecture behavior of fetch is
  signal PC:std_logic_vector(31 downto 0):=(others=>'0');
  signal instr:std_logic_vector(31 downto 0);
  signal bd:std_logic;
  alias opcode:std_logic_vector(5 downto 0) is
    instr(31 downto 26);
  alias offset:std_logic_vector(15 downto 0) is
    instr(15 downto 0);
  alias address:std_logic_vector(25 downto 0) is
    instr(25 downto 0);
begin
process
  variable branch_offset:std_logic_vector(31 downto 0);
begin
  send(imem_address,PC);
  receive(imem_data,instr);
  PC <= PC + 1;
  wait for delay(5,10);
  case opcode is
```

```
  when "000100" => -- beq
    send(decode_instr,instr);
    receive(branch_decision,bd);
    if (bd = '1') then
      branch_offset(31 downto 16):=(others=>instr(15));
      branch_offset(15 downto 0):=offset;
      PC <= PC + branch_offset;
      wait for delay(5,10);
    end if;
  when "000110" => -- j
    PC <= (PC(31 downto 26) & address);
    wait for delay(5,10);
  when others =>
    send(decode_instr,instr);
  end case;
end process;
end behavior;
```

The major component of the *decode* block is the register file. For simplicity, our register file has only eight registers, whose values are stored in an array of *std_logic_vectors* in this block. The *decode* block receives an instruction from the *fetch* block, from which it extracts the *rs* and *rt* fields to index into the register array to find these registers' current values. It also extracts the opcode and sends it to the execute block. If the instruction is an ALU operation (i.e., add, sub, and, or), it sends the function field and the register values to the *execute* block. It then waits to receive the result from the *execute* block, and it stores this result into the register pointed to by the *rd* field in the instruction. If it had been a branch instruction, it sends the *rs* and *rt* registers to the *execute* block. If it is a load word, it sends the *rs* register and the offset to the *execute* block. It then waits to receive the result of the load from the *dmem* block, and it stores this value in the *rt* register. Finally, if it is a store, it sends the *rs* register and the offset to the *execute* block, and it sends the *rt* register to the *dmem* block. The VHDL code is shown below.

```
--------------------
-- decode.vhd
--------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity decode is
  port(decode_instr:inout channel:=init_channel;
       execute_op:inout channel:=init_channel;
       execute_rs:inout channel:=init_channel;
       execute_rt:inout channel:=init_channel;
       execute_rd:inout channel:=init_channel;
```

```
        execute_func:inout channel:=init_channel;
        execute_offset:inout channel:=init_channel;
        dmem_datain:inout channel:=init_channel;
        dmem_dataout:inout channel:=init_channel);
end decode;
architecture behavior of decode is
  type registers is array (0 to 7) of
    std_logic_vector(31 downto 0);
  signal instr:std_logic_vector(31 downto 0);
  alias op:std_logic_vector(5 downto 0) is
    instr(31 downto 26);
  alias rs:std_logic_vector(2 downto 0) is
    instr(23 downto 21);
  alias rt:std_logic_vector(2 downto 0) is
    instr(18 downto 16);
  alias rd:std_logic_vector(2 downto 0) is
    instr(13 downto 11);
  alias func:std_logic_vector(5 downto 0) is
    instr(5 downto 0);
  alias offset:std_logic_vector(15 downto 0) is
    instr(15 downto 0);
  signal reg:registers:=(X"00000000",
                         X"11111111",
                         X"22222222",
                         X"33333333",
                         X"44444444",
                         X"55555555",
                         X"66666666",
                         X"77777777");
  signal reg_rs:std_logic_vector(31 downto 0);
  signal reg_rt:std_logic_vector(31 downto 0);
  signal reg_rd:std_logic_vector(31 downto 0);
begin
process
begin
  receive(decode_instr,instr);
  reg_rs <= reg(conv_integer(rs));
  reg_rt <= reg(conv_integer(rt));
  wait for delay(5,10);
  send(execute_op,op);
  case op is
  when "000000" => -- ALU op
    send(execute_func,func,execute_rs,reg_rs,
         execute_rt,reg_rt);
    receive(execute_rd,reg_rd);
    reg(conv_integer(rd)) <= reg_rd;
    wait for delay(5,10);
  when "000100" => -- beq
    send(execute_rs,reg_rs,execute_rt,reg_rt);
```

```
  when "100011" => -- lw
    send(execute_rs,reg_rs,execute_offset,offset);
    receive(dmem_dataout,reg_rt);
    reg(conv_integer(rt)) <= reg_rt;
    wait for delay(5,10);
  when "101011" => -- sw
    send(execute_rs,reg_rs,execute_offset,offset,
         dmem_datain,reg_rt);
  when others => -- undefined
    assert false
      report "Illegal instruction"
      severity error;
  end case;
end process;
end behavior;
```

The *execute* block begins by waiting to receive an opcode from the *decode* block. If this opcode is an ALU-type instruction, it then waits to receive the *func* field and its two operands from the *decode* block. Using the *func* field, it determines which type of ALU operation to perform, and it performs it on the two operands. After waiting for a short time to model delay through the ALU, it sends the result back to the *decode* block. If the opcode had been a branch, it would then wait for two operands, which it needs to compare. If they are equal, it will send back a '1' to the *fetch* block to indicate that a branch has been taken. If they are not equal, it will send a '0' back to the *fetch* block to indicate that the branch is not taken. If it is a load, it waits to receive the register containing the base address and the offset from the *decode* block. It sign extends the offset and adds it to the base to determine the address. It then sends the address and a read indication to the *dmem* block. Finally, if it is a store, it again waits to receive the base address and offset from the *decode* block and computes the address. It then sends the address and a write indication to the *dmem* block. The VHDL code is shown below.

```
----------------------
-- execute.vhd
----------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity execute is
  port(execute_op:inout channel:=init_channel;
       execute_rs:inout channel:=init_channel;
       execute_rt:inout channel:=init_channel;
       execute_rd:inout channel:=init_channel;
       execute_func:inout channel:=init_channel;
       execute_offset:inout channel:=init_channel;
```

```
        dmem_addr:inout channel:=init_channel;
        dmem_rw:inout channel:=init_channel;
        branch_decision:inout channel:=init_channel);
end execute;
architecture behavior of execute is
  signal rs:std_logic_vector(31 downto 0);
  signal rt:std_logic_vector(31 downto 0);
  signal rd:std_logic_vector(31 downto 0);
  signal op:std_logic_vector(5 downto 0);
  signal func:std_logic_vector(5 downto 0);
  signal offset:std_logic_vector(15 downto 0);
  signal rw:std_logic;
  signal bd:std_logic;
begin
process
  variable addr_offset:std_logic_vector(31 downto 0);
begin
  receive(execute_op,op);
  case op is
  when "000000" => -- ALU op
    receive(execute_func,func,execute_rs,rs,
            execute_rt,rt);
    case func is
    when "100000" => -- add
      rd <= rs + rt;
    when "100010" => -- sub
      rd <= rs - rt;
    when "100100" => -- and
      rd <= rs and rt;
    when "100101" => -- or
      rd <= rs or rt;
    when others =>
      rd <= (others => 'X'); -- undefined
    end case;
    wait for delay(5,10);
    send(execute_rd,rd);
  when "000100" => -- beq
    receive(execute_rs,rs,execute_rt,rt);
    if (rs = rt) then bd <= '1';
    else bd <= '0';
    end if;
    wait for delay(5,10);
    send(branch_decision,bd);
  when "100011" => -- lw
    receive(execute_rs,rs,execute_offset,offset);
    addr_offset(31 downto 16):=(others => offset(15));
    addr_offset(15 downto 0):=offset;
    rd <= rs + addr_offset;
    rw <= '1';
```

```
      wait for delay(5,10);
      send(dmem_addr,rd);
      send(dmem_rw,rw);
    when "101011" => -- sw
      receive(execute_rs,rs,execute_offset,offset);
      addr_offset(31 downto 16):=(others => offset(15));
      addr_offset(15 downto 0):=offset;
      rd <= rs + addr_offset;
      rw <= '0';
      wait for delay(5,10);
      send(dmem_addr,rd);
      send(dmem_rw,rw);
    when others => -- undefined
      assert false
        report "Illegal instruction"
        severity error;
    end case;
    end process;
  end behavior;
```

The last block is the *dmem* block, which models a small memory (eight words), which is again modeled as an array. This block first waits to receive an address and a read/write indication from the *execute* block. If the *read/*$\overline{write}$ bit, *rw*, is '1', it is a read. A read uses the address it receives to index into the memory array. It then sends this value to the *decode* block. If it is a write, it waits to receive the data to write from the *decode* block. It then writes this value into the memory array. The VHDL code is shown below.

```
-------------------
-- dmem.vhd
-------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity dmem is
  port(address:inout channel:=init_channel;
       data_in:inout channel:=init_channel;
       data_out:inout channel:=init_channel;
       read_write:inout channel:=init_channel);
end dmem;
architecture behavior of dmem is
  type memory is array (0 to 7) of
    std_logic_vector(31 downto 0);
  signal addr:std_logic_vector(31 downto 0);
  signal d:std_logic_vector(31 downto 0);
  signal rw:std_logic;
```

```
  signal dmem:memory:=(X"00000000",
                       X"11111111",
                       X"22222222",
                       X"33333333",
                       X"44444444",
                       X"55555555",
                       X"66666666",
                       X"77777777");
begin
process
begin
  receive(address,addr);
  receive(read_write,rw);
  case rw is
  when '1' =>
    d <= dmem(conv_integer(addr(2 downto 0)));
    wait for delay(5,10);
    send(data_out,d);
  when '0' =>
    receive(data_in,d);
    dmem(conv_integer(addr(2 downto 0))) <= d;
    wait for delay(5,10);
  when others =>
    wait for delay(5,10);
  end case;
end process;
end behavior;
```

2.7.2 Optimized MiniMIPS

In synchronous design, the performance of a processor is improved through *pipelining*. The idea of pipelining is whenever possible to keep all parts of the circuit doing useful work concurrently. This is analogous to an assembly line in an auto factory. One worker puts the tires on a car and passes the car to the next person, who puts on the doors. However, while that worker is putting the doors on that car, the previous worker is putting the tires on the next car. In a microprocessor, a simple assembly line could have one stage fetching an instruction, another decoding an instruction, while a third executes an instruction. Ideally, all three stages could be working on different instructions at the same time.

The question, therefore, is whether our asynchronous design exhibits behavior akin to pipelining. The answer is "yes and no." If you simulate our design and watch the flow of instructions through our three stages (*fetch*, *decode*, and *execute*), what you see is that the *fetch* block can fetch the next instruction (assuming that the current instruction is not a branch) while the *decode* block is decoding the current instruction. However, the *decode* block and *execute* block operate sequentially. The reason for this is that even though

the *decode* block passes an instruction to the *execute* block, it must wait for that block to complete before it can decode the next instruction. For example, if the instruction is an ALU operation, the *decode* block is stuck waiting for the result of the operation, which is to be stored in the register file.

To eliminate this problem, we must split the *decode* block into two processes. The first process is responsible for decoding the instruction and forwarding it to the *execute* block. The second process is responsible for collecting the results of the execution and writing them back to the register file. There is a problem, though. If we naively split the process into two parts, we can introduce the possibility of *data hazards*. For example, a read after write (RAW) hazard is one where an instruction that writes a register is followed by one that reads that same register. If we allow the *decode* block to work concurrently with the *execute* block in this case, it is possible that the old value of the register will be read instead of the new value. This would obviously cause the program to generate wrong results. For example, let's assume that initially, *r1* contains 1 and *r2* contains 2, and we execute the following two instructions:

```
add r1,r2,r2
add r4,r1,r1
```

If the second instruction is decoded and has its operands fetched before the first completes its execution and has written back its result, *r4* will end up containing 2 when it should contain 8.

Perhaps the simplest way to eliminate data hazards is through *register locking*. In register locking, while an instruction is being decoded and before it is dispatched to the *execute* block, the destination register for that instruction is locked. When the next instruction arrives, if it needs to read that register, it will find that it is locked and stall until the lock is released. Thus, this mechanism prevents stale data from ever being read. Our *decode* block, rewritten to allow concurrent decoding and execution without data hazards, is shown below.

```
---------------------
-- decode.vhd
---------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity decode is
  port(decode_instr:inout channel:=init_channel;
       execute_op:inout channel:=init_channel;
       execute_rs:inout channel:=init_channel;
       execute_rt:inout channel:=init_channel;
       execute_rd:inout channel:=init_channel;
```

```
        execute_func:inout channel:=init_channel;
        execute_offset:inout channel:=init_channel;
        dmem_datain:inout channel:=init_channel;
        dmem_dataout:inout channel:=init_channel);
end decode;
architecture behavior of decode is
  type registers is array (0 to 7) of
        std_logic_vector(31 downto 0);
  type booleans is array (natural range <>) of boolean;
  signal instr :  std_logic_vector( 31 downto 0);
  alias op:std_logic_vector(5 downto 0) is
    instr(31 downto 26);
  alias rs:std_logic_vector(2 downto 0) is
    instr(23 downto 21);
  alias rt:std_logic_vector(2 downto 0) is
    instr(18 downto 16);
  alias rd:std_logic_vector(2 downto 0) is
    instr(13 downto 11);
  alias func:std_logic_vector(5 downto 0) is
    instr(5 downto 0);
  alias offset:std_logic_vector(15 downto 0) is
    instr(15 downto 0);
  signal reg:registers:=(X"00000000",
                         X"11111111",
                         X"22222222",
                         X"33333333",
                         X"44444444",
                         X"55555555",
                         X"66666666",
                         X"77777777");
  signal reg_rs:std_logic_vector(31 downto 0);
  signal reg_rt:std_logic_vector(31 downto 0);
  signal reg_rd:std_logic_vector(31 downto 0);
  signal reg_locks:booleans(0 to 7):=(others => false);
  signal decode_to_wb:channel:=init_channel;
  signal wb_instr:std_logic_vector(31 downto 0);
  alias wb_op:std_logic_vector(5 downto 0) is
    wb_instr(31 downto 26);
  alias wb_rt:std_logic_vector(2 downto 0) is
    wb_instr(18 downto 16);
  alias wb_rd:std_logic_vector(2 downto 0) is
    wb_instr(13 downto 11);
  signal lock:channel:=init_channel;
begin
decode:process
begin
  receive(decode_instr,instr);
  if ((reg_locks(conv_integer(rs))) or
      (reg_locks(conv_integer(rt)))) then
```

```
    wait until ((not reg_locks(conv_integer(rs))) and
                (not reg_locks(conv_integer(rt))));
  end if;
  reg_rs <= reg(conv_integer(rs));
  reg_rt <= reg(conv_integer(rt));
  send(execute_op,op);
  wait for delay(5,10);
  case op is
  when "000000" => -- ALU op
    send(execute_func,func,execute_rs,reg_rs,
         execute_rt,reg_rt);
    send(decode_to_wb,instr);
    receive(lock);
  when "000100" => -- beq
    send(execute_rs,reg_rs,execute_rt,reg_rt);
  when "100011" => -- lw
    send(execute_rs,reg_rs,execute_offset,offset);
    send(decode_to_wb,instr);
    receive(lock);
  when "101011" => -- sw
    send(execute_rs,reg_rs,execute_offset,offset,
         dmem_datain,reg_rt);
  when others => -- undefined
    assert false
      report "Illegal instruction"
      severity error;
  end case;
end process;
writeback:process
begin
  receive(decode_to_wb,wb_instr);
  case wb_op is
  when "000000" => -- ALU op
    reg_locks(conv_integer(wb_rd)) <= true;
    wait for 1 ns;
    send(lock);
    receive(execute_rd,reg_rd);
    reg(conv_integer(wb_rd)) <= reg_rd;
    wait for delay(5,10);
    reg_locks(conv_integer(wb_rd)) <= false;
    wait for delay(5,10);
  when "100011" => -- lw
    reg_locks(conv_integer(wb_rt)) <= true;
    wait for 1 ns;
    send(lock);
    receive(dmem_dataout,reg_rd);
    reg(conv_integer(wb_rt)) <= reg_rd;
    wait for delay(5,10);
    reg_locks(conv_integer(wb_rt)) <= false;
```

```
        wait for delay(5,10);
      when others => -- undefined
        wait for delay(5,10);
      end case;
    end process;
    end behavior;
```

Let's first consider the *decode* process. After this process receives an instruction, it checks if either of its source operands, *rs* or *rt*, is locked, and if either is locked, it stalls until they are both released. After checking that all the necessary registers are unlocked, it can then dispatch the instruction to the *execute* block. If this instruction results in data being written into the register file (i.e., it is an ALU operation or a load word), it then also forwards the instruction to the *write_back* process and waits to receive from this process a lock on the destination register. Note that the *receive* procedure call here takes a channel but no data, since it is used only for synchronization. Once it has received a lock, it can then go ahead and begin decoding the next instruction.

The *write_back* process waits to receive an instruction. Once it does, it locks the appropriate destination register and sends the lock back to the *decode* process. It then waits to receive the result from the *execute* block or the *dmem* block. Once it does, it writes the result to the register file and removes the lock on that register.

2.8 SOURCES

The channel model described in this chapter is inspired by Hoare's *communicating sequential processes* (CSP) [166, 167], with Martin's addition of the probe [248]. Martin first adapted CSP to model asynchronous circuits and systems [249, 250, 252, 254, 255]. A similar channel based model is used in the *Tangram* language proposed by van Berkel et al.[37, 39]. *Occam*, a parallel programming language based on CSP, is used by Brunvand to model asynchronous circuits at a communication level [52, 53]. Gopalakrishnan and Akella use a language called *hopCP* [150].

When designing systems using the channel model, it is particularly important to avoid deadlock. Friedman and Menon investigated interconnections of modules operating using asynchronous codes and determined conditions under which they could deadlock [130]. They also showed that by adding buffers the system performance could actually be improved. Bruno and Altman developed conditions under which an interconnected control structure composed of `JUNCTION`, `WYE`, `SEQUENCE`, `ITERATE`, and `SELECT` would deadlock [51]. Jump and Thiagarajan developed a methodology for checking for deadlock in an interconnection of asynchronous control structures which translates the network into a marked graph and then performs a simple analysis [184].

Problems

2.1 In this problem you will specify a 4-bit adder using the channel model. The adder has three input ports and two output ports. The input ports are X and Y, which are used to pass in the 4-bit integer operands, and *Cin*, which is a 1-bit carry-in. The output ports are *Sum*, which is a 4-bit integer sum, and *Cout*, which is a 1-bit carry-out.

2.1.1. Specify the 4-bit adder in VHDL using the channel model. The *adder4* process should accept the two operands a and b and the carry-in c from the corresponding input ports (X, Y, Cin). It should then compute the sum, s, and carry-out, d, and output the results on the corresponding output ports (*Sum*, *Cout*). Create environment processes to generate random data using the selection procedure and consume the data. Simulate until you are convinced that it works.

2.1.2. Specify a 1-bit full adder using VHDL. The *adder1* process should accept two 1-bit operands a and b and the carry-in c from the corresponding ports (X, Y, Cin). It should then compute the 1-bit sum, s, and carry-out, d, and output the results on the corresponding output ports (*Sum*, *Cout*). You may use only logic functions (no arithmetic functions). Create an environment process which communicates with the 1-bit adder and simulate until you are convinced that it works.

2.1.3. Use structural VHDL to build a 4-bit adder using your 1-bit full adders from Problem 2.1.2. Create an environment process which communicates with the 4 bits of the adder. Convince yourself through simulation that it performs the same function as your 4-bit adder from Problem 2.1.1.

2.1.4. Optimize your 1-bit full adder from Problem 2.1.2 so that it takes advantage of the fact that for most computations, the longest carry chain is significantly shorter than n, where n is the number of bits in your adder.

2.2 Specify and simulate a four-element stack using a channel-level model. The stack should be constructed using a number of identical modules which will each hold a single 8-bit data value. Each module has two passive channels to its left, *Push* and *Pop*, and two active channels to its right, *Put* and *Get*. The module should wait to get either a communication on its *Push* or its *Pop* channel. If it detects a communication on its *Push* channel and it is empty, it should receive the data and store the value received internally. If it is full, it should communicate its data to the right using its *Put* channel, then complete the receive from the *Push* channel. If it gets a communication on its *Pop* channel and it is full, it should send its data over the *Pop* channel. If it is empty, it should request data over its *Get* channel and then forward the received data out the *Pop* channel. For simplicity, assume that the environment will never push data into a full stack or pop data from an empty stack.

2.3 Specify and simulate a 4-bit shifter. It should be constructed using three identical modules which each hold a single bit of data, and a special module at the end of the shifter. Each module has a *Load* channel, *Shift_in* channel, *Shift_out* channel, *Done_in* channel, *Done_out* channel, and an *Output* channel. Each module waits to receive a communication on its *Load* channel, at which time it accepts a bit of data. Next, the most significant bit waits to receive a communication from the environment on either its *Shift_in* channel or its *Done_in* channel. If it is on the *Shift_in* channel, it sends its bit to the next most significant bit over the *Shift_out* channel and it accepts a new bit from the *Shift_in* channel. If it receives a communication on its *Done_in* channel, it sends its bit out the *Output* channel and sends a communication on its *Done_out* channel. Note that the *Done_in* and *Done_out* channels do not need to carry any data; they are used only for synchronization. The block at the end does not have a *shift_out* or *Done_out* channel.

2.4 In this problem you are going to specify a simple entropy decoder in VHDL using the channel model. An entropy decoder relies on a standard entropy code that represents fixed-length symbols from a source alphabet as variable-length code symbols. The idea is that common symbols are represented using short codes, and uncommon symbols are represented using longer codes. The result is that the average code length is significantly smaller than the symbol length. Such entropy codes are used in numerous audio, video, and data compression schemes (e.g., JPEG, MPEG, etc.). As an example, consider a set of 2-bit symbols (i.e., 00, 01, 10, 11) where 00 has a probability of 90 percent, 01 has a probability of 9 percent, 10 has a probability of 0.9 percent, and 11 has a probability of 0.1 percent. If we encode the symbols as follows:

- 00 - 0
- 01 - 10
- 10 - 110
- 11 - 1110

the average code length would be

$$0.9 \times 1 + 0.09 \times 2 + 0.009 \times 3 + 0.001 \times 4 = 1.11$$

This is nearly half the size of the fixed-length size of the symbols.

2.4.1. Specify in VHDL using the channel model an entropy decoder for 2-bit symbols which are encoded as described above. Assume that you have a 1-bit input port, *In*, and a 2-bit output port, *Out*. The module should receive 1 bit of data at a time and output the 2-bit symbol once it has recognized the code word. Specify environment processes to generate and consume the data. Simulate the design until you are convinced that it works.

2.4.2. We have decided to break up the entropy decoder from the first problem into four identical blocks, and we also would like to make it programmable. Each block has a 1-bit input port, *L*, which receives bits from its left neighbor, a 1-bit output port, *R*, which transmits bits to its right neighbor, a 2-bit input port, *Load*, which receives the 2-bit symbol this block transmits when it recognizes the code, and a 2-bit output port, *Out*, which transmits its stored symbol when it recognizes the code. The behavior of each block is that it can either receive a 2-bit symbol from the *Load* port or 1 bit from its left neighbor. If it receives a 2-bit symbol from the *Load* port, that symbol is stored in an internal register and becomes the symbol for which this stage is responsible. If it receives a bit from its left neighbor, it checks if it is a 0 or 1. If it is a 0, it will output its stored symbol. If it is a 1, it strips this bit and passes the remaining bits that it receives from its left neighbor one by one to its right neighbor until it sees a 0. At that point it loops back to the beginning of the cycle and waits for the next code to be transmitted. Specify one block in VHDL. Create an environment and simulate until you are convinced that it works.

2.4.3. Use structural VHDL to model an entropy decoder which can recognize codes of length 4.

2.4.4. The results being transmitted from the *Out* ports from each of the four blocks from the last problem need to be multiplexed together to produce the symbol that has just been recognized. There is a potential race problem with this design as described above. One stage may come to the end of forwarding bits (i.e., it sees the trailing 0) and immediately sees another 0, causing it to transmit its symbol on its *Out* port. This transmission may occur before the transmission by the later stage of the previous symbol. In other words, the symbols may get recognized out of order. Add another channel to forward acknowledgments of transmissions of symbols back to the previous blocks. Try to do this in a way which will prevent transmissions from getting out of order and does not hold up the recognition step.

2.5 Add a jump and link (*jal*) instruction to the MiniMIPS example from Section 2.7. This instruction is of the jump format, and the opcode is 000011. This instruction should send the old PC+1 to register 31 (effectively register 7 in our scaled-down design), and set the PC to the address.

2.6 Extend the MiniMIPS to include the set less than (*slt*) instruction. This instruction is of the register format, and the opcode is 000000 with function code 101010. It checks if register *rs* is less than *rt*. If so, it sets *rd* to 1. Otherwise, it sets *rd* to 0. This instruction typically is used before a branch.

2.7 Extend the MiniMIPS to include immediate instructions: add immediate (*addi*, op = 001000), and immediate (*andi*, op = 001100), or immediate (*ori*, op = 001101), and set less than immediate (*slti*, op = 001010). These instructions are of the load/store/branch format. The *rs* field points to one operand and the offset is the other operand. The offset is sign extended for

addi and *slti*, while it is zero extended for *andi* and *ori*. The result is stored in the register pointed to by the *rt* field.

2.8 Extend the MiniMIPS to include shift instructions: shift left logical (*sll*), shift right logical (*srl*), and shift right arithmetic (*sra*). These instructions are of the register format with opcode 000000. The function codes are 000000 for *sll*, 000010 for *srl*, and 000011 for *sra*. They shift *rs* by the amount in *rt* and store the result in *rd*. In an arithmetic shift, the high-order bit is shifted in. In logical shifts, a 0 is shifted in.

2.9 Extend the MiniMIPS to include a *trap* and a return from exception (*rfe*) instruction.

2.10 Extend the MiniMIPS to support exceptions. In particular, *add* and *sub* instructions can cause an arithmetic overflow exception, and illegal instructions should also cause an exception.