Chapter 3
Hardware Design Using Verilog

If I designed a computer with 200 *chips, I tried to design it with* 150*. And then I would try to design it with* 100*. I just tried to find every trick I could in life to design things real tiny.*

– S. Wozniak

Abstract The goal of this chapter is not to give a comprehensive guide to Verilog development but to provide enough of an introduction that the reader can understand and implement concepts used as examples in subsequent sections. Although these concepts become increasingly complex they build on the basic principles outlined in this chapter and, as such, it should provide enough of a reference that no other text is necessary. We omit the study of low-level concepts like transistor-level design, subjects related to Verilog such as formal verification, and advanced topics such as the Programming Language Interface (PLI).

3.1 Introduction

3.1.1 The Problem of Design Complexity

Looking back as recently as fifty years ago, the number of components used to construct a given circuit was in the range of 10 to 100, all selected from a small set of choices. Operating at this scale, it was not unrealistic for a single engineer to design an entire circuit on paper and then sit with a soldering iron to construct it by hand. Malone describes one extreme example of what could be achieved [38, Pages 148–152]. Development of a floppy disk controller for the Apple II was, in early 1977, viewed as a vital task so that the computer could remain competitive in a blossoming market. Steve Wozniak, a brilliant technical mind and co-founder of Apple, developed and improved on existing designs and within two weeks had a working circuit. Not content with eclipsing the man-hour and financial effort required by other companies to reach the same point, Wozniak redesigned and optimised the entire circuit

over a twelve-hour period after spotting a potential bug before it was sent into production !

As circuit complexity has increased over the years, and lacking a Wozniak to deploy in every design company, this model fails to scale. For example, a modern computer processor will typically consist of many millions of transistors and related components. Just as designing robust, large-scale software is a challenge, designing a million-component circuit is exceptionally difficult. The sheer number of states the circuit can be in requires splitting it into modular sub-systems, the interface between which must be carefully managed. Without doing this, verifying the functional correctness of the circuit is next to impossible; one must instead verify and test each module in isolation before integrating it with the rest of the design. Physically building the circuit by hand is equally impossible because of the size of the features: one would need an exceptionally tiny soldering iron to wire connections together ! Furthermore, one would need almost infinite patience because of the number of connections. Since humans are bad at repetitive tasks like this, errors will inevitably be introduced. Coping with design constraints such as propagation delay becomes massively time consuming since the act of component layout and connection with so many items is a mind-bogglingly complex optimisation problem. The increased number of components introduces further problems; issues of power consumption and heat dissipation start to become real problems.

Several key advances have tamed the problems in this gloomy picture. Certainly improvements in implementation and fabrication technologies that allow miniaturisation of components have been an enabling factor, but the advent of **design automation** (i.e., software tools to help manage the complexity of hardware design) has arguably been just as important.

3.1.2 Design Automation as a Solution

Verilog is a **Hardware Description Language (HDL)**. A language like C is used to describe software programs that execute on some hardware device; a language like Verilog can describe the hardware device itself. Using a HDL, usually within some form of **Electronic Design Automation (EDA)** software suite, offers the engineer similar benefits to a programmer using a high-level language like C over a low-level machine language.

The Verilog language was first developed in 1984 at a company called Gateway Design Automation as a hardware modelling tool. Some of the language design goals were simplicity and familiarity; as a result Verilog resembles both C and Pascal with many similar constructs. Coupled with the provision of a familiar and flexible development environment, Verilog promotes many of the same advantages that software developers are used to. That is, the language makes it easy to abstract away from the implementation and concentrate on design; it makes design modularisation and reuse easier; associated tools such as simulators make test and verification easier and more efficient. There are two standards for the language: an initial version

termed Verilog-95 was improved and resubmitted to the IEEE for standardisation as Verilog-2001 or IEEE 1364-2001. Much of this improvement was driven by input from users in companies such as Motorola and National Semiconductor. This standardisation, along with the production of software to synthesise real hardware from Verilog descriptions at a variety of abstraction levels, has made the language a popular choice for engineers.

