Designing Digital Circuits Using VHDL©

partial draft

January 2012

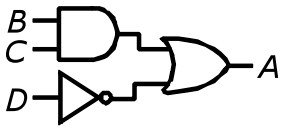
Jonathan Turner

1. Getting Started

Let's start with the basics. *Signal assignments* are the most common element of a VHDL circuit specification. Here's an example.

```
A <= (B and C) or (not D);
```

Here, `A`, `B`, `C` and `D` are names of VHDL *signals*; `<=` is the *signal assignment operator* and the keywords `and`, `or` and `not` are the familiar logical operators. The parentheses are used to determine the order of operations (in this case, they are not strictly necessary, but do help make the meaning more clear) and the semicolon terminates the assignment. This assignment can be implemented by the *combinational circuit* shown below.



Any logic circuit made up of AND gates, OR gates and inverters in which there are no feedback paths is a combinational circuit (a feedback path is a circuit path that leads from a gate output back to an input of the same gate). Every VHDL assignment corresponds to a combinational circuit, and any combinational circuit can be implemented using one or more VHDL assignments. The specific circuit shown above is only one possible implementation of the given signal assignment. Any *logically equivalent* circuit is also an acceptable implementation (when we say logically equivalent circuit, we mean any circuit that produces the same output value as the above circuit for every set of input values). The *meaning* of the

given assignment is any circuit that is logically equivalent to the one shown above.

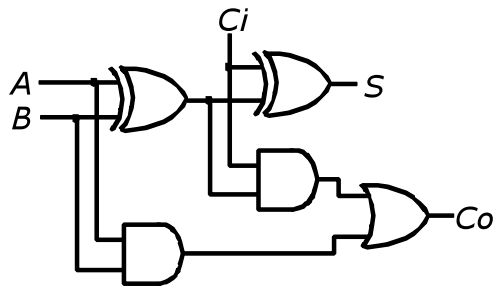
The following pair of signal assignments specifies one bit position of an *n* bit adder.

```
S <= A xor B xor Ci;
Co <= (A and B) or ((A xor B) and Ci);
```

Here, `A` and `B` represent corresponding bits of the two binary numbers being added together and `Ci` represents the carry into this bit position. `S` is the sum for this bit position and `Co` is the carry out of this bit position. The `xor` keyword represents the exclusive-or operator. For any expressions `X` and `Y`, `X xor Y` is equivalent to

```
(X and (not Y)) or ((not X) and Y)
```

We note that no parentheses are required in the assignment to `S` since the exclusive-or operator is associative. This pair of assignments could be implemented by two separate circuits that happen to share the same inputs, but the circuit shown below provides a more efficient implementation, since it uses the first exclusive-or gate to produce both of the output signals.



The signal assignments are only one part of a VHDL circuit specification. To completely define a circuit, we must also specify its inputs and outputs. As an example, here is a complete specification of the full adder circuit.

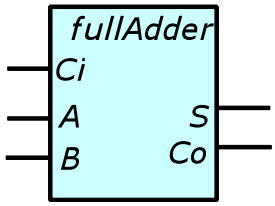
```
-- Full adder.
-- Produces sum bit (S) and carry-out (Co),
-- given data inputs (A,B) and carry-in (Ci).

entity fullAdder is port(
   A, B, Ci: in std_logic;
   S, out: out std_logic);
end fullAdder;

architecture faArch of fullAdder is
begin
   S <= A xor B xor Ci;
   Co <= (A and B) or ((A xor B) and Ci);
end faArch;
```

The first three lines are a *comment* describing the circuit. In general, a pair of dashes introduces a comment, which continues to the end of the line. Comments don't affect the meaning of the specification, but are essential for making it readable by other people. It's a good idea to use comments to explain the inputs and outputs of your circuits, document what your circuits do and how they work. Get in the habit of documenting all your code.

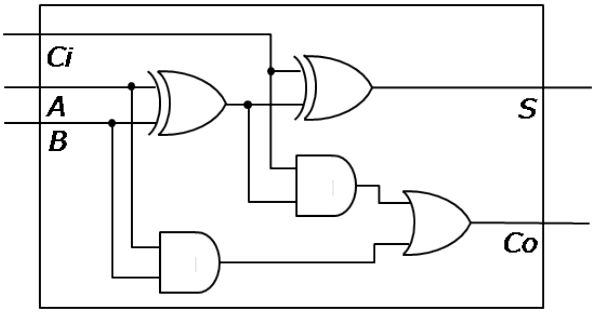
The next few lines are the *entity declaration* for our full adder circuit. The entity declaration defines the name of the circuit (`fullAdder`), its inputs and outputs and their types (`std_logic`). The inputs and outputs are specified in a *port list*. Successive elements of the port list are separated by semicolons (note that there is no semicolon following the last element).

The last five lines above, constitute the architecture specification, which includes two signal assignments. VHDL permits you to have multiple architectures for the same entity, hence the architecture has its own label, separate from the entity name. Note that the architecture name also appears in the statement that ends the architecture specification.

It's important to understand the distinction between the entity declaration and the architecture. The entity declaration defines the circuit's external interface and the architecture defines its internal implementation. In a block diagram or abridged schematic, we often show a portion of a larger circuit as a block with labeled inputs and outputs, as illustrated below for the `fullAdder`.



This corresponds directly to the entity declaration. When we supplement such a diagram, by filling in the block with an appropriate schematic, we are effectively specifying its architecture.



We often find it convenient to define internal signals that are used in other assignments, but are not outputs of the circuit we are specifying. For example, we might specify the full adder using the following assignments.

```
generate <= A and B;
propagate <= A xor B;
S <= propagate xor Ci;
Co <= generate or (propagate and Ci);
```

This specifies a circuit that is logically equivalent to the previous one. Before we can use these internal signals, we must declare them. Here's a modified version of the architecture that includes the necessary declarations.

```
architecture faArch2 of fullAdder is
signal generate, propagate: std_logic;
begin
generate <= A and B;
propagate <= A xor B;
   S <= propagate xor Ci;
   Co <= generate or (propagate and Ci);
end faArch2;
```

This example provides a good illustration of the difference between VHDL and conventional programming languages. Suppose that we wrote the assignments as shown below.

```
S <= propagate xor Ci;
Co <= generate or (propagate and Ci);
generate <= A and B;
propagate <= A xor B;
```

If these were assignments in a conventional programming language this version would not mean the same thing as the original version. Indeed, it might trigger an error message since `propagate` and `generate` are being used before they have been assigned values. However, in VHDL this version has exactly the same meaning, as the original because they both specify the same circuit. The order in which the statements appear makes no difference.

VHDL also supports composite signals or *signal vectors* which allow several simple signals to be treated as a unit. For example, if A and B are both signal vectors that represent the component signals A_0, A_1, A_2 and B_0, B_1, B_2 the assignment `A<=B;` is equivalent to the three simple assignments

```
A(0) <= B(0); A(1) <= B(1); A(2) <= B(2);
```

If C is a similar signal vector, the assignment `A<=B and C;` is equivalent to

```
A(0) <= B(0) and C(0);
A(1) <= B(1) and C(1);
A(2) <= B(2) and C(2);
```

We can also refer to parts of signal vectors. So, the assignment

```
A(0 to 2) <= B(3 downto 2) & C(2);
```

is equivalent to

```
A(0) <= B(3); A(1) <= B(2); A(2) <= C(2);
```

Here the ampersand (&) is a *signal concatenation* operator that is used to combine signals or signal vectors to form longer signal vectors. The *direction indicators*, `to` and `downto`, determine which end of a range of signals is considered the *left end* and which is the *right end*. Signal assignments use this notion of left-to-right ordering to determine which signals of the right-hand side vector are paired with signals on the left-hand side.

The right-hand side of a signal assignment may also include constant values, and there are several ways to specify constants. Single bit values are enclosed in single quotes (`'0'` or `'1'`). Multi-bit signals are written as strings enclosed by double quotes (`"001"` or `"11001"`). For multi-bit signals with more than a few bits, it's convenient to use hexadecimal constants. The constant `x"c4"` specifies the same value as `"11000100"`. VHDL allows constants to be specified in more general ways as well. For example, if A is an 8 bit signal vector, then the assignment

```
A <= (7|6 =>'1', 5 downto 3 =>'0', others =>'1');
```

is equivalent to `A <= "11000111"`. The special case

```
A <= (others => '0');
```

provides a convenient way to specify that all bits of `A` are `0`. This works no matter how many bits `A` actually has.

2. Warming Up

In Chapter 1, we saw how we could use VHDL to design some very simple combinational circuits. In this chapter, we'll introduce some additional features of the language and see how they can be used to design more complex circuits. As an example, we'll use a circuit that implements a very simple arithmetic calculator. The circuit has an 8 bit data input called `dIn`, an 8 bit data output called `result` and control inputs `clear`, `load` and `add`. It has an internal storage register that can store an 8 bit value. The `result` output is the value stored in this register. When the `clear` input is asserted, the stored value is cleared, when the `load` input is asserted the value of `dIn` is stored, and when the `add` input is asserted, the value of `dIn` is added to the stored value. The entity declaration for the circuit is shown below.

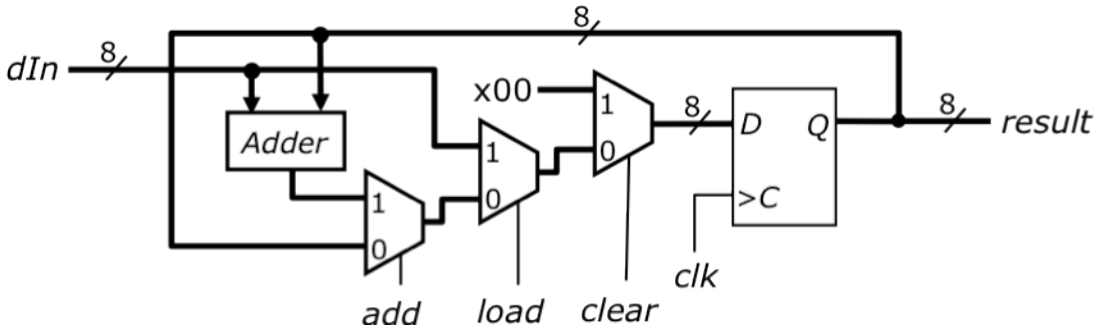
```
entity calculator is port (
   clk: in std_logic;
   clear, load, add: in std_logic;
   dIn: in std_logic_vector(7 downto 0);
   result: out std_logic_vector(7 downto 0));
end calculator;
```

The `clk` input controls when the circuit responds to control inputs. In particular, it performs an operation only when the `clk` input makes a

transition from low to high. So here is the architecture of a circuit that implements the desired functionality.

```
architecture calcArch of calculator is
signal dReg: std_logic_vector(7 downto 0);
begin
   process (clk) begin
     if rising_edge(clk) then
        if clear = '1' then
           dReg <= x"00";
        elsif load = '1' then
           dReg <= dIn;
        elsif add = '1' then
           dReg <= dReg + dIn;
        end if;
     end if;
   end process;
   result <= dReg;
end calcArch;
```

The storage register `dReg` is declared at the beginning of the architecture specification. The `process` block implements the core functionality of the circuit. The initial if-statement defines a *synchronization condition* on the `clk` signal. All assignments within the scope of the synchronization condition take place when there is a low-to-high transition on the `clk` signal. What this means, in terms of the circuit implementation, is that the signals that are assigned values within the scope of the synchronization condition must be stored in *clocked registers* that are triggered by the rising edge of `clk`. In this case, it means `dReg` is stored in such a register. Here's a diagram of a circuit that implements the given VHDL specification.



This circuit diagram includes some components that we haven't seen so far. The block labeled `Adder` implements an eight bit binary addition function, so its 8 bit output is the sum of its two 8 bit inputs. The trapezoidal symbols are *2:1 multiplexors*. Each of these has a pair of *data inputs* labeled 0 and 1, and a data output, on the right. They also have a *control input* at the bottom. The output of a 2:1 multiplexor (or mux, for short) is the value on one of its two data inputs. In particular, if the control input is low, the output is equal to the value on the data input labeled 0, and if the control input is high, the output is equal to the value on the data input labeled 1. The adder and multiplexor components are combinational circuits, so they can be implemented using AND gates, OR gates and inverters with no feedback.

The rectangular component at the right of the diagram is a clocked register made up of *positive edge-triggered D flip flops*. A flip flop is a storage device capable of storing one bit of information. The clocked D flip flop is the particular type of flip flop that is used most often. It has a data input and a clock input. The value stored in the flip flop appears as its output. A new value is stored in the flip flop every time the clock makes a transition from low to high. In particular, the value that is present on the D input is stored whenever such a transition occurs. Clock signals like `clk` are usually periodic, square wave signals with a fixed frequency. So

for example, if `clk` were a 50 MHz square wave, the calculator circuit could potentially perform an operation every 20 ns.

Considering the block diagram, we can see that on every clock transition, a value is stored in the register. If the `clear` signal is high, this value is x00. If the `clear` signal is low, but the `load` signal is high, the value stored comes from the data input. If `clear` and `load` are low, but `add` is high, the stored value is the sum of the "old" value and the data input. Finally, if all the control signals are low, the register is loaded with its old value (so, it doesn't change). Note how this reflects the logic expressed in the VHDL specification. Also notice how various elements of the circuit correspond to the VHDL. In particular, because the `dReg` signal is assigned within the scope of the synchronization condition, a register of clocked flip flops is included in the circuit to hold its value. The inner if-then-elsif-elsif construct in the VHDL specification is implemented using the sequence of multiplexors in the circuit diagram. Each condition determines the value of the control input of one of the multiplexors.

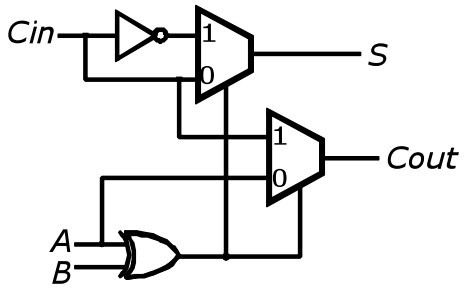
There are a couple more details worth noting. First, observe that the process statement includes the `clk` signal in parentheses. In general, the process statement can have a list of signals associated with it. This is called the *sensitivity list*. By placing a signal in the sensitivity list, we are making an assertion that the signals that are controlled by the process only change when one of the signals in the sensitivity list changes. In this case, because all assignments in the process fall within the scope of the synchronization condition, the signals controlled by the process can only change when `clk` changes.

Second, notice that the result output is assigned a value outside the process block. We could have placed this assignment inside the process block so long as it was placed *outside* the scope of the synchronization condition. If we had placed it inside the scope of the synchronization condition, the implementation would have included an additional register for `result`, connected in pipeline fashion to the register for `dReg`. This would effectively delay the appearance of the result by one clock tick, which is not what we want in this case.

Processes can also be used to define purely combinational circuits. For example, we could re-write the full-adder circuit from the previous chapter as shown below.

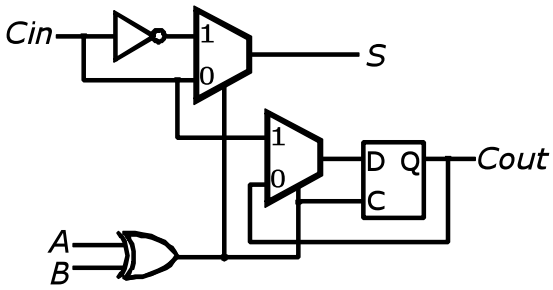
```
architecture a1 of fullAdder is
begin
   process (A, B, Cin) begin
      Cout <= A;
      if A /= B then
         S <= not Cin;
         Cout <= Cin;
      else
         S <= Cin;
      end if;
   end process;
end;
```

This VHDL specification can be implemented by this circuit.



which is logically equivalent to the circuit we saw in the previous chapter. Notice that the process does not include an if-statement with a synchronization condition. This means that the signals that are specified by the process (`S` and `Cout`) need not be stored in flip flops. Also notice that `S` and `Cout` are defined for all possible combinations of the input signals `A`, `B` and `Cin`. It's important that this be true for any process that we are using to define a combinational circuit. If we do not specify an

output for all possible input values, the VHDL language processor assumes that we intended for that output to retain its "old" value for that undefined set of inputs. For example, suppose we left out the default assignment of `A` to `Cout`. This would imply that whenever `A` and `B` go from being unequal to equal, the value of `Cout` should not change. The circuit shown below uses a *D-latch* to provide the required behavior.



A D-latch is a storage element that is similar to a flip flop, but it stores a new value not on the rising clock edge, but anytime the control input (`C`) is high. It useful to think of the D latch as being "transparent" when the `C` input is high (that is, the output equals the input), and it retains its "old value" whenever the `C` input is low. Notice that the symbol for the D-latch while similar to that the D flip flop is slightly different, since the clock input of the D flip flop is labeled with '>C', where the '>' indicates that the input is sensitive to the clock edge.

It's easy, when writing a process to implement a combinational circuit, to accidentally leave out an assignment that's needed to guarantee that the process' output signals are defined for all combinations of its input signals. When this happens, a circuit synthesizer will *infer a latch* to produce the specified behavior. If this is not what we want, the result will be a circuit that behaves differently from how we intended. This can be baffling when we are trying to verify the circuit's operation. A good way to avoid this problem is to assign some *default value* to every output signal of a process, right at the top of the process.

This example brings out another point about VHDL that is worth emphasizing. Let's take another look at the process.

```
process (A, B, Cin) begin
   Cout <= A;
   if A /= B then
      S <= not Cin;
      Cout <= Cin;
   else
      S <= Cin;
   end if;
end process;
```

In Chapter 1, we discussed an example in which the relative order of several assignment statements did not have any effect on the meaning of the VHDL. However, within a process, *the relative order of assignments to the same signal does matter*. So for example, if we moved the assignment `Cout <= A` so that it came after the if-then-else, the resulting circuit would cause `Cout` to equal `A` all the time. This is what we would expect, based on our experience with ordinary programming languages, but is different from what we might expect, based on our earlier discussion about ordering of assignments. The crucial distinction is that here we are talking about two assignments to the same signal within a process. VHDL treats the initial assignment to `Cout` as a *default value* to be used under any conditions where the value of `Cout` is otherwise unspecified. This leads to behavior that is similar to what we are used to from ordinary programming languages, but is not quite the same, since here we are still defining a circuit, not specifying a sequence of memory storage operations, as in conventional programming. It's important to keep this in mind when writing circuit specifications using processes.

There is one last aspect of this example that merits a closer look, and that is the sensitivity list. Notice that in this case, the sensitivity list includes the signals `A`, `B` and `Cin`. These signals are included, because a change to any one of these signals can cause the outputs of the process (`S` and `Cout`) to change. When using a process to define a combinational

circuit, it's best to include all signals that appear in conditional expressions or on the right side of assignments, in the sensitivity list. Now, you might be wondering, what's the point of the sensitivity list, and that's a reasonable question. The sensitivity list is included in the language primarily as an aid to simulation and modeling tools. If a circuit simulator knows that the only signals that can affect the outputs of a process are the ones included in its sensitivity list, it only needs to update its simulation state for the process when one of those signals changes. Unfortunately, if one inadvertently neglects to include a signal in the sensitivity list, this can cause a circuit simulation to behave differently than we expect. So, one of the first things to do when a simulation of a combinational process is behaving strangely, is to double-check the sensitivity list and make sure that every signal that can affect the process' outputs is included.

Before concluding this chapter, there are a few more things it's important to point out about processes. First, an architecture may contain more than one process, but there is an important restriction on how different processes interact. In particular, each signal that is defined within an architecture should be specified (assigned a value) by only one process. If two different processes had assignments to the same signal, the resulting circuit would contain two different sub-circuits, each trying to control the value of the shared signal. While one can build circuits like this, the usual outcome of doing so is the destruction of the circuit, often accompanied by a puff of blue smoke and a nasty smell. Now, signal assignments that appear outside of processes, are treated as one-line processes, so far as this rule is concerned. So if a signal is assigned a value outside of any process, it should not also be assigned a value inside a process. Also, a signal should typically be assigned a value by at most one assignment that lies outside of any process.

We noted earlier that a process may contain some statements that lie within the scope of a synchronization condition and others that lie outside the scope of the synchronization condition. Those statements that lie "within scope" determine the values of signals that are stored in registers. Those statements that lie outside the scope usually specify combinational

sub-circuits. Consequently, if a signal x is assigned a value within the scope of a synchronization condition, it should not also be assigned a value outside the scope of the condition.

3. Simulating Digital Circuits

Simulation is a powerful tool for verifying that a digital circuit works the way we intend it to. In order to do a circuit simulation, we must specify a sequence of inputs to the circuit that allows us to observe the circuit's operation under a wide range of conditions. To illustrate this, let's start by considering the calculator circuit from Chapter 2. The VHDL specification is repeated below.

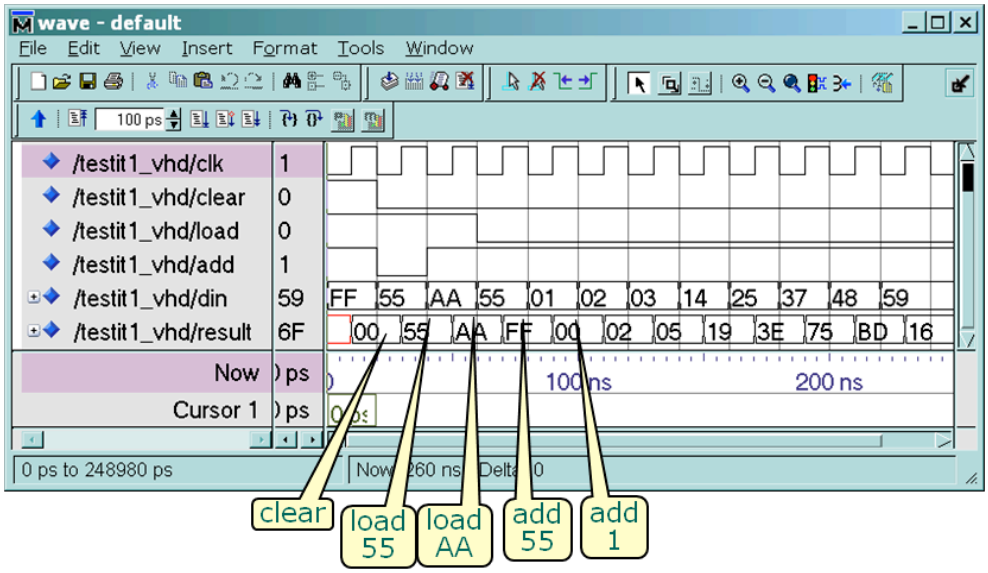
```
entity calculator is port (
   clk: in std_logic;
   clear, load, add: in std_logic;
   dIn: in std_logic_vector(7 downto 0);
   result: out std_logic_vector(7 downto 0));
end calculator;
architecture a1 of calculator is
signal dReg: std_logic_vector(7 downto 0);
begin
    process (clk) begin
      if rising_edge(clk) then
         if clear = '1' then
            dReg <= x"00";
         elsif load = '1' then
            dReg <= dIn;
         elsif add = '1' then
            dReg <= dReg + dIn;
         end if;
      end if;
   end process;
   result <= dReg;
end a1;
```

As it stands our circuit specification is not quite complete. Before we can simulate it, we need to add the following lines before the entity declaration.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;
```

The first line instructs the language system to load a standard library of common definitions called IEEE (this library was standardized by the Institute of Electrical and Electronics Engineers). The three *use statements* instruct the system to use three specific *packages* contained within the IEEE library. We will defer discussion of what's in these packages. For now, just keep in mind that the library and use clauses must be included in your VHDL specification every time you declare a new entity-architecture pair.

Now, let's look at a simulation of the calculator. Here is a waveform display that demonstrates the operation of the circuit.



The signal names appear at left, starting with `clk`, followed by the three control inputs, the data input and the results output. Notice how the `clear` signal causes the result output to go to zero, how the `load` signal causes a new value to be loaded from `dIn` and how the `add` signal causes the value on `dIn` to be added to the stored value. Also, note how every change to `result` happens on the rising clock edge, since this is when the internal register changes.

To produce such a simulation, we need to specify the sequence of input values to be applied to the circuit. When designing circuits with VHDL, this is done using a *testbench*. A testbench is essentially an ordinary program that specifies the sequence of input values to be applied to the circuit we are testing. It turns out that we can use VHDL to write the testbench program, which is convenient, but can be a little confusing, since the use of VHDL for testbench programming is distinctly different from the use of VHDL for specifying circuits.

At this point, it makes sense to discuss a little bit of the history of VHDL. When hardware description languages like VHDL were first developed, they were intended primarily as modeling and simulation

tools, to be used in the early stages of a hardware design project, before the development of a detailed design, which was typically done in the form of a schematic diagram. Consequently, the semantics of VHDL was specified with reference to a sequential circuit simulation. Only later, did CAD tool developers start creating circuit synthesizers that used VHDL as a circuit specification language. When one is using VHDL for synthesis, it makes much more sense to think about the VHDL language statements as specifying a circuit, rather than trying to understand them in terms of a sequential simulation. Many books on VHDL do describe the semantics of the language by reference to a simulation, but this makes it much more difficult for students (and even experienced computer engineers) to understand the circuit synthesis process and how to use the language effectively. This is why, in this book we emphasize the correspondence between language statements and circuits as directly as we can.

However, when discussing VHDL testbenches, we have to acknowledge the other side of VHDL, since in this context, the language is used not to specify a circuit, but to create a program, much like the programs we create in other languages. Because we are using VHDL for these different purposes, the way we use it and the interpretation of the language statements is a little bit different in the two contexts. It's important to keep this in mind as we look at how we can use the language to create testbenches. Now, let's take a look at a testbench for the calculator simulation. We'll start with the entity declaration.

```
entity testit1_vhd is end testit1_vhd;
```

Note that there is no port specification here, as the testbench has no inputs and outputs. So, the entity declaration just specifies the name of the entity. Here is the start of the architecture.

```
architecture a1 of testit1_vhd is
component calculator  port(
   clk : in std_logic;
   clear, load, add : in std_logic;
```

```
    dIn : in std_logic_vector(7 downto 0);
    result : out std_logic_vector(7 downto 0));
  end component;
```

This is a component declaration and it specifies the interface to the calculator component. This contains the same information that is present in the entity declaration for the calculator, but VHDL requires that every architecture include component declarations for all entities that they use. This makes the architecture more self-contained, making it easier for readers to understand the relationships among different parts of a complex design. Moving on, we have a series of signal declarations that include *initializers* that assign initial values to the testbench's internal signals.

```
  signal    clk: std_logic := '0';
  signal  clear: std_logic := '0';
  signal   load: std_logic := '0';
  signal    add: std_logic := '0';
  signal    dIn: std_logic_vector(7 downto 0)
              := (others=>'0');
  signal result: std_logic_vector(7 downto 0);
```

Let's move onto the body of the architecture.

```
  begin
    uut: calculator port map(clk, clear, load,
                             add, dIn, result);
```

The two lines following the `begin` contain a *component instantiation* statement. Its function is to *instantiate* a copy of the calculator circuit and connect the inputs and outputs of the calculator circuit to the testbench's local signals, `clk`, `clear`, `load`, `add`, `dIn` and `result`. Note that it's not necessary to use the same names for the local signals that one uses within the *calculator* circuit, but it is common to do so, as it reduces opportunities to make mistakes. The port map portion of the component instantiation statement lists the signals in the order that they are listed in the component declaration (and the entity declaration). There is an alternative way to write this, using explicitly named signals.

```
uut: calculator port map(clk=>clk, clear=>clear,
```

```
                        load=>load, add=>add,
                        dIn=>dIn, result=>result);
```

Here, the notation `x=>y` means that the component's signal `x` is equated to the local signal `y`. When we specify the port map in this way, the order in which the signal pairs are listed does not matter. Note also that the port instantiation statement starts with a label `uut`. This label is used by the simulator to identify this particular instantiation of the component. It is common to refer to the component being tested by a testbench as the *unit under test*, and this is what the label `uut` stands for. Now, let's move onto consider the first of two processes in the testbench.

```
process begin  -- clock process for clk
  clk_loop: loop
    clk <= '0'; wait for 10 ns;
    clk <= '1'; wait for 10 ns;
  end loop clk_loop;
end process;
```

This process specifies the clock signal `clk`, that controls the timing of operations within the calculator circuit. Note that it consists of an infinite loop in which `clk` alternates between high and low states, with a transition every 10 ns. The `wait` statements provide the delays needed to create the desired timing relationship. Now, let's look at the process that creates the other input signals to the calculator.

```
tb: process  begin
  clear <= '1'; load <= '1'; add <= '1';
    dIn <= x"ff"; wait for 20 ns;
  clear <= '0'; load <= '1'; add <= '0';
    dIn <= x"55"; wait for 20 ns;
  clear <= '0'; load <= '1'; add <= '1';
    dIn <= x"aa"; wait for 20 ns;
  clear <= '0'; load <= '0'; add <= '1';
    dIn <= x"55"; wait for 20 ns;
```

```
      ... -- omitting additional input vectors
      clear <= '0'; load <= '0'; add <= '1';
        dIn <= x"59"; wait for 20 ns;
      wait for 20 ns;

      assert (false)
        report "Simulation ended normally."
        severity failure;
    end process;
end;
```

The first pair of lines in the process specifies values for all the inputs to the calculator, and then the wait statement creates a pause of 20 ns before going onto the next set of assignments. So, each of pair of lines specifies one of the set of input values that we can observe in the waveform display that we saw at the start of this chapter. Notice that the two processes operate independently of one another, each generating its own set of signals. The pattern we see here, with one process for the clock, and another for the other inputs, is a very common one.

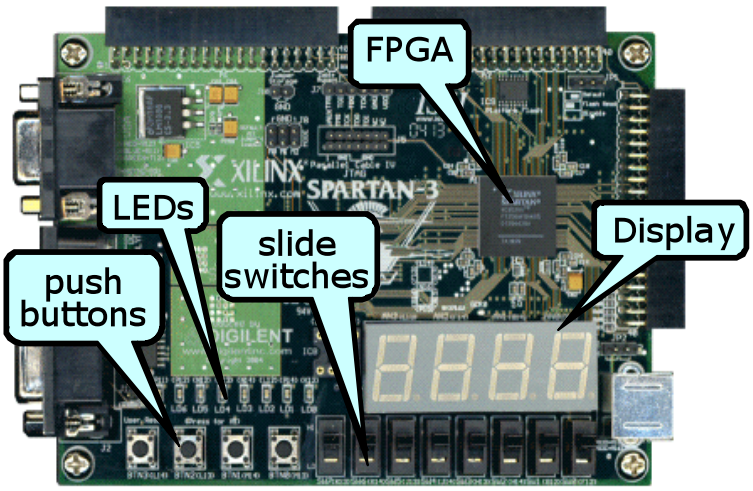
At the end of the second process, there is an `assert` statement. This is used to force the simulation to terminate. In general, an `assert` statement starts with an expression that evaluates to true-or-false. We typically use assert statements, as part of a circuit verification strategy, to state some condition that we expect to be true. Simulators evaluate these expressions and if they are false, display a message and optionally terminate the simulation. In particular, the simulation fails if the `severity` is specified to be `failure`. In this case, we are just using the assertion mechanism as a way of forcing the testbench to terminate, and this is stated explicitly in the message provided in the `assert` statement.

We do not discuss the specifics of individual simulation tools here, but there are some general things you should keep in mind when using simulation to verify a circuit. First, think carefully about what inputs are needed to exercise the functionality of the circuit. It's important to be as thorough as you reasonably can, without going overboard. In the calculator circuit, for example, we tested all the various control operations

and went through a handful of different addition operations. We made no attempt to test all pairs of operands for the addition operation, since the circuit that implements the addition operation is provided by the VHDL language system, and we can safely assume that it's correct. As we study more complex circuits in later chapters, we will provide some additional pointers on how to use simulations effectively.

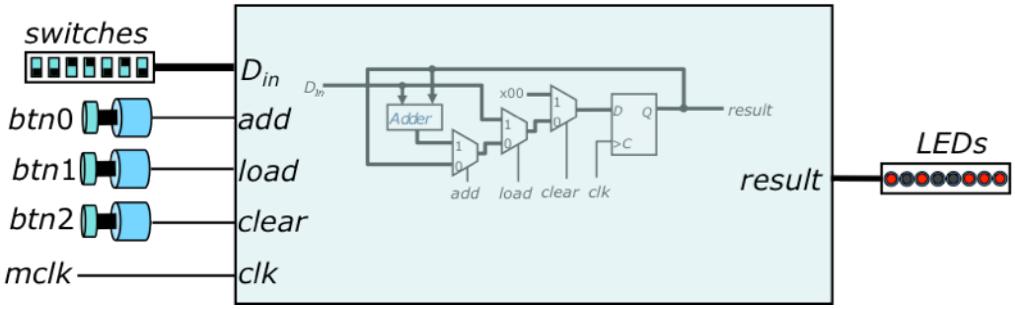
4. Prototyping Circuits with an FPGA

In this chapter, we discuss some of the issues that arise when prototyping digital circuits using a Field Programmable Gate Array. FPGAs are devices that can be configured to implement a wide variety of digital circuits. We won't discuss how FPGAs work in this chapter, but we will see how we can use them to for prototyping and testing. To make the discussion concrete, we will show how we can prototype and test our simple calculator circuit from Chapter 1, using a prototyping board made by Digilent, which is shown below.



This board contains a Xilinx Spartan-3 FPGA and we'll refer to the board as the S3 board for the remainder of this chapter. In addition to the FPGA, the S3 board has four push buttons and eight slide switches, that can be used to provide inputs to the FPGA circuit, plus eight Light Emitting Diodes (LED) and a four digit numeric display that can be used for

output. These external devices are wired to specific physical pins on the FPGA. By specifying connections between those pins and the inputs and outputs of our circuit, we can effectively connect those pins to our circuit. For our calculator, we will use the push buttons, slide switches and LEDs, as illustrated below.



As shown in the picture, we'll connect the slide switches to the data input signal and three of the push buttons to the three control inputs. The external signal `mclk` is a 50 MHz clock signal that comes from a clock generator circuit on the board and is connected to one of the input pins of the FPGA. We'll connect the calculator's `result` output to the LEDs on the board.

To establish the connections between the external pins and our calculator circuit, we will embed the calculator circuit in a top level circuit whose inputs and outputs are the external signals we'll be using. Here is the entity declaration of that top level circuit.

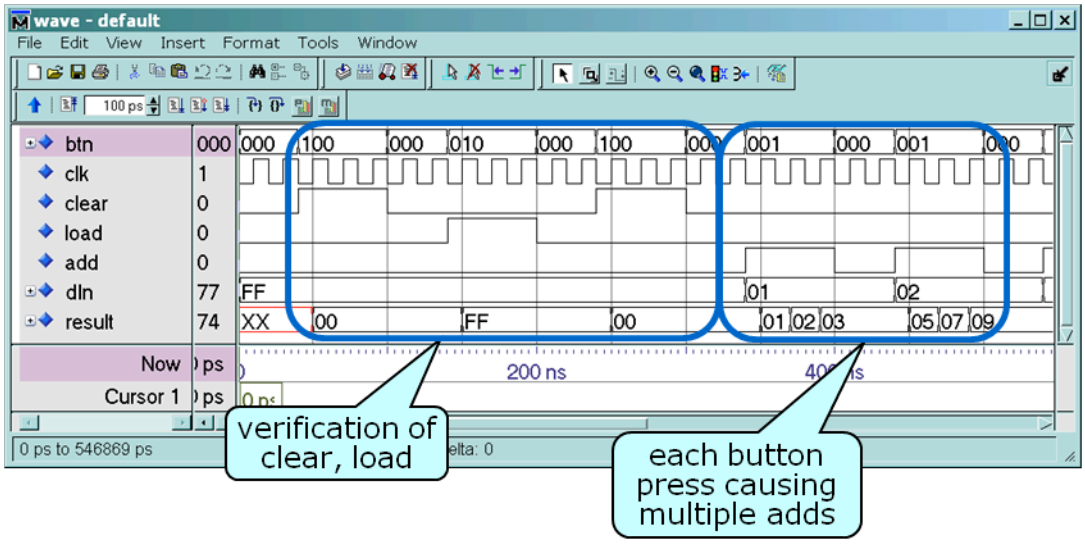
```
entity top is port(
   mclk: in STD_LOGIC;
   btn: in std_logic_vector(3 downto 0);
   swt : in std_logic_vector(7 downto 0);
    led : out std_logic_vector(7 downto 0));
end top;
```

We will embed a copy of our calculator circuit within top, by using structural VHDL, just as we did with our simulation testbench in the last chapter.

```
architecture topArch of top is
component calculator port (
   clk: in std_logic;
   clear, load, add: in std_logic;
   dIn: in std_logic_vector(7 downto 0);
   result: out std_logic_vector(7 downto 0));
end component;
signal clear, load, add: std_logic;
signal dIn: std_logic_vector(7 downto 0);
signal result: std_logic_vector(7 downto 0);
begin
   clear <= btn(3); load <= btn(2); add <= btn(1);
   dIn <= swt;  led <= result;
   calc: calculator port map(mclk,clear,load,
                             add,dIn,result);
end topArch;
```

Note that the architecture must include a component declaration for the calculator, and that we define a set of appropriately named internal signals which we associate with the corresponding external signals. The component instantiation statement identifies these signals with the appropriate inputs and outputs of the calculator. Note that these local signals are not strictly necessary, but they do make the purpose of the various signals a bit more obvious.

Now, let's simulate this version of our circuit to verify that it still works as expected.



Notice that the three buttons we are using are shown in the top line of the waveform display, with `btn(3)` on the left, followed by `btn(2)` and `btn(1)`. The corresponding internal control signals (`clear`, `load`, `add`) are shown below the button signals on separate lines. Note that when we prototype the circuit on the S3 board, buttons will be held down for many clock ticks, since a clock tick is just 20 ns long and even a very quick button press will last 100 ms or more. To produce a similar effect, our testbench for the simulation specifies that the button signals be held for several clock ticks, each time one is pressed. This can be seen clearly in the waveform display.

The first part of the simulation seems to be working correctly, but when we come to the portion that tests the addition operation, we observe something unexpected. The single button press results in several addition operations taking place, not just one. This happens because the calculator performs a new operation on every clock tick. So, if the add signal is held high for several clock ticks in a row, the calculator performs an addition on every clock tick. But this means that when we try to test the circuit on

the S3 board, every button press will cause millions of addition operations to be performed, which is clearly not what we want.

To get the desired behavior, we're going to need to modify the circuit. In particular, we're going to need to generate an internal signal that goes high for a single clock tick every time the button is pressed. This can be done by detecting a 0-1 transition on the button signal and using that transition to create the desired pulse. We can do this by modifying the architecture of our `top` circuit.

```
architecture topArch of top is
component calculator ... end component;
signal clear, load, add: std_logic;
signal dIn: std_logic_vector(7 downto 0);
signal result: std_logic_vector(7 downto 0);
signal prevBtn: std_logic_vector(3 downto 0);
begin
   -- generate internal signals for button pushes
   process (mclk) begin
      if rising_edge(mclk) then
         prevBtn <= btn;
      end if;
   end process;
   clear <= btn(3) and (not prevBtn(3));
   load <= btn(2) and (not prevBtn(2));
   add <= btn(1) and (not prevBtn(1));
   dIn <= swt;  led <= result;
   calc: calculator port map(mclk, clear, load,
                             add, dIn, result);
end topArch;
```

The signal `prevBtn` is used to create a delayed version of the button inputs. The process does this, by assigning the `btn` to `prevBtn` within the scope of a synchronization condition. Recall that this implies that `prevBtn` will be implemented using a clocked register, whose input is connected to `btn`. Following the process, we see that each of the control signals is generated using one of the `btn` inputs together with the corresponding value of `prevBtn`. This causes the control signal to be high

for one clock tick whenever there is a 0-to-1 transition on `btn`. Note that we could have left the `clear` and `load` inputs alone, since it does no real harm if these operations are performed multiple times per button press. However, for consistency, we have chosen to implement all the control signals in the same way.

If we re-run the simulation with our modified `top` circuit, we'll observe that each button press does in fact cause the corresponding control signal to be high for one clock tick, and as a result we get a single addition operation every time we press the button. We're now ready to try out our circuit on the S3 board. In order to do this, we must create an FPGA *bitfile* containing the information needed to configure the FPGA so that it implements our calculator circuit. The bitfile must then be downloaded to the FPGA using a special programming cable that is provided for this purpose. We will not discuss the details of how this is done, since those details depend on the specific CAD tool set that you are using.

Now, when we configure the FPGA to implement our circuit, we will observe something curious. When we use the buttons and switches on the board to test the various operations, we observe that while the clear and load operations work correctly, the add operation works correctly only some of the time. In particular, it appears that some of our button presses are producing multiple addition operations. This seems a bit strange. What's going on? The answer to this question has to do with the fact that the mechanical buttons on the S3 board vibrate as we press and release them, and these tiny vibrations can cause the button signals to make several 0-1 transitions every time we press down on the button. Most mechanical buttons behave this way, so we need to make some additional changes to our circuit to deal with this. Specifically, we need to add a *debouncer* and connect the buttons to the internal logic through this debouncer.

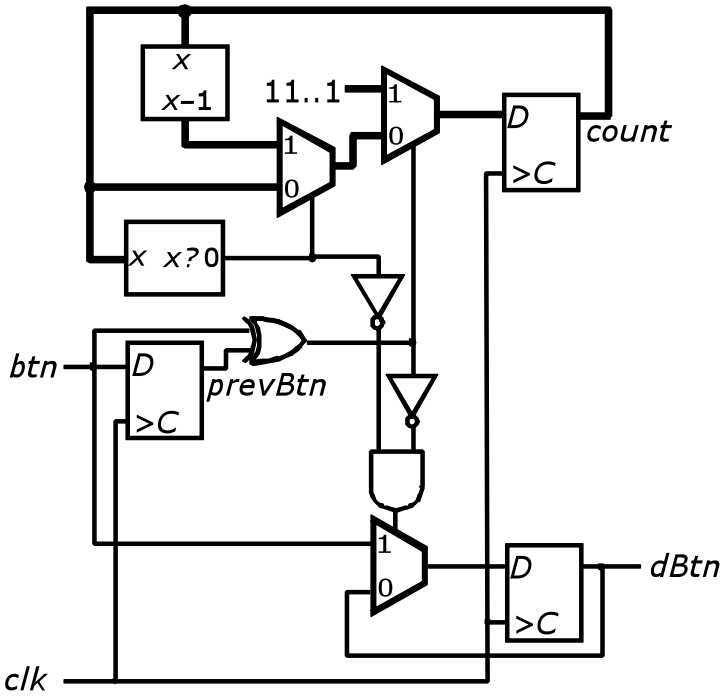
A debouncer filters out the fluctuations cause by the mechanical vibrations of a button. It does this by delaying the response to a transition until the button signal has been stable for a long enough time that we can be confident that the button is no longer bouncing. Here is the entity declaration for the debouncer. Its inputs are the clock and the `btn` signals. Its output `dBtn` is a debounced version of `btn`.

```
entity debouncer is port(
   clk: in std_logic;
   btn: in std_logic_vector(3 downto 0);
   dBtn: out std_logic_vector(3 downto 0));
end debouncer;
```

Now, let's look at the architecture.

```
architecture debArch of debouncer is
signal prevBtn: std_logic_vector(3 downto 0);
signal count: std_logic_vector(19 downto 0);
begin
   process(clk) begin
      if rising_edge(clk) then
         prevBtn <= btn;
         if prevBtn /= btn then
            count <= (others => '1');
         elsif count /= (count'range => '0') then
            count <= count - 1;
         else dBtn <= btn;
         end if;
      end if;
   end process;
end debArch;
```

The architecture includes two internal signals `prevBtn` and `count`. `PrevBtn` is used to create a delayed version of `btn` and `count` is used to determine when the `btn` signals have been stable for a long enough time that it's safe to allow them to propagate to the output, `dBtn`. Notice that every time `btn` changes, `count` is set to all 1's. For every clock tick when `btn` does not change and `count` is not equal to zero, `count` is decremented by 1. On any clock tick where `count` is equal to zero, the value of `btn` is assigned to `dBtn`, meaning that the register that implements `dBtn` is loaded from `btn`. Here is a diagram of a circuit that implements the VHDL specification.



Because the count signal is defined to be 20 bits long, the debouncer will count down for 2^{20} clock ticks following a transition on `btn`, before reflecting that transition at the outputs. Since 2^{20} is slightly more than one million and since the period of the clock on the S3 board is 20 ns, this means that the circuit implements a delay of just over 20 ms, which is long enough to filter out any spurious transitions due to vibration of the button mechanism.

There is one detail of the VHDL code that bears further explanation. The line

```
elsif count /= (count'range => '0') then
```

tests if count is equal to zero. The right side of the inequality includes the expression `count'range`. In this expression, `range` is an *attribute* of the signal count. Specifically, it refers to the range of indices in the signal vector, so here, `count'range` is 19 downto 0. We could have written

```
elsif count /= (19 downto 0 => '0') then
```

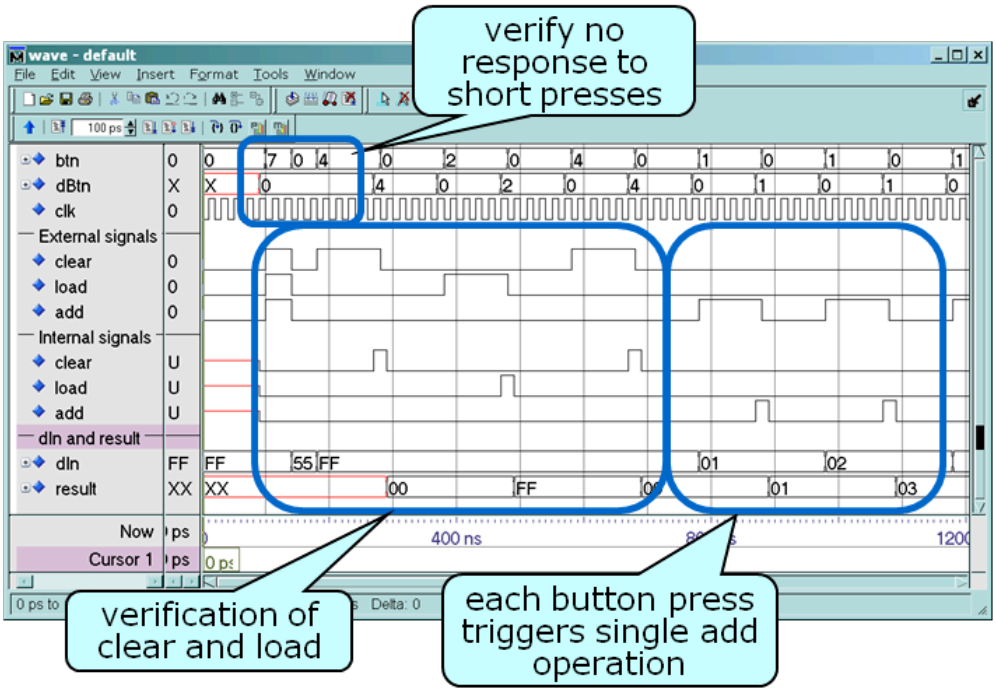
but we chose to write it the way we did to make the expression independent of the actual length of `count`. In general, it's a good practice to make our VHDL code independent of the actual lengths of signal vectors, but in this case we had a very specific reason for doing so. That reason has to do with simulating the circuit. If we were to simulate the debouncer in order to verify its operation, we would have to wait for more than a million clock cycles to observe the propagation of signals through the debouncer. To avoid this, we can modify the circuit for the purposes of simulation, by reducing the length of `count` from 20 bits to 2 or 3 bits. Since the architecture is independent of the length of `count`, the only thing we need to change in order to do this, is the declaration of `count`.

Here's a new version of our top circuit that uses the `deBouncer` and the `calculator` circuits.

```
architecture topArch of top is
component calculator ... end component;
component debouncer ... end component;
signal clear, load, add: std_logic;
signal dIn: std_logic_vector(7 downto 0);
signal result: std_logic_vector(7 downto 0);
signal dBtn, prevDB: std_logic_vector(3 downto 0);
begin
   -- debounce buttons
   dbnc: debouncer port map(mclk, btn, dBtn);
   -- generate pulses when debounced signals
   -- go high
   process (mclk) begin
      if rising_edge(mclk) then
         prevDB <= dBtn;
      end if;
   end process;
   clear <= dBtn(3) and (not prevDB(3));
```

```
    load <= dBtn(2) and (not prevDB(2));
    add <= dBtn(1) and (not prevDB(1));

    -- connect signals to the calculator
    dIn <= swt;  led <= result;
    calc: calculator port map(mclk, clear, load,
                              add, dIn, result);
end topArch;
```

The internal signal `dBtn` is the debounced version of `btn`. The top circuit creates a delayed version of `dBtn` (called `prevDB`) and uses it to generate the internal control signals. These are then connected to the calculator circuit. Now, let's look at a simulation of this circuit.



Before we look closely at the actual simulation output, it worth taking a few moments to notice a few things about the waveform display. At the left, where the signal names are listed, the signals have been organized into groups, with labels added to identify each related group of signals.

Most circuit simulators allow you to order the signals in the way that's most convenient for you, and to add labels. It's worth taking the time when doing a simulation, to organize the symbols in some logical fashion and add labels to help you keep track of things. Once you have done this one time, the simulator will typically allow you to save the layout of your waveform display to a file, so that you can re-use it on subsequent simulation runs. When testing a complex circuit, we may need to make a sequence of modifications to our circuit, in order to get it to work correctly. Spending a little time getting your simulation set up properly at the start, can save you a lot of time later, as you go through the simulate-modify-and-simulate-again cycle.

Now, let's take a closer look at the actual signals. The group of signals labeled "External Signals" are just the original button signals (notice how they change when the `btn` signals change). We've relabeled them in the simulator to make it easier to see the connection with the internal control signals that appear below. For this simulation, we have reduced the debouncer's counter from 20 bits to 2 bits. This means that the `btn` signals propagate through the debouncer after they have been stable for four clock ticks. Notice that changes to the `btn` signals that are not stable for four ticks, never get propagated. Looking ahead, we can see that all the control operations, including the `add` operation, work as expected.

Of course, our previous simulation worked as expected too, and we only discovered the need for the debouncer, when we tried to check out the circuit on the S3 board. So, in order to verify that we really have it right, we need to generate a bitfile for the modified circuit and download it to the S3 board. Before doing this, we need to change the length of the count signal in the `deBouncer` from 2 bits back to 20 bits. If we do this, and load the resulting circuit on the S3 board, we will find that our calculator now does work correctly, with every button press triggering one (and only one) operation.

5. Additional Language Features

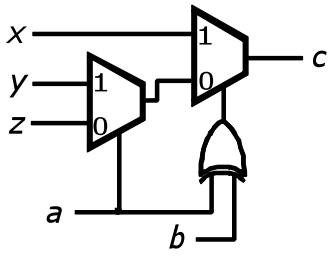
In this chapter, we introduce a number of additional features of the VHDL language.

More Signal Assignments

As we have seen, we can define any combinational logic circuit using just simple signal assignments, but VHDL also provides two additional types of assignments that often allow us to express a circuit design in a more convenient way. The first of these is the *conditional signal assignment*, which allows us to make the value assigned to a signal dependent on a series of conditions. Here's an example.

```
c <= x when a /= b else
     y when a = '1' else
     z;
```

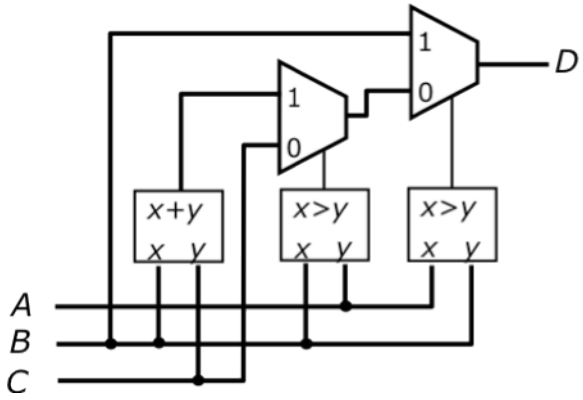
This assignment can be implemented by the following circuit.



Note the similarity between this and the circuit that implements the if-then-elsif statement in the calculator circuit from Chapter 2. The conditional signal assignment can have any number of `when-else` clauses, and each successive clause can be implemented using another multiplexor. We can also use the conditional signal assignments with signal vectors. If `A`, `B` and `C` are all 8 bit signal vectors, then the assignment

```
D <= B when A > B else
     B+C when A < B else
     C;
```

is equivalent to the circuit

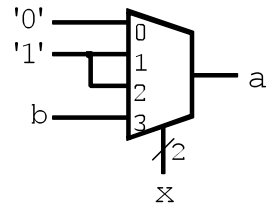


Here the labeled blocks designate combinational sub-circuits that implement an inequality comparison function and an addition function.

There is one more type of signal assignment in VHDL called the *selected signal assignment*. For example, if "x" is a signal vector with 2 bits, then

```
with x select
   a <=    '0' when "00",
         '1' when "01" | "10",
         b when others;
```

specifies the circuit



This diagram includes a 4:1 multiplexor, which has 4 data inputs and a 2 bit control input. It's output is equal to the data input whose index is specified by the control value. That is, if the value on the control input is 10, the output is equal to the value on data input 2. VHDL allows the conditional and selected signal assignments to be used only outside of process blocks.

Case Statement

VHDL provides a case statement that is useful for specifying different results based on the value of a single signal. For example,

```
architecture a1 of foo is begin
  process(c,d,e) begin
    b <= '1'; -- provide default value for b
    case e is
    when "00" => a <= c; b <= d;
    when "01" => a <= d; b <= c;
    when "10" => a <= c xor d;
    when others => a <= '0';
    end case;
  end process;
end a1;
```

This circuit can be implemented using decoder that converts the two bit signal `e` into four 1 bit signals. These can then be used to select which values propagate to the outputs `a`, and `b`. We could have also written this

using an if-elsif statement, but the resulting circuit would implemented using a series of 2:1 multiplexors (as would be typical), it would be slower than the one synthesized for the case-statement.

For Loops

VHDL provides a for-loop which is similar to the looping constructs in sequential programming languages. We can use it to define repetitive circuits, like the adder circuit shown below.

```
entity adder is
   port(A, B: in std_logic_vector(15 downto 0);
   Ci: in std_logic;
   S: out std_logic_vector(15 downto 0);
   Co: out std_logic );
end adder;
architecture a1 of adder is
signal C: std_logic_vector(16 downto 0);
begin
   process (A,B,C,Ci) begin
      C(0) <= Ci;
      for i in 0 to 15 loop
         S(i) <= A(i) xor B(i) xor C(i);
         C(i+1) <= (A(i) and B(i)) or
                   ((A(i) xor B(i)) and C(i));
      end loop;
      Co <= C(wordSize);
   end process;
end a1;
```

Note that the for-loop does not imply the repetitive execution of program statements. Rather, it signifies the repeated instantiation of digital circuits. The example for-loop is equivalent to `wordSize` pairs of assignments, each of which specifies the circuitry for one bit position of the adder.

You might wonder why we used a logic vector for the signal `C` that implements the carry signals between successive bit positions. Wouldn't it be simpler to write

```
architecture a1 of adder is
signal C: std_logic;
begin
   process (A,B,C,Ci) begin
      C <= Ci;
      for i in 0 to 15 loop
         S(i) <= A(i) xor B(i) xor C;
```

```
        C <= (A(i) and B(i)) or
             ((A(i) xor B(i)) and C);
      end loop;
      Co <= C;
    end process;
end a1;
```

While this would make perfect sense in a sequential programming language, it does not have the intended meaning in VHDL, since here, we're specifying circuits, not sequential execution. The signal `C` can only be 'wired' in one way. It cannot be re-defined by different expressions at different times. Consequently, the circuit defined by this specification will not behave as intended.

It's instructive to look at the circuit that would be produced by this second VHDL specification. Assume for the moment that we've modified the VHDL to implement a 3 bit adder, instead of a 16 bit adder; then the for-loop would be equivalent to the following sequence of statements.

```
S(0) <= A(0) xor B(0) xor C;
C <= (A(0) and B(0)) or ((A(0) xor B(0)) and C);
S(1) <= A(1) xor B(1) xor C;
C <= (A(1) and B(1)) or ((A(1) xor B(1)) and C);
S(0) <= A(2) xor B(2) xor C;
C <= (A(2) and B(2)) or ((A(2) xor B(2)) and C);
```

Note that there are three assignments to the signal `C`, but only one of these can be used to actually implement the circuit for `C`. While VHDL allows us to write these three assignments for `C` (when inside a process block), all but the last of them are ignored. What this means is that the above sequence of statements is actually equivalent to this sequence.

```
S(0) <= A(0) xor B(0) xor C;
S(1) <= A(1) xor B(1) xor C;
S(2) <= A(2) xor B(2) xor C;
C <= (A(2) and B(2)) or ((A(2) xor B(2)) and C);
```

Now, if these were assignments in a sequential programming language, you might object that `C` is not defined in the first three assignments. But again, the semantics of assignment in VHDL are not the same as the

semantics of sequential execution. In VHDL, the assignments simply define how the circuit elements are wired together. It's also worth observing that the circuit defined by these statements contains a cycle, since `C` depends on itself. We will see later that there is another way to write the for-loop using a VHDL *variable*, that does not require a vector of signals to implement the carries. However, we will defer the discussion of variables for now.

User-Defined Types

Like conventional programming languages, VHDL allows users to define their own types. To start with, let's look at how constants are defined in VHDL.

```
constant wordSize: integer := 16;
```

This declaration declares `wordSize` to be an integer constant with a value of 16. Given such a constant declaration, we can write

```
signal dataReg: std_logic_vector(wordSize-1 downto
0);
```

By associating a name with a constant value, we can make it easier for someone else reading our VHDL code to understand our intent. Moreover, if we use `wordSize` consistently in our code, we can easily modify our circuit specification to accommodate a different value for `wordSize` at some point in the future. Instead of changing possibly dozens of individual numerical constants, we can just change a single constant declaration.

VHDL also allows us to declare new signal types. For example, we can define a register type using the declaration

```
type regType is std_logic_vector(wordSize-1 downto
0);
```

and then use it to define signals of this type.

```
signal regA, regB: regType;
```

VHDL also supports *enumeration types*. The declaration

```
type color is (red, green, blue, black, white);
```

declares color to be a type that takes on five named values. Signals of type color might be implemented as a three bit vector, with specific bit combinations selected for the different values. VHDL also allows one to specify subtypes of an enumeration. For example,

```
subtype primary is color range red to blue;
```

defines `primary` to take on the values `red`, `green` and `blue`. With these definitions, we can define the following signals

```
signal c1, c2, c3: color;
signal p: primary
```

and write statements like the following.

```
c1 <= red;
p <= green;
if c2 /= green then
   c3 <= p;
else
   c3 <= blue;
   p <= blue;
end;
```

The language allows assignments from a signal of type `primary` to a signal of type `color`, but not assignments from `color` to `primary`.

VHDL also allows us to define more complex signal types. For example, the declarations

```
type regFileType is array(0 to 15) of regType;
signal reg: regFileType;
```

defines `regFileType` to be an array of items, each of type `regType`, and declares `reg` to be a signal of this type. Given these declarations, we can write

```
reg(2) <= x"3fe5";
reg(3) <= reg(5) + reg(7);
reg(8 to 15) <= (9 => x"abcd", 11 => x"ffff",
                 others => x"0000");
```

```
reg(4)(2) <= '1';
```

Note that we can think of the signal `reg` as defining a two dimensional array. VHDL allows multidimensional arrays to be declared more directly, but unfortunately, most circuit synthesizers cannot generate circuits based on multidimensional array declarations, although they can generate circuits based on the declarations given above. This situation may well change in the future, but to be consistent with the current state of circuit synthesis tools, we limit ourselves to one-dimensional arrays here.

A signal of type `regFileType` may be implemented in one of several different ways, depending on how it is used. The most general implementation is a collection of separate registers, each of which is implemented using flip flops. We'll discuss later how synthesizers can sometimes implement signals of this type using a Random Access Memory (RAM) or Read-Only Memory (ROM).

Type declarations can also be used to declare composite signals that combine components of different types. For example

```
type item is record
   dp: std_logic;
   key, value: regType;
end record item;
signal i1, i2, i3: item;
```

declares signals of type `item` to contain a single bit signal called `dp`, and two signals, `key` and `value`, of type `regType`. Given these declarations we can write

```
i1 <= (dp => '1', key => x"0000", value=> x"ffff");
i2.value <= i1.key(15 downto 8) & x"3d"
i3 <= i1;
```

We can also define a new type that is an array of type item.

```
type itemVecTyp is array(0 to 15) of item;
signal ivec: itemVecTyp;
```

allowing us to write statements like

```
ivec(1) <= ivec(4); ivec(2).key <= ivec(3).value
```

Later, we'll see examples of how record types can be used effectively in complex circuits to organize related data elements and make circuit specifications that are easier to understand.

Signal Attributes

In the last chapter, we saw an example of the use of the range attribute of a signal. In general, if `x` is a `std_logic_vector`, then `x'range` is the range of indexes for which `x` was was defined. For example, the declaration

```
signal x std_logic_vector(0 to 5);
signal y std_logic_vector(8 to 3);
```

means that `x'range` is `0 to 5`. VHDL defines a number of other attributes for signal vectors. In particular `y'left` refers to the "left endpoint" of the index range for `y` (`8`, in this case), while `y'right` refers to the right endpoint (`3`). Similarly `y'low` refers to the smaller of the two index values (3) defining the range, while `y'high` refers to the larger value (8); `x'length` is the number of distinct indexes (6) and `x'ascending` is true if the index range was defined so that the indices increase in value from left to right (`true` for `x`, `false` for `y`).

Structural VHDL

Structural VHDL refers to a style of VHDL coding in which circuits are defined in a very explicit way by specifying the connections among different components. The key element of structural VHDL is the *component instantiation statement* that we introduced in Chapter 2. Structural VHDL is best used sparingly to connect larger circuit elements that are themselves defined using processes and the various high level control statements that we have discussed. Our use of structural VHDL in Chapter 3, to combine the calculator circuit with the debouncer is typical

of this kind of usage. However, structural VHDL can be used to construct smaller sub-circuits as well. We can illustrate this by constructing a 4 bit adder using the full adder module defined earlier as a building block.

```
entity adder4 is port(
   A, B: in std_logic_vector(3 downto 0);
   Ci: in std_logic;
   S: out std_logic_vector(3 downto 0);
   Co: out std_logic);
end adder4;
architecture a1 of adder4 is
component fullAdder port(
   A, B, Ci: in std_logic;
   S, Co: out std_logic );
end component;
signal C: std_logic_vector(3 downto 1);
begin
   b0:fullAdder port map(A(0),B(0),Ci,S(0),C(1));
   b1:fullAdder port map(A(1),B(1),C(1),S(1),C(2));
   b2:fullAdder port map(A(2),B(2),C(2),S(2),C(3));
   b3:fullAdder port map(A(3),B(3),C(3),S(3),Co);
end a1;
```

This architecture instantiates four copies of the full adder component and specifies how the copies are connected to the inputs and outputs of the top level circuit and how they are connected to each other through the carry signal. Each component instantiation statement has a label that is used to distinguish the components from one another. The port map portion of the statements uses positional association of the ports. That is, the position of a signal in the port map list determines which signal in the component declaration it is associated with. VHDL also allows named association. For example, we could write

```
b0: fullAdder port map(A=>A(0),B=>B(0),S=>S(0),
                       Ci=>Ci,C0=>C(1));
```

Note that if we use named association, the order in which the arguments appear does not matter. For larger circuit blocks with many inputs and outputs, named association is preferred.

Structural VHDL also supports iterative definitions so that we need not write a whole series of similar component instantiation statements. This allows us to write a 16 bit adder as

```
architecture a1 of adder16 is
component fullAdder port(
   A, B, Ci: in std_logic;
   S, Co: out std_logic );
end component;
signal C: std_logic_vector(16 downto 0);
begin
   C(0) <= Ci;
   bg:for i in 0 to 15 generate
        b:fulladder
           port map(A(i),B(i),C(i),S(i),C(i+1));
     end generate;
   Co <= C(16);
end a1;
```

Observe that in this version, we've declared `C` to be a 17 bit signal, and equated `Ci` with `C(0)` and `Co` with `C(16)`. This avoids the need for separate component instantiation statements for the first and last bits of the adder. Note that the labels on the for-generate statement and on the component instantiation statement are both required.

We note that it is possible to define circuits entirely using structural VHDL, with AND gates, OR gates and inverters as basic primitives used to construct larger components. However, this style of VHDL is tedious, error prone and produces circuits that are difficult for others to understand. Essentially, this amounts to using VHDL as a textual way of specifying a schematic diagram, which defeats the purpose of using a tool like VHDL in the first place. If one wants to design at the schematic level, it makes more sense to use a graphical editor, rather than a language like VHDL. Having said that, structural VHDL does give the designer a great

deal of control over the circuit that is generated by a circuit synthesizer, and there are times when it may make sense to use it for small parts of a design that merit special treatment.

The Separation Principle

It's very easy to allow our experience with sequential programming languages to lead us to misunderstand the meaning of a VHDL circuit specification. This is why we've been highlighting the differences between the two, in all of our discussions of the language. The most important single thing to keep in mind is that VHDL is used to define connections among circuit components, not the sequential execution of program statements. There is a useful conceptual tool that you can often use to help you understand the circuitry that is being defined by a given set of statements. Consider the following code fragment that defines several signals.

```
x <= x"0000";
y <= x"abcd";
if a = b then
   x <= y; z <= b;
elsif a > c then
   y <= b; z <= a;
else
   z <= x + y;
end if;
```

Here is an equivalent code fragment in which the assignments to `x`, `y` and `z` are separated from each other.

```
-- code segment defining x
x <= x"0000";
if a = b then  x <= y; end if;
-- code segment defining y
```

```
y <= x"abcd";
if a /= b and a > c then y <= b; end if;
-- code segment defining z
if a = b then z <= b;
elsif a > c then z <= a;
else   z <= x + y;
end if;
```

This is an example of the *separation principle*, which states that we can separate the assignments to different signals without affecting the meaning of the VHDL (where the 'meaning' is the circuit that implements the VHDL). Note that the three code segments above can be re-arranged in any order, without changing the meaning, although within each code segment, the order of statements does matter. For example if we swap the two statements that assign values to `x`, we are defining a different circuit than is defined by the original version.

Underlying the separation principle is the fact that a VHDL circuit specification is ultimately just a way to define signals, in terms of other signals using logic equations. Because VHDL specifies the "wiring" of the circuit, rather than sequential execution, the order in which different signals are defined does not matter, any more than the way that we draw the components in a schematic diagram does; so long as we define the same connections among components, the placement of the symbols for those components in a drawing doesn't matter. In principle, we could specify each signal using a single simple signal assignment, but higher level constructs like if-statements, case statements and for-loops allow us to state our intentions more concisely, and to organize the various elements of the circuit in a way that makes the overall design easier to comprehend. However we express the design, the language statements must eventually be reduced to a set of gates and the wires that connect them.

6. Digital Circuit Elements

When we design circuits using VHDL, we are specifying a circuit consisting of elementary components like gates and flip flops that are connected to one another as needed to implement our circuit specification. In this chapter, we take a closer look at these basic building blocks, as well as some larger components that can be constructed using the basic building blocks.

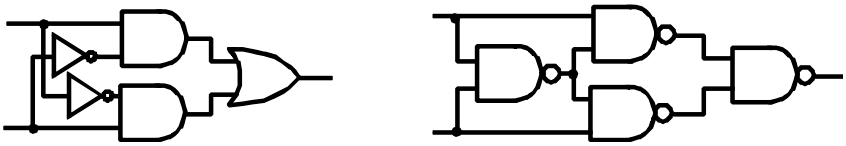
Basic components

We typically think of components like AND gates. OR gates and inverters as the basic elements of a digital logic circuit, but it turns out that gates are actually constructed from even more basic elements called *transistors*. Transistors can be used essentially as on-off switches that can connect a gate output to a high voltage (to produce a logic 1) or to a low voltage to produce a logic 0. In CMOS circuit technology (which is the dominant technology in widespread use today), we can implement an inverter with two transistors, and a 2 input AND gate (or an OR gate) with six transistors. Gates with more than two inputs can be constructed using additional transistors, with a three input gate using eight transistors, a four input gate using ten transistors, and so on.

A NAND gate is an AND gate in which the output has been inverted. Similarly, a NOR gate is an OR gate in which the output has been inverted. The symbols for NAND and NOR gates are shown below.

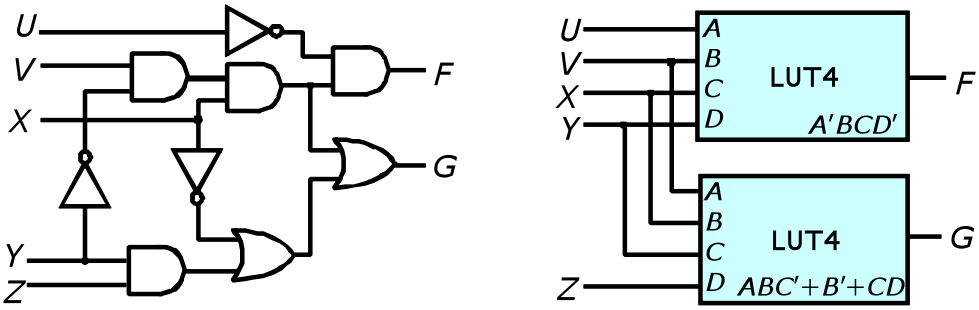
You might wonder, why we would bother to define these alternate types of gates, but it turns out that in CMOS technology, NAND and NOR gates are actually simpler to construct than AND and OR gates. We can implement a 2 input NAND gate using just four transistors and we actually implement an AND gate by adding an inverter to the output of a NAND.

The exclusive-or operation occurs frequently enough in circuits, that it's convenient to define a special gate symbol for it, as we have seen in earlier chapters. An XOR gate can be implemented using either of the circuits shown below.

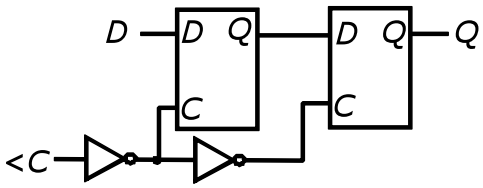


The one on the left corresponds directly to the definition of the exclusive-or operation, but the one on the right is significantly more efficient, using just 16 transistors, vs. 22 for the one on the left.

FPGAs implement logic using configurable components called lookup tables (LUT). A typical LUT has four inputs and can be configured to implement any 4 input logic function. A combinational circuit with more than four inputs can be implemented using several LUTs, appropriately configured and interconnected. An example is shown below.

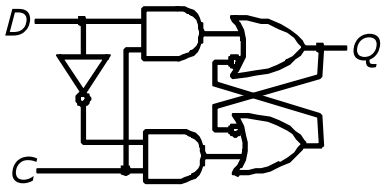


As we have seen in earlier chapters, flip flops and latches can be used to store data in digital circuits. An edge-triggered D flip flop can be implemented using a pair of D-latches, as illustrated below.



When the clock input is low, the first latch is in the "transparent" state, so its output follows the input signal. When the clock makes its transition from low to high, the first latch stores the value that was on its input at the time of the transition and the second latch becomes transparent, reflecting the value that was stored by the first latch, on its output. When the clock goes low again, the second latch becomes opaque, so it will continue to hold this value, ignoring changes on its input until the next clock transition. Thus, the flip flop captures the input state at the time of the rising clock transition, but ignores changes on the input at other times. It's worth noting that if we omit the first inverter on the clock input, we get a flip flop that is triggered by the falling clock edge, rather than the rising edge.

Latches can be implemented using gates, as shown below.



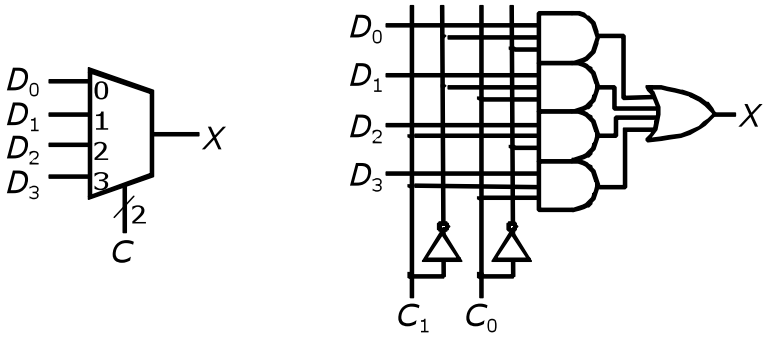
The cross-coupled gates with the single inverted input implement the storage function of the latch. When the control input is high, the latch is transparent. That is, a 1 on the *D* input will appear as a 1 on the *Q* output, and a 0 on the *D* input will appear as a 0 on the *Q* output. When the control input goes from high to low, the pair of OR gates become isolated from the *D* input, so they behave like a pair of cross-coupled inverters. This ensures that the *Q* output remains unchanged until the control input goes high again. If we replace all of the gates in the above circuit with NAND gates, we get a circuit that behaves exactly the same way. Consequently, we can see that a *D* latch can be implemented with 18 transistors and a *D* flip flop can be implemented with 40.

Larger building blocks

We have already seen how multiplexors can be used to implement the logic defined by if statements and selected signal assignments. The output of the 2:1 mux can be expressed by the logic equation

$$X = C' \cdot D_0 + C \cdot D_1$$

where C is the control input and D_0 and D_1 are the two data inputs. Thus it can be implemented with three gates and an inverter. It can also be implemented with a single 4 input LUT. We can implement a 4:1 mux using three 2:1 muxes, or we can implement it directly with gates, as shown below.

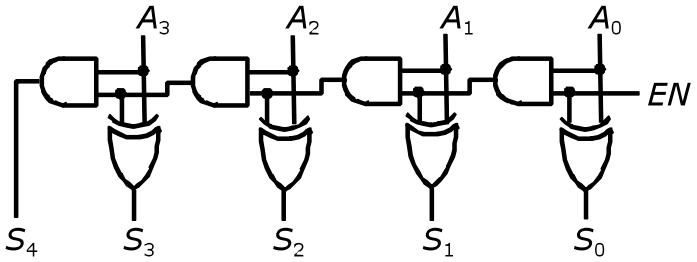


this circuit can be generalized to implement muxes with 8, 16 or more inputs. A *demultiplexor* performs the inverse function of the mux. It has a single data input, k control inputs and 2^k data outputs. If the value represented by the control signals is i, then output D_i is equal to the data input, while all other data outputs are 0.

If we remove all the data inputs and the OR-gate from the circuit for the 4:1 mux above, we get a 2→4 *decoder*. This circuit has four outputs, exactly one of which is high at all times. The output that is high is specified by the binary value on the inputs. We can think of this circuit as converting a 2 bit binary value to a 4 bit unary value. Larger decoders (say 3→8 or 4→16) can be constructed in a similar way. Decoders can be used to implement case statements. If the control signal for the case statement is connected to the inputs of a decoder, its outputs can be used to control the sub-circuits associated with each of the individual cases.

An *encoder* implements the inverse function of a decoder. For example, a 4→2 encoder converts a 4 bit unary input value to a 2 bit binary output value. That is if input D_i is high and the other three inputs are low, the binary value represented by the outputs will be i. In a *priority encoder*, the binary value on the outputs equals the index of the *first* of the inputs that is high. A special *valid* output is also usually provided for a priority encoder. If any of the encoders data inputs is high, the valid output will be high; otherwise, it will be low.

An i*ncrement circuit* takes an n-bit data input and produces an (n+1)-bit data output that has a numerical value that is one larger than the input value. An example of a 4 bit incrementer, with an *enable* (EN) input is shown below.



When the enable is low, the output is also low, but when the enable is high, the output is one larger than the input. The series of AND gates that pass from right to left form the *carry-chain* for the circuit. The carry into a given bit position is high, if the enable is high and all lower order bits are high. When the carry into a given bit is high, the output for that position is the complement of its input bit. This circuit is referred to as a ripple-carry incrementer, since the carry must propagate through all bit positions.

Circuit cost/complexity

Because circuits are physical devices, the number of elementary components in a circuit is directly related its cost. Consequently, we will often try to find circuit implementations that use as few elementary components as possible, in order to minimize cost. The most basic components of a circuit are gates and to allow uniform comparisons of different circuit designs, we will often restrict ourselves to *simple gates*, that is, gates with at most two inputs. If we are implementing circuits using FPGAs, it typically makes more sense to count LUTs and flip flops, rather than gates.

Often, we design circuits that have configurable sizes. For example, an increment circuit with n inputs or a multiplexor with k control inputs and 2^k data inputs. In cases like this, we'll express the circuit cost as a function of the "size parameter". So for example, the ripple-carry increment circuit discussed above requires $2n$ simple gates to handle n bit data, if we count the exclusive-or gates as one gate each. This is not exactly accurate, since an implementation of the XOR is more costly than a single AND or OR

gate. To be more precise, we could count each XOR gate as several "gate equivalents" or count transistors, rather than just gates. While this can give us more precision, in most situations, the added precision is of limited value, so we will often rely on simpler methods, such as counting all gates (including XOR gates) as roughly equivalent.

One of the reasons, we can get away with a bit of sloppiness in our accounting is that we're typically more interested in the rate at which our circuits grow in cost as the size parameter increases. For example, the ripple-carry increment circuit is particularly attractive because its cost grows in direct proportion to the number of bits. There are alternate designs for increment circuits that have a cost that grows in proportion to $n \lg n$. where lg denotes the base-2 logarithm. Such circuits are more expensive, but have the advantage of producing output results more quickly when n gets large.

When implementing a circuit using an FPGA, it makes more sense to count LUTs than gates. CAD tools will report the number of LUTs used to implement a given circuit and it's worth paying close attention to these numbers, at least in situation where cost is an important consideration. It's also worthwhile to be able to estimate the number of LUTs that a given circuit may require, so that we can get an idea of a circuit's cost before we go to the trouble of designing it in detail. Estimating the number of LUTs needed to implement a given circuit can be tricky and we won't discuss it in detail here, but we do note a useful few rules of thumb.

First, since each LUT has just one output, a combinatorial circuit with n distinct outputs will require at least n LUTs. This implies for example, that an n input increment circuit requires at least n LUTs. It's easy to see how to implement an increment circuit with $2n$ LUTs (use a LUT for each gate), but we can do a little better than this by recognizing that each of the first three outputs is a function of just four inputs, so one gate suffices for each of these outputs. Also, the carry out of the third bit position is a function

of just four inputs. So, we can implement the first three bits of the incrementer using 4 LUTs and if we apply this approach repeatedly, we can implement a k bit increment circuit using $4k$ LUTs. So, in this case, we get within 33% of the lower bound on the number of LUTs implied by the number of outputs.

For circuits with more inputs than outputs, it's worth noting that a combinatorial circuit that uses k 4 input LUTs to produce n output signals can accommodate no more than $3k+n$ input signals, since each LUT whose output is not one of the circuit outputs must have its output connected to the input of another LUT. As a result, a circuit with m inputs and n outputs requires at least $\lceil (n-m)/3 \rceil$ LUTs. So for example, a mux with four data inputs (total of six inputs and one output) requires a minimum of 2 LUTs and a mux with eight data inputs (plus three control inputs) requires at least four LUTs. In practice, we need three LUTs to implement the 4-MUX and seven for the 8-MUX, so the lower bounds can only be viewed as rough estimates. Still, they provide a simple starting point for estimating the number of LUTs that a given circuit may require.

7. Designing Clocked Sequential Circuits

Clocked sequential circuits store values in flip flops, most often, edge-triggered D flip flops. VHDL provides a "synchronization condition" for use in if-statements that allows us to specify signals whose values are to be stored in flip flops or registers of flip flops. Here's an example.

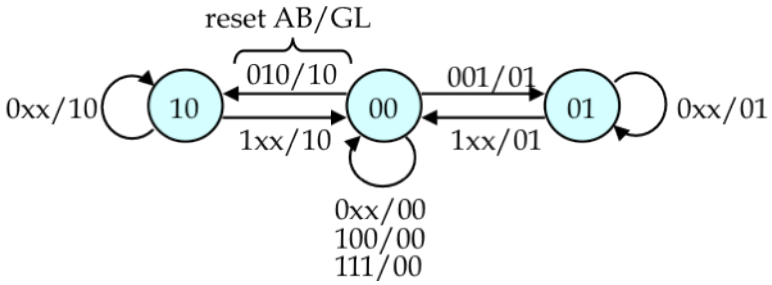
```
if rising_edge(clk) then x <= a xor b; end if;
```

The condition in the if-statement is the synchronization condition. The "scope" of the synchronization condition is the body of the if-statement. What this means is that assignments to signals within the body of the `if` statement are to occur only when the signal `clk` makes a transition from low to high. We obtain this behavior in a digital circuit by associating the signal `x` with a positive edge-triggered D flip flop, with `clk` connected to the clock input and the signal `a xor b` connected to the D input. Note that since `x` is associated with a D-flip flop controlled by the rising edge of `clk`, it does not make sense to have other assignments to `x` with incompatible synchronization conditions or no synchronization condition at all. It's usually most convenient to arrange one's VHDL specification so that all assignments to signals whose values are stored in flip flops lie within the scope of a single if-statement containing the synchronization condition. In fact, while the language doesn't require this, many circuit synthesizers cannot handle specifications with more than one synchronization condition in the same process. For this reason, we adopt

the common convention of using at most one synchronization condition in the processes used to specify sequential circuits.

Serial Comparator

VHDL makes it easy to write a specification for a sequential circuit directly from the state transition diagram for the circuit. The state diagram shown below is for a sequential comparator with two serial inputs, *A* and *B* and two outputs *G* and *L*. There is also a *reset* input that disables the circuit and causes it to go the 00 state when it is high. After *reset* drops, the *A* and *B* inputs are interpreted as numerical values, with successive bits presented on successive clock ticks, starting with the most significant bits. The *G* and *L* outputs are low initially, but as soon as a difference is detected between the two inputs, one or the other of *G* or *L* goes high. Specifically, *G* goes high if *A*>*B* and *L* goes high if *A*<*B*. Notice that *G* and *L* go high before the clock tick that causes the transition to the 10 and 01 states.



Here is a VHDL module that implements the comparator entity.

```
entity serialCompare is port(
    clk, reset: in std_logic;
    A, B : in std_logic;  -- inputs to be compared
    G, L: out std_logic); -- G=1 => A>B, L=1 => A<B
end serialCompare;

architecture scArch is
signal state: std_logic_vector(1 downto 0);
```

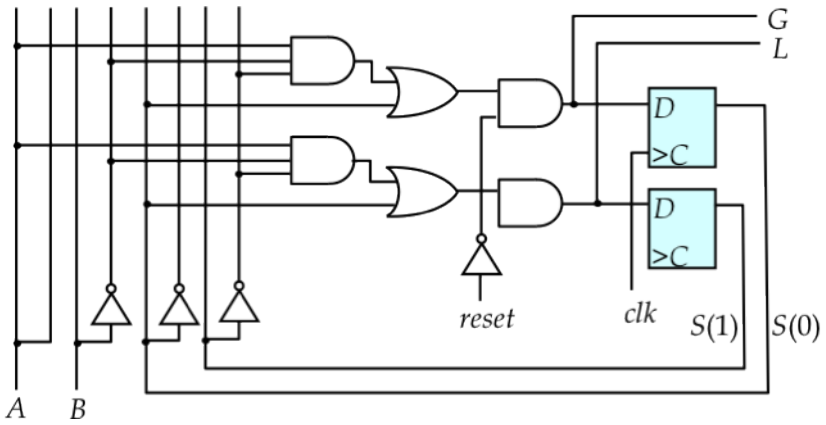
```
begin
    -- process that defines state transitions
    process (clk) begin
    if rising_edge(clk) then
        if reset = '1' then
            state <= "00";
        else
            if state = "00" then
                   if A > B then state <= "10";
               elsif A < B then state <= "01";
               end if;
            end if;
        end if;
    end process;

    -- process that defines the outputs
    process(A, B, state) begin
        G <= '0'; L <= '0';
        if (state="00" and A>B) or state = "10" then
            G <= '1';
        end if;
        if (state="00" and A<B) or state = "01" then
            L <= '1';
        end if;
    end process;
end scArch;
```

There are two processes in this specification. The first defines the state transitions and starts with an if-statement containing a synchronization condition. All assignments to the state signal occur within the scope of this if-statement causing them to be synchronized to the rising edge of the *clk* signal. We start the synchronized code segment by checking the status of *reset* and putting the circuit into state "00" if reset is high. The rest of the process controls the transition to the 10 or 01 states, depending on which

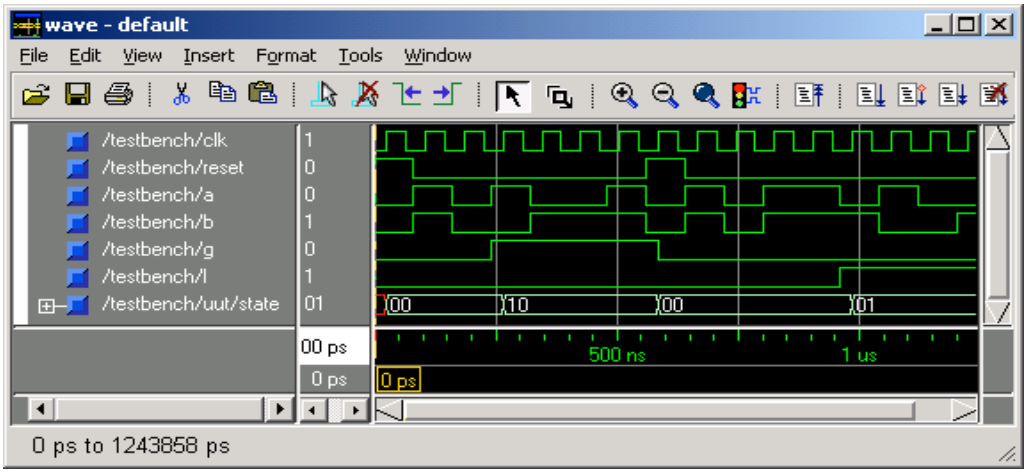
of the two inputs is larger. Notice that there is no code for the "self-loops" in the transition diagram, since these involve no change to the state signal. The synthesizer will generate the appropriate logic to handle the "no-change" conditions, but we need not write any explicit code for them. Also note that the sensitivity list in the first process contains only the clock signal. This is sufficient because signal changes only occur when *clk* changes. So unlike a process specifying a purely combinational circuit, there is no need to include the other signals that are used in the process.

The second process specifies the output signals *G* and *L*. Although it's not essential to define the outputs in a separate process, it's generally considered good practice to do so. Notice that this second process has no synchronization condition and specifies a purely combinational sub-circuit.

The VHDL synthesizer analyzes the two processes and determines that the state signal must be stored in a pair of flip flops. It also determines the logic equations needed to generate the next state and output values and uses these to create the required circuit. The diagram below shows a circuit that could be generated by the synthesizer from this specification.



The figure below shows the output of a simulation run for the serial comparator. Notice that the changes to the state variable are synchronized to the rising clock edges but the low-to-high transitions of *G* and *L* are not.



VHDL allows us to define signals with enumerated types so that we can associate meaningful names to values of signals. This is particularly useful for naming the states of state machines, as illustrated below.

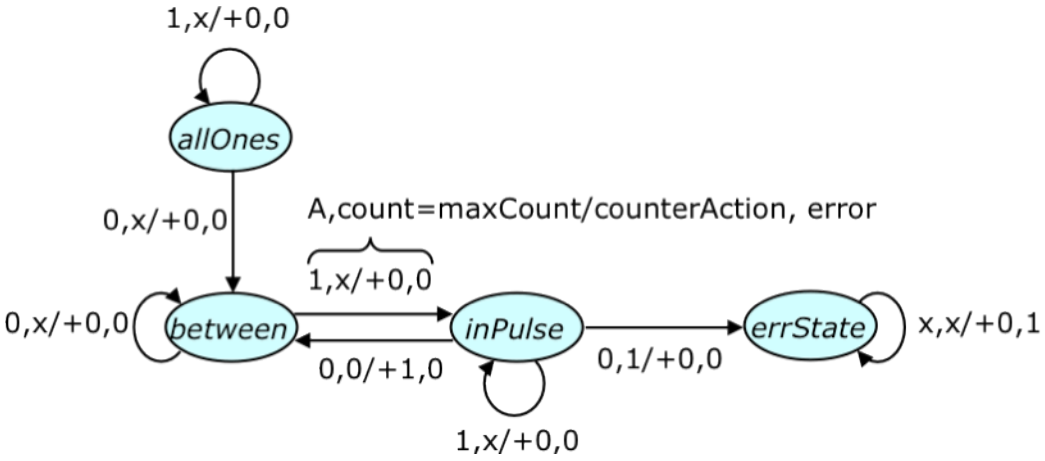
```
architecture scArch2 of serialCompare is
type stateType is (unknown, bigger, smaller);
signal state: stateType;
begin
    -- process that defines state transition
    process(clk) begin
        if rising_edge(clk) then
        if reset = '1' then
            state <= unknown;
        else
            if state = unknown then
                   if A > B then state <= bigger;
               elsif A < B then state <= smaller;
               end if;
            end if;
        end if;
    end process;
```

```
    -- code that defines the outputs
    G <= '1' when state = bigger or
                 (state = unknown and A > B)
         else '0';
    L <= '1' when state = smaller or
                 (state = unknown and A < B)
         else '0';
end scArch2;
```

In this version, we have also defined the outputs with two conditional signal assignments, instead of a process. In situations where the outputs are fairly simple, this coding style is preferable.

Counting Pulses

Next, we look at an example of a more complex sequential circuit that combines a small control state machine with a register to count the number of "pulses" observed in a serial input bit stream, where a pulse is defined as one or more clock ticks when the input is low, followed by one or more clock ticks when it is high, followed by one or more clock ticks when it is low. In addition to the data input *A*, the circuit has a *reset* input, which disables and re-initializes the circuit. The primary output of the circuit is the value of a counter. There is also an *error* output which is high if the input bit stream contains pulses than can be represented by the counter. If the number of pulses observed exceeds the counter's maximum value, the counter "sticks" at the maximum value. The simplified state transition diagram shown below does not explicitly include the reset logic, which clears the counter and puts the circuit in the *allOnes* state. Also, note that the counter value is not shown explicitly, since this would require that the diagram include separate '"between'" and "inPulse" states for each of the distinct counter values. Instead, we simply show whether the counter is incremented or not.



Here is a VHDL module that implements the pulse counter. In this example, we have introduced two constants, one for the word size and another for the maximum number of pulses that we can count.

```
--
-- Count the # of pulses in the input bit stream.
-- A pulse is a 01...10 pattern.
--
-- A is the input bit stream
-- count is the number of pulses detected
-- errFlag is high if the number of pulses exceeds
-- the number of pulses counted exceeds the capacity
-- of count
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use work.commonConstants.all;
```

```
entity countPulse is port (
   clk, reset: in std_logic;
   A: in std_logic;
   count: out std_logic_vector(wordSize-1 downto 0);
   errFlag: out std_logic);
end countPulse;
architecture a1 of countPulse is
type stateType is
    (allOnes, between, inPulse, errState);
signal state: stateType;
signal countReg:
        std_logic_vector(wordSize-1 downto 0);
begin
    process(clk) begin
        if rising_edge(clk) then
            if reset = '1' then
                countReg <= (others => '0');
                state <= allOnes;
            else
                case state is
                when allOnes =>
                    if A = '0' then
                        state <= between;
                    end if;
                when between =>
                    if A = '1' then
                        state <= inPulse;
                    end if;
                when inPulse =>
                    if A = '0' and
                       countReg /= maxPulse then
                        countReg <= countReg + "1";
                        state <= between;
                    elsif A = '0' and
                          countReg = maxPulse then
                        state <= errState;
                    end if;
                when others =>
```

```
                end case;
            end if;
        end if;
    end process;
    count <= countReg;
    errFlag <= '1' when state = errState else '0';
end a1;
```

Notice that we have defined a *countReg* signal separate from the *count* output signal because VHDL does not allow output signals to be used in expressions. The standard way to get around this is to have an internal signal that is used within the module and then assign the value of this internal signal to the output signal. The *countPulse* circuit illustrates a common characteristic of many sequential circuits. While strictly speaking, the state of the circuit consists of both the state signal and the value of *countReg*, the two serve somewhat different purposes. The state signal keeps tack of the "control state" of the circuit while the *countReg* variable holds the "data state". We can generally simplify the state transition diagram for a sequential circuit by representing only the control state explicitly, while indicating the modifications to the data state as though they were outputs to the circuit. This leads directly to a VHDL representation based on the transition diagram.

Priority Queue

We finish this section with a larger sequential circuit that implements a hardware priority queue. A priority queue maintains a set of (*key*,*value*) pairs. Its primary output is called *smallValue* and it is equal to the value of the pair that has the smallest key. So for example, if the priority queue contained the pairs (2,7), (1,5) and (4,2) then the *smallValue* output would be 5. There are two operations that can be performed on the priority queue. An *insert* operation adds a new (*key*,*value*) pair to the set of stored

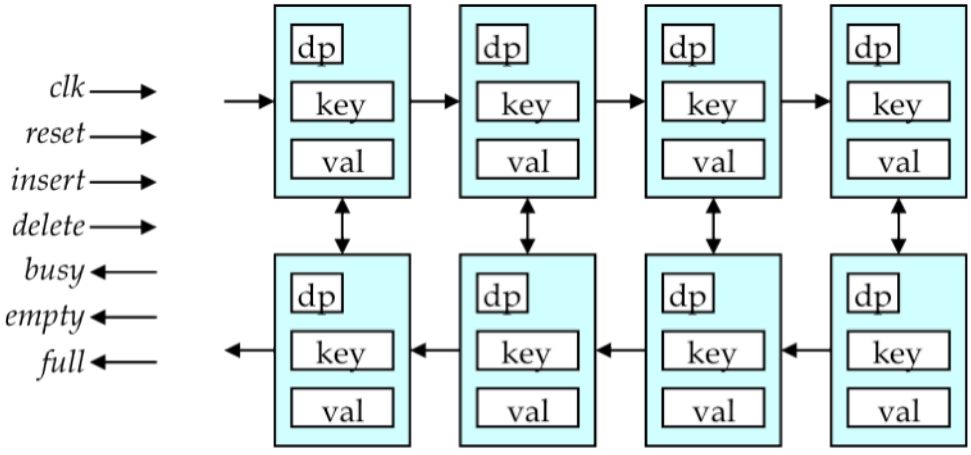
pairs. A *delete* operation removes the (*key*,*value*) pair with the smallest key from the set. The circuit has the following inputs.

clk	the clock signal
reset	initializes the circuit, discarding any stored values that may be present
insert	when high, it initiates an insert operation
delete	when high, it initiates a delete operation; however if insert and delete are high at the same time, the delete signal is ignored
key	is the key part of a new (*key*,*value*) pair
value	is the value part of a new (*key*,*value*) pair

The circuit has the following outputs, in addition to *smallValue*.

busy	is high when the circuit is in the middle of performing an operation; while busy is high, the insert and delete inputs are ignored; the outputs are not required to have the correct values when busy is high
empty	is high when there are no pairs stored in the priority queue; delete operations are ignored in this case
full	is high when there is no room for any additional pairs to be stored; insert operations are ignored in this case

The figure below shows a block diagram for one implementation of a priority queue.



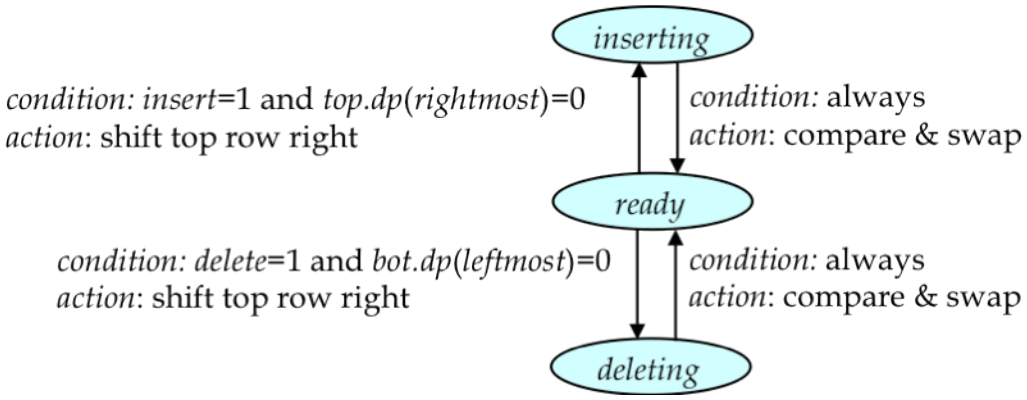
In this design, there is a set of *cells* arranged in two rows. Each cell contains two registers, one storing a *key*, the other storing a *value*. In addition, there is a flip flop called *dp*, which stands for data present. This bit is set for every block that contains a valid (*key*,*value*) pair. The circuit maintains the set of stored pairs so that three properties are maintained.

- For adjacent pairs in the bottom row, the pair to the left has a key that is less than or equal to that of the pair on the right.
- For pairs that are in the same column, the key of the pair in the bottom row is less than or equal to that of the pair in the top row.
- In both rows, the empty blocks (those with *dp*=0) are to the right and either both rows have the same number of empty blocks or the top row has one more than the bottom row.

When these properties hold, the pair with the smallest key is in the leftmost block of the bottom row. Using this organization, it is straightforward to implement the *insert* and *delete* operations. To do an *insert*, the (*key*,*value*) pairs in the top row are all shifted to the right one position, allowing the new pair to be inserted in the leftmost block of the top row. Then, within each column, the keys of the pairs in those columns are compared, and if necessary, the pairs are swapped to maintain properties 2 and 3. Note that the entire operation takes two steps. While it is in progress, the busy output is high.

The *delete* operation is similar. First, the pairs in the bottom row are all shifted to the left, effectively deleting the pair with the smallest key. Then, for each column, the *key* values are compared and if necessary, the pairs are swapped to maintain properties 2 and 3. Based on the third property, we can determine if the priority queue is full by checking the rightmost *dp* bit in the top row and we can determine if it is empty by checking the leftmost *dp* bit in the bottom row. The complete state of this circuit

includes all the values stored in all the registers, but we can express the control state much more simply, as shown in the transition diagram below.



This is a somewhat ideallized state transition diagram, but it captures the essential behavior we want. In particular, the labels on the arrows indicate the condition that causes the given transition to take place and any action that should be performed at the same time. The variable *top*(*rightmost*).*dp* refers to the rightmost data present flip flop in the top row and *bot*(*leftmost*).*dp* refers to the leftmost data present flip flop in the bottom row.

In the ready state, the circuit is between operations and waiting for the next operation. If it gets an insert request and it is not full, it goes to the *inserting* state and shifts the new (*key*,*value*) pair into the top row and shifts the whole row right. From there it makes a transition back to the ready state while doing a "compare & swap" between all vertical pairs.

If the circuit gets a *delete* request when it is in the ready state and is not empty, it goes to the *deleting* state and shifts the bottom row to the left. From there, it returns to the ready state, while performing a compare & swap. A VHDL module implementing this design is shown below.

```
-- Priority Queue module implements a priority queue
```

```
-- storing a set of (key,value) pairs.
--
-- When the priority queue is not empty, the output
-- smallValue is the value of a pair with the
-- smallest key. The empty and full outputs report
-- the status of the priority queue. The busy output
-- remains high while an insert or delete operation
-- is in progress. While it is high, new operation -
- requests are ignored
--
entity priQueue is port (
    clk, reset: in std_logic;
    insert, delete: in std_logic;
    key, value:
        in std_logic_vector(wordSize-1 downto 0);
    smallValue:
        out std_logic_vector(wordSize-1 downto 0);
    busy, empty, full : out std_logic);
end priQueue;

architecture pqArch of priQueue is
constant rowSize: integer := 4;
type pqElement is record
    dp: std_logic;
    key: std_logic_vector(wordSize-1 downto 0);
    value: std_logic_vector(wordSize-1 downto 0);
end record pqElement;
type rowTyp is array(0 to rowSize-1) of pqElement;
signal top, bot: rowTyp;
type state_type is (ready, inserting, deleting);
signal state: state_type;
begin
    process(clk) begin
        if rising_edge(clk) then
```

```
            if reset = '1' then
                for i in 0 to rowSize-1 loop
                    top(i).dp <= '0';
                    bot(i).dp <= '0';
                end loop;
                state <= ready;
            elsif state = ready and insert = '1' then
                if top(rowSize-1).dp /= '1' then
                    top(1 to rowSize-1) <=
                            top(0 to rowSize-2);
                    top(0) <= ('1',key,value);
                    state <= inserting;
                end if;
            elsif state = ready and delete = '1' then
                if bot(0).dp /= '0' then
                    bot(0 to rowSize-2) <=
                            bot(1 to rowSize-1);
                    bot(rowSize-1).dp <= '0';
                    state <= deleting;
                end if;
            elsif state = inserting or
                  state = deleting then
                for i in 0 to rowSize-1 loop
                    if top(i).dp = '1' and
                       (top(i).key < bot(i).key or
                                bot(i).dp = '0') then
                        bot(i) <= top(i);
                        top(i) <= bot(i);
                    end if;
                end loop;
                state <= ready;
            end if;
        end if;
    end process;
    smallValue <= bot(0).value when bot(0).dp = '1'
                else (others => '0');
    empty <= not bot(0).dp;
    full <= top(rowSize-1).dp;
```

```
    busy <= '1' when state /= ready else '0';
end pqArch;
```

Sequential circuits can be used to build systems of great sophistication and complexity. The challenge, to the designer is to manage that complexity so as not to be overwhelmed by it. VHDL is one important tool that can help in meeting the challenge, but to use it effectively, you need to learn the common patterns that experience has shown are most useful in expressing the functionality of digital systems. This section has introduced some of the more useful patterns. You should study the examples carefully to make sure you understand how they work and to develop a familiarity with the patterns they follow.

9. Still More Language Features

Variables

In addition to signals, VHDL supports a similar but different construct called a "variable", which has an associated variable assignment statement, which uses a distinct assignment operation symbol (:=). We have already seen a special case of variables, in the loop indexes used in for-loops. However, we can also declare variables and use them in ways that are similar to the way signals are used.

Unlike signals, variables do not correspond to wires or any other physical element of a synthesized circuit. The best way to think of a variable is as a short-hand notation representing the expression that was most recently assigned to the variable in the program text. So for example, the VHDL code fragment shown below with assignments to the variable *y*

```
a <= x"3a";
y := a + x"01";
b <= y;
y := y + x"10";
c <= y;
```

is exactly equivalent to the fragment

```
a <= x"3a";
b <= a + x"01";
c <= (a+x"01") + x"10";
```

Note that unlike with signal assignments, successive assignments to variables define different values; that is, following each variable assignment, the variable name refers to the most recent assigned value. Such uses of variables are not necessary, since we can always eliminate the variables by replacing each occurrence of a variable with the variable-free expression it represents. However, judicious use of variables can make code easier to both write and to understand. For example, the code fragment shown below implements an adder circuit, using a logic vector to represent the carries joining stages together.

```
architecture a1 of adder is
signal C: std_logic_vector(wordSize downto 0);
begin
    process (A,B,C,Ci) begin
        C(0) <= Ci;
        for i in 0 to wordSize-1 loop
            S(i) <= A(i) xor B(i) xor C(i);
            C(i+1) <= (A(i) and B(i)) or
                        ((A(i) xor B(i)) and C(i));
        end loop;
        Co <= C(wordSize);
    end process;
end a1;
```

When we introduced this example earlier, we pointed out that it was important to define C to be a logic vector, rather than a simple signal. However, we can re-write it using a variable for "C", making the code a little simpler.

```
architecture a1 of adder is
variable C: std_logic;
begin
    process (A,B,C,Ci) begin
        C <= Ci;
```

```
        for i in 0 to wordSize-1 loop
            S(i) <= A(i) xor B(i) xor C;
            C <= (A(i) and B(i)) or
                        ((A(i) xor B(i)) and C);
        end loop;
        Co <= C;
    end process;
end a1;
```

To understand why this works, remember that the meaning of the for-loop can be understood by unrolling the loop and substituting the appropriate values for the loop index *i*. Thus, each variable assignment to *C* will correspond to a different value of the loop index and when the signal assignment for each value of *S*(*i*) is processed, it will substitute the appropriate expression in place of the variable name *C*.

The behavior of variable assignments in VHDL is similar to the behavior of variable assignments in conventional sequential programming languages and distinctly different from the behavior of signal assignments. However, the underlying mechanisms that lead to this behavior are distinctly different, and this can lead to subtle differences. While variable assignment in a conventional language corresponds to updating a memory location in a computer's memory, variable assignment in VHDL corresponds to defining an abbreviation for whatever expression appears on the right.

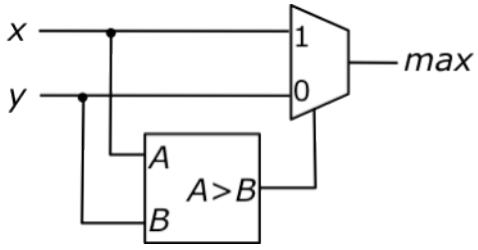
The use of both signals and variables in VHDL can be confusing, especially at first. Indeed, while variables can be helpful, the fact that they behave differently from signals can make code harder to understand, rather than easier. You may prefer to limit your use of variables to contexts where they can't be avoided (like loop indexes), and this is certainly a reasonable choice to make, since most uses of variables are not really necessary. To reiterate, it's generally best to think of a variable as just a short-hand notation for the expression most recently assigned to the variable name. Variables correspond to nothing physical in the synthesized circuit, but they can simplify the specification of a circuit.

Functions and Procedures

Like conventional programming languages, VHDL provides a subroutine mechanism to allow you to encapsulate circuit components that are used repeatedly in different contexts. The example below shows how a function can be used to specify a circuit that select the larger of two input values.

```
function max(x,y: word) return word is
begin
    if x > y then return x; else return y; end if;
end function max;
```

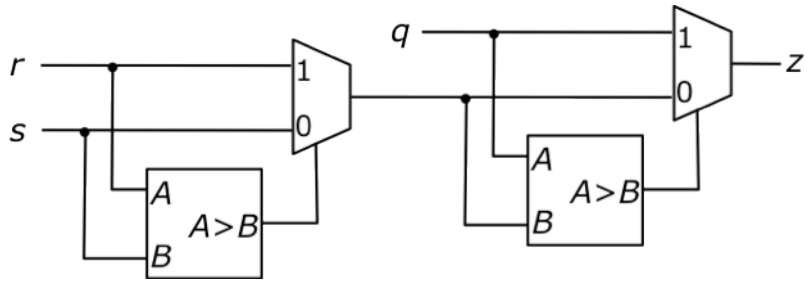
This function can be implemented by the circuit shown below.



Given this declaration, we can write instantiate this circuit multiple times, simply by referencing the function in an expression.

```
z <= max(q,max(r,s);
```

defines the circuit



Note that a function can only specify a single output of the sub-circuit it specifies (the return value). VHDL also defines *procedures* that allow one to define sub-circuits with multiple outputs. Here is an example of a procedure that compares an input value to a lower and upper bound and outputs three signals, indicating whether the input value is within the range defined by the two bounds, is above the range or below it.

```
procedure inRange( x: in word;
                   inRange: out std_logic;
                   tooHigh: out std_logic;
                   tooLow: out std_logic) is
constant loBound: word := x"0005";
constant hiBound: word := x"0060";
begin
    tooLow := '0'; inRange := '0'; tooHigh := '0';
       if x <  loBound then tooLow := '1';
    elsif x <= hiBound then inRange := '1';
    else                    tooHigh := '1';
    end if;
end procedure;
```

Note that the formal parameters to a function or procedure are variables, not signals. This means that within a procedure, we must use the variable assignment operator to assign values to its output parameters. When a function or procedure is used, the input arguments may be signals, but the output arguments must be declared as variables in the context where the function or procedure is used. These variables may then be assigned to signals.

Functions and procedures can be important components of a larger VHDL circuit design. They can eliminate much of the repetition that can occur in larger designs, facilitate re-use of design elements developed by others and can make large designs easier to understand and manage.