Verilog **models**, so-called because they are somewhat abstract models of physical circuits, can be described at several different levels. Each level is more abstract than the last, leaving less work for the engineer and allowing them to be more expressive:

Switch Level At the lowest level, Verilog allows one to describe a circuit in terms of transistors. Although this gives very fine-grained control over the circuit composition, developing large designs is still a significant challenge since the building blocks are so small.

Gate Level Above the transistor level, and clearly more attractive for actually getting anything useful done, is the concept of gate-level description. At this level, the designer describes the functionality of the circuit in terms of basic logic gates and the connections between them; this is somewhat equivalent to the level of detail in the previous chapter.

Register Transfer Level (RTL) RTL allows the designer to largely abstract away the concept of logic gates as an implementation technology and simply describe the flow of data around the circuit and the operations performed on it.

Behavioural Level The most abstract and hence most expressive form of Verilog is so-called behavioural-level design. At this level, the designer uses high-level constructs, similar to those in programming languages like C, to simply specify the required behaviour rather than how that behaviour is implemented.

Roughly speaking, development of real hardware from a Verilog model is a three-phase process which we describe as being managed by a suite of software tools called the Verilog **tool-chain**:

Simulate During the first phase, the Verilog model is fed into a software tool which simulates the behaviour of the circuit. This simulation is highly accurate, modelling how gates and signals change as time progresses, but is abstract in the sense that many constraints of real hardware are not considered. Since by using software simulation it is easy to alter the model and debug the result, a development cycle is relatively quick. As a result, the simulation phase is typically used to ensure a level of functional correctness before the model is processed into real hardware. That is, if the model simulates correctly, then one can progress to implementing it in real hardware with a greater confidence of success.

Synthesise The second phase of development is similar to the compilation step whereby a C program is translated into low-level machine language. The Verilog model is fed into synthesis software which converts the model into a list of real hardware components which implement the required behaviour. This is often called a **netlist**. Optimisation steps may be applied by the synthesiser to improve the time or space characteristics of the design. Such optimisations are usually

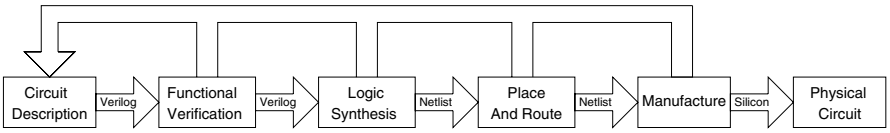

Figure 3.1 A waterfall style hardware development cycle using Verilog.

conceptually simple but repetitive: ideal fodder for a program but not so for the engineer.

Implement The synthesis phase produces a list of components and a description of how they are connected together to form the implementation. However, a further step is required to translate this description into real hardware. Firstly, we need to select components from the implementation technology with which to implement the components in the netlist. For example, on an FPGA it might be advantageous to implement an AND gate one way whereas on an ASIC a different method might be appropriate. Secondly, we need to work out where the components should be placed and how the wiring between them should be organised or routed. Long wires are undesirable since they increase propagation delay; the place and route process attempts to minimise this through intelligent organisation of the components. Again, this is a problem which is well suited to solution by a computer but not by the engineer.

In reality, these steps form part of a waterfall style development cycle (or similar) which iterates each stage to remove errors after some form of verification.

3.2 Structural Design

3.2.1 Modules

Programs written in C can be thought of as collections of functions that use each other to perform some computation. The computation is determined by how the functions call each other and by what goes on inside each function. The analogous construct in Verilog is a **module**. Basically speaking, a module defines the **interface** of a hardware component, i.e., the inputs and outputs, and how the component does computation to produce the outputs from the inputs and any stored state.

Like functions, modules provide us with a first layer of abstraction: we can view them as a black-box, with inputs and outputs, without worrying how the internals work. For example, the multiplexer device introduced previously selects between several values to produce a result; we might draw such a device as previously shown in Figure 2.16a. From a functional point of view, we do not really care what is inside

```
module mux2_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire s0 );

  ...

endmodule
```

Listing 3.1 The interface for a 1-bit, 2-way multiplexer.

```
module mux2_1bit( r, i0, i1, s0 );

  output wire  r;
  input  wire i0;
  input  wire i1;
  input  wire s0;

  ...

endmodule
```

Listing 3.2 The interface for a 1-bit, 2-way multiplexer.

the component. That is, in order to use a multiplexer we do not care how it works or what gates are used within it, just what it does.

We can describe the interface of Figure 2.16a, i.e., the inputs and outputs rather than the internals, using the Verilog module definition shown in Listing 3.1. A module definition like this specifies a name or **identifier**, this one is called `mux2_1bit`, and a list of inputs and outputs which it uses for communication with other modules. We term the inputs and outputs defined by a module as the **port list**; each input or output is a **port** over which it communicates, the direction of communication is determined by the `input` and `output` keywords. In this case, we have replaced the **body** of the module, which determines how it works, with some continuation dots that we will fill in later.

Listing 3.2 offers a different way of defining the same interface. This time, instead of describing the input and output characteristics of each port inline with the definition, we simply name the ports and give them a type inside the module. Although the two methods are largely equivalent, there are some advantages to the second method which we will see later. Typically, for simple modules the first method is more compact and so is preferred unless there is a good reason not to use it.

3.2.2 Wires

In Verilog, a **net** is the term used to describe the connection between two modules. By net we essentially we mean a **wire** which is the exact analogy of the same thing in real life: a conducting medium that allows a signal to be sent between the two

```
1 wire              w0;      // 1-bit unsigned wire
2 wire        [3:0] w1;      // 4-bit unsigned wire vector
3 wire        [0:3] w2;      // 4-bit unsigned wire vector
4 wire        [4:1] w3;      // 4-bit unsigned wire vector
5 wire signed [3:0] w4;      // 4-bit signed wire vector
```

Listing 3.3 Several different examples of wire definition.

ends. We say that we **drive** a value onto one end before it propagates along the wire and can be read at the other end. We define Verilog wires using a similar syntax to variable declaration in C, as shown in Listing 3.3.

Line 1 defines a wire called `w0` capable of carrying 1-bit values. Although 1-bit values are fine for some situations, often we will want to communicate larger values. So-called **wire vectors** allow us to do exactly this by bundling 1-bit wires into larger groups. Since we can select the individual wires from a given wire vector, we can think of the vector as either as an array of four 1-bit wires, or a single compound wire that can carry 4-bit values. Line 2 demonstrates this by defining a wire vector called `w1` which can carry 4-bit values. One can read the range notation `[3:0]` as 3 down to 0 or 3,2,1,0. This is a means of specifying the size of the wire vector, as well as the order and index of wires within it. For example, Line 3 defines another 4-bit wire vector but the order of the range is reversed, the left-most bit of `w2` is bit 0 while the right-most is bit 3. In the previous example, the left-most bit was bit 3 while the right-most was bit 0. Line 4 uses a third variation where the order of wires within the wire vector is the same as `w1` but their index is changed. This time the left-most bit of `w3` is numbered 4, rather than 3, while the right-most is numbered 1, rather than 0. These variations might seem confusing but they are useful in practise. For example, reversing the order of the bits in a wire allows us to easily change the endianness and hence easily comply with demands set by the system we want to connect our design to.

It is worth noting that more recent Verilog language specifications include the facility to declare signed wire vectors, i.e., wire vectors that carry signed, twos-complement values rather than unsigned values. For example, Line 5 in Listing 3.3 declares a 4-bit signed wire vector called `w4`; it can take the values between -8 and 7 inclusive. This differs from Line 2 where `w1` is declared as a 4-bit unsigned wire vector capable of taking the values between 0 and 15 inclusive. Clearly this is a matter of interpretation to some extent since one can view a given bit pattern as representing a signed or unsigned value regardless of the wire vector type. However, the addition of the signed keyword "hints" to the Verilog tool-chain that, for example, if the value on signed wire `w4` is inspected or displayed somehow it should be viewed as signed.

Value	Meaning
`1'b0`	Logical low or false
`1'b1`	Logical high or true
`1'bX`	Undefined or unknown value
`1'bZ`	High impedance value

Table 3.1 A list of the four different value types.

Strength	Meaning
`supply`	Strongest
`strong`	
`pull`	
`large`	
`weak`	↓
`medium`	
`small`	
`highz`	Weakest

Table 3.2 A list of the eight different strength types.

3.2.3 Values and Constants

In a perfect hardware system, a wire should only ever take the values 0 or 1 (i.e., true and false) written in Verilog as `1'b0` or `1'b1`. However, physical properties of such systems dictate they are never perfect. As a result, Verilog supports four different values, shown in Table 3.1, to model the reality of imperfect hardware. In a sense, there is a similar motivation here as in the discussion of 3-state logic in Chapter 2.

The **undefined** or **unknown** value `1'bX` is required to model what happens when some form of conflict occurs. It is not that we do not care what the value is, we genuinely do not know what the value is. As a simple example, consider driving the value `1'b1` onto one end of a wire and the value `1'b0` onto the other end, what value does the wire take ? The answer is generally undefined, but roughly depends on how strong the two signals are. Verilog offers a way to model how strong a value of `1'b0` or `1'b1` is; one simply prefixes the value with one of the strength types shown in Table 3.2. However, this sort of assumption should be used with care: generally speaking, if you rely on an unknown value being resolved to something known, there is probably a better way to describe the design and avoid the problem. The **high impedance** value `1'bZ` is a kind of pass-through described previously in the context of 3-state logic. If high impedance collides with any value, it instantly resolves to that value. This is useful for modelling components which must produce

some continuous output but might not want to interfere with another component using the same wire.

Constant values, which one can use to drive wires or to compare against values read from wires for example, can be written in a variety of ways using the general format

```
[<sign>] [<size>'<base>] <value>
```

such that the fields in square brackets are optional. The `<sign>` field is an optional minus symbol to denote negative constants; `<size>` denotes the number of bits in the constant; `<base>` is the number base the constant is expressed in, for example `H` or `h` for hexadecimal, `D` or `d` for decimal and `B` or `b` for binary; and value is the actual constant. So, for example we have that:

- `2'b10` is a 2-bit binary value $10_{(2)}$ or $2_{(10)}$ or $2_{(16)}$.
- `4'hF` is a 4-bit hexadecimal value $1111_{(2)}$, $15_{(10)}$ or $F_{(16)}$.
- `8'd17` is an 8-bit decimal value $17_{(10)}$ or $00010001_{(2)}$ or $11_{(16)}$.
- `3'bXXX` is a 3-bit unknown binary value.
- `8'hzz` is an 8-bit high impedance value.
- `4'b10XZ` is a 4-bit binary value where bit zero is the high impedance value, bit one is the unknown value, bit two is logical low and bit one is logical high. This value does not really translate into another base; although some of the bits are defined, because some are not the whole value is undefined.

Where size is not important, one can omit the `<size>` field and accept a default. Likewise, one can omit the `<base>` field to accept the default of decimal numbers so that the constant `10` is the same as `4'd10`. Clearly we can also specify signed constants by prefixing them with a negation symbol such as `-10` or `-4'd10`.

3.2.4 Comments

Like most other programming languages, Verilog allows one to **comment** source code. Comments are used with the same syntax as C and Java so that single-line comments

```
// comment
```

and multi-line comments

```
/*
comment
comment
...
*/
```

are both possible. It goes without saying that since Verilog models can contain just as many subtle implementation issues as a C program, comments are a vital way to convey meaning which might not be present in the source code.

```
1 module or_gate( output wire r,
2                 input  wire x,
3                 input  wire y );
4
5   ...
6
7 endmodule
```

Listing 3.4 The interface for a 2-input OR gate.

```
1 or t0( r, x, y       ); // 2-input or gate instance
2 or t1( r, x, y, z    ); // 3-input or gate instance
3 or t2( r, w, x, y, z ); // 4-input or gate instance
```

Listing 3.5 Instantiating primitive gates with multiple inputs.

3.2.5 Basic Instantiation

Continuing the analogy between C functions and Verilog modules, a function clearly does not do anything until it is called: the function is just a description. Equally, a module does not do anything until it is **instantiated**. Instantiating a module is the equivalent of actually placing a physical copy of it down on our design. If we instantiate a multiplexer module, this is the analogy of taking a multiplexer component from a box and placing it on a circuit board.

Verilog offers a number of **primitive modules** that represent logic gates. In a sense you can think of primitive modules as similar to the C standard library which makes basic operations available to *all* C programs. We can connect these together to describe more complex circuits and as such they offer a convenient way to introduce the instantiation syntax. Consider a primitive two input OR gate. Although it is provided for us by Verilog, we can think of it having an interface similar to that shown in Listing 3.4 (we deliberately name it `or_gate` and not `or` to avoid conflict with the built-in name). In fact, Verilog is nicer to us than simply providing a 2-way version, it provides multi-input versions of these modules; one simply adds more inputs to the instantiation as shown in Listing 3.5. For example, the instantiation of `t1` in Line 2 names three inputs `x`, `y` and `z`.

Using this primitive OR gate module, and similar ones for AND, NOT and XOR, we can start filling in the module body of Listing 3.1 so it matches the multiplexer design from Figure 2.16a whose truth table is shown in Table 2.10a. Listing 3.6 describes an implementation by first defining some internal wires to connect the components and then instantiating primitive modules to represent those components. Each instantiation is built from a few syntactic parts: the type of the module, an identifier for the instance, and a port list. In this case, the types are `not`, `and` and `or`; the identifiers are `t0`, `t1`, `t2` and `t3`. The port list describes how wires connect to the communication ports of the module. For example, in the case of the `or` gate we have connected wires `w1` and `w2` to the `or` gate input ports, and output of the multiplexer `r` to the `or` gate output port. As another example, consider the imple-

```
module mux2_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire s0 );

  wire w0, w1, w2;

  not t0( w0, s0     );

  and t1( w1, i0, s0 );
  and t2( w2, i1, w0 );

   or t3( r,  w1, w2 );

endmodule
```

Listing 3.6 An implementation of a 1-bit, 2-way multiplexer using primitive modules.

```
module add_1bit( output wire co,
                 output wire  s,
                 input  wire ci,
                 input  wire  x,
                 input  wire  y );

  wire w0, w1, w2;

  xor t0( w0, x,  y  );
  and t1( w1, x,  y  );

  xor t2(  s, w0, ci );
  and t3( w2, w0, ci );

   or t4( co, w1, w2 );

endmodule
```

Listing 3.7 An implementation of a full-adder using primitive modules.

mentation of a full-adder in Listing 3.7. Exactly the same principles apply as in the multiplexer implementation: we again wire together a number of primitive gate instances to replicate the functionality required by the truth table shown in Table 2.8b.

Stated in the above form, module instantiation appears very similar to a C function call which will also typically have an identifier and arguments. But there are some crucial differences between the two:

- Typically, a compiler translates each function declared in a C program into *one* implementation in the executable. That is, two separate calls to the same function actually use the same code: they call the same function implementation. With module instantiation, rather than being shared, *each* instance is a *separate* item of hardware.
- Within a typical C program, each function call in the program is executed *sequentially* so that at any given time we know where in the program execution is: it cannot be in two functions at the same time. With a Verilog design, every mod-

```
module mux4_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire i2,
                  input  wire i3,
                  input  wire s0,
                  input  wire s1 );

  wire w0, w1;

  mux2_1bit t0( w0, i0, i1, s0 );
  mux2_1bit t1( w1, i2, i3, s0 );
  mux2_1bit t2( r,  w0, w1, s1 );

endmodule
```

Listing 3.8 An implementation of a 1-bit, 4-way multiplexer using user-defined modules.

ule instance executes in *parallel* with every other instance so that one module can be doing things at the exact same time as another.

Since each instance executes in parallel with every other, the `or` gate used above can be described as **continuous** in the sense that it does not wait for the inputs to be ready before computing the output: it is always computing an output. We could just as well write Listing 3.6 with the `or` gate instantiated before both the `and` gates. In C, this would make no sense because now the results `w1` and `w2` are not produced before they are used. In Verilog however, all three modules are continuously working in parallel so it does not matter what order we instantiate them in as long as they are wired together correctly.

Clearly this continuous behaviour can cause problems if you are used to the sequential nature of C. For example, what happens if some of the inputs have values driven onto them while others do not ? Since the module is continually generating an output, the output must be something, but it will typically be resolved as the undefined value. When executing a C program, the processor deals with this issue for us by forcing each instruction to execute in order; from a functional perspective a given instruction never starts until previous instructions are completed. In Verilog however, we are left on our own to enforce such rules.

This presents us with the first challenge of timing within Verilog models: we need to be sure that when we use the output of a module, enough time has elapsed to generate it. We have already seen that propagation delay in gates takes some time which could impact on when the output becomes valid. As the model size increases these small delays can accumulate and present a significant delay which needs to be accommodated. In the simple combinatorial circuits we have looked at thus far, timing is not a major problem since we typically just drive the input values and wait until they generate the output. With models that store state, which we encounter in Section 3.4, timing becomes much more important. For example, we need to ensure that we only set the value of our state with values that are valid, i.e., are not undefined.

3.2.6 Nested Instantiation

Primitive modules are just like modules you describe yourself except that they are provided by Verilog. Therefore we can consider a design composed of several of our own modules in the same way as we did above with the 2-way multiplexer. Consider, for example, extending the 2-way multiplexer so it has four inputs selected between using two control signals. One can think of at least two methods of implementing this design which were described previously: either we use the same primitive gates as in the 2-way case or we build it using our previously defined module. Ignoring which is the most efficient in terms of the number of gates, the latter method clearly leans on one of the advantages of Verilog in that reuse of modules is made very easy. Listing 3.8 takes advantage of this by implementing a 4-way multiplexer called `mux4_1bit` using three instances of the 2-way multiplexer `mux2_1bit`. Exactly the same rules apply when instantiating `mux2_1bit` as when instantiating a primitive `or` gate: we have a type, an identifier and a port list for each instance and each instance of our 2-way multiplexer operates continuously and in parallel with every other.

Verilog instantiation creates a **hierarchy** of instances. Each instance is said to contain those modules it in turn instantiates. To keep track of this hierarchy, Verilog outlines a simple naming scheme. For example, the 4-way multiplexer in Listing 3.8 has three instances of a 2-way multiplexer named `t0`, `t1` and `t2`. Within each of these 2-way multiplexers there is a `not` gate named `t0`, two `and` gates named `t1` and `t2` and an `or` gate named `t3`. Hierarchical naming means each primitive gate has a unique name: the `not` gate within 2-way multiplexer `t1` is called `t1.t0` while the one in multiplexer `t2` is called `t2.t0`. This is a useful property since we can embed debugging mechanisms within a module and determine exactly where the messages are coming from; without the naming scheme, all debugging from the module `mux2_1bit` would be mixed together and become useless with respect to locating errors.

3.2.7 User-Defined Primitives

For some simple functions, building a module from gates is overly verbose. Additionally, it defeats one of our reasons to use Verilog which was to hand off as much work to the Verilog tool-chain as we can. The **User-Defined Primitive (UDP)** offers a more convenient way to build simple modules by essentially allowing the designer to specify the truth table and leave the implementation to the tool-chain. Given such a truth table, the tool-chain tries to find the most optimal way to realise it, using for example the Quine-McCluskey method.

UDPs have a similar interface to modules. For example, we might define our 2-way multiplexer as shown in Listing 3.9. The body of the UDP is the truth table which states, given some combination of input, what output should be produced. Each line in the table lists the values of the input ports followed by a colon and

```
primitive mux2_1bit( output  r,
                     input  i0,
                     input  i1,
                     input  s0 );
  table
    0 ? 0 : 0;
    1 ? 0 : 1;
    ? 0 1 : 0;
    ? 1 1 : 1;

    ? ? X : X;
  endtable

endprimitive
```

Listing 3.9 An implementation of a 1-bit, 2-way multiplexer using a UDP.

then the value required on the output port. We are free to include the unknown value `1'bX` but the high impedance value `1'bZ` is treated as `1'bX`. We are also free to include don't care states in the table as we might with a Karnaugh map; such states are written using a question mark. Thus, the first line above can be read as "when both `i0` and `s0` are 0 and `i1` is any value, set the result `r` to 0". The last line can be read as "when `s0` is unknown and both `i0` and `i1` are any value, set the result `r` to unknown".

Compared to the description of the same module in terms of gates, this seems a much nicer way to do things. However, there are some caveats to consider when defining a UDP:

- There can only be one output port although there can be as many inputs as required; the output port must be specified first in the port list.
- Both input and output ports can only be single wires; wire vectors are not allowed.

3.3 Higher-level Constructs

Expressing a design purely in terms of basic logic gates is analogous to writing software directly in a low-level machine language: it offers great control over the result but progress requires significant effort on the part of the programmer. To combat this, we can use RTL style design to construct Verilog models at a higher-level of abstraction. Rather than worrying exactly how the functionality is realised, RTL allows us to concentrate on how data flows around the design and the operations performed on it. Clearly there is some overlap between RTL and gate-level design; essentially RTL offers a short hand way to describe combinatorial logic.

```
module nand_gate( output r,
                  input  w,
                  input  x,
                  input  y,
                  input  z );

  assign r = ~( w & x & y & z );

endmodule
```

Listing 3.10 Using a continuous assignment to replicate the behaviour of a primitive module.

```
module mux2_1bit( output wire  r,
                  input  wire i0,
                  input  wire i1,
                  input  wire s0 );

  assign r = ( s0 ? i1 : i0 );

endmodule
```

Listing 3.11 An implementation of a 1-bit, 2-way multiplexer using a continuous assignment.

3.3.1 Continuous Assignments

In Section 3.2 we described the operation of primitive gates as continuous in the sense that they were continuously calculating outputs from their inputs rather than waiting for the inputs to be available. We can use a higher-level, **continuous assignment** RTL statement to do the same thing. Such a statement looks similar to an assignment in C in the sense that it has left-hand side (LHS) and right-hand side (RHS) expressions. The LHS is said to be assigned a value calculated by evaluating the RHS. In a C program we typically expect the LHS to be a variable; in a Verilog continuous assignment the LHS is a wire or wire vector. Furthermore, and unlike in a C program, a Verilog continuous assignment is continually being executed rather than only being executed when control-flow reaches the assignment. This is a fundamental difference and relates to the fact that instantiations of Verilog modules, which a continuous assignment will be translated into by the tool-chain, are continuously operating in parallel with each other.

The beauty of the construct is that one can use a huge range of C style logical and arithmetic operators as part of the RHS expression; Table 3.3 contains a list of them, their types and number of operands. Consider the construction of a 4-input NAND gate shown in Listing 3.10 (again we name the module `nand_gate` and not `nand` to avoid conflict with the built-in name). Of course, one could use a built-in primitive module to achieve the same thing, but this simple assignment neatly demonstrates the expressive power of a continuous assignment. Instead of several module instantiations and wire connections, we simply write the required RHS expression and let the Verilog tool-chain convert this into an efficient low-level implementation. Virtually all C operators are permitted so, for example, we can re-

Type	Symbol	Meaning	Operands
Arithmetic	*	multiply	2
	/	divide	2
	+	add	2
	-	subtract	2
	%	modular reduction	2
	**	exponentiation	2
Logical	!	logical NOT	1
	&&	logical AND	2
	||	logical OR	2
Relational	>	greater than	2
	<	less than	2
	>=	greater than or equal to	2
	<=	less than or equal to	2
	==	equal to	2
	!=	not equal to	2
	===	case equal to	2
	!==	case not equal to	2
Bitwise	~	bitwise NOT	1
	&	bitwise AND	2
	|	bitwise OR	2
	^	bitwise XOR	2
Reduction	&	reduce AND	1
	~&	reduce NAND	1
	|	reduce OR	1
	~|	reduce NOR	1
	^	reduce XOR	1
	~^	reduce XNOR	1
Shift	<<	logical left-shift	2
	<<<	arithmetic left-shift	2
	>>	logical right-shift	2
	>>>	arithmetic right-shift	2
Misc	[...]	selection	*n*
	{ ... }	concatenation	*n*
	{ { ... } ... }	replication	*n*
	... ? ... : ...	conditional	3

Table 3.3 A list of Verilog operator types.

x	$x \wedge y$
0	1
1	0
X	X

(a) NOT.

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1
0	X	0
1	X	X
X	0	0
X	1	X
X	X	X

(b) AND.

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1
0	X	X
1	X	1
X	0	X
X	1	1
X	X	X

(c) OR.

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0
0	X	X
1	X	X
X	0	X
X	1	X
X	X	X

(d) XOR.

Table 3.4 Logical operators in the presence of unknown values.

implement our 2-way multiplexer using an inline-conditional expression as shown in Listing 3.11. In this case, the LHS is the output wire `r` whose value is calculated by evaluating the RHS expression. If wire `s0` is 1, the expression evaluates the value of `i1`, otherwise it evaluates to the value of `i0`.

One subtle caveat exists with the analogy between gates and logical operators: the latter deals explicitly with cases where one or more of the inputs are the unknown or high impedance values `1'bX` or `1'bZ`. Firstly, in this context the value `1'bZ` is treated as unknown or `1'bX`. Then, the rules in Table 3.4 are applied to describe the output based on the inputs. Clearly there are additional entries versus traditional truth tables for NOT, AND, OR and XOR. These additional entries show that, for example, if one were to AND together two values using `1'b0 & 1'bX`, then the result would be `1'b0`. Intuitively this might seem odd: how can one decide on a result if one of the inputs is unknown ? In reality, the properties of AND mean this is possible since when the first input is 0, it does not matter what the other input is because the output will always be 0. Similar shortcuts exist for the other operators. For example, if either input of an OR operator is 1 it does not matter what the other one is (even if it is unknown) because the output will always be 1.

3.3.2 Selection and Concatenation

Although the RHS expression of a continuous assignment closely resembles a C expression, the fact that C is designed for writing software programs means operations we expect in hardware are absent. To address this, Verilog includes **selection**, **concatenation** and **replication** operators which split, join and replicate wires and wire vectors.

```
module add_1bit( output wire co,
                 output wire  s,
                 input  wire ci,
                 input  wire  x,
                 input  wire  y );

  assign { co, s } = x + y + ci;

endmodule
```

Listing 3.12 An implementation of a full-adder using RTL-level design features.

Given the definitions in Listing 3.3, we can split the wire vector `w1` into parts using an operator similar to array sub-scripting in C. The expression

```
w1[0]
```

selects wire number zero from the vector `w1`; the type of this expression is a 1-bit wire. If `w1` takes the value `4'b1010`, then the expression `w1[0]` evaluates to `1'b0`. We can extend this selection to include ranges so that the expression

```
w1[2:0]
```

is a 3-bit wire we have split from `w1` by selecting bits two, one, and zero. The expression evaluates to `3'b010`.

In a similar way, we can join together wires using the concatenation operator so that the expression

```
{ w0, w1 }
```

is a 5-bit wire vector. Wire number four in this vector is equal to the value of `w0` while wires three, two, one and zero are equal to the corresponding wires in `w1`. So if `w0` takes the value `1'b0` then the above expression evaluates to `5'b01010`. Note that there is something of a duality between selection and concatenation since the expression

```
w1[2:0]
```

is in fact equivalent to

```
{ w1[2], w1[1], w1[0] }
```

where in both cases we are creating a 3-bit wire vector with component wires drawn from the vector `w1`. Finally, the replication operator allows us to create repeated sequences by specifying a replication factor and a value. The expression

```
{ 4{w1[0]} }
```

is equivalent to the expression

```
{ w1[0], w1[0], w1[0], w1[0] }
```

with both evaluating to `4'b0000`.

As a demonstration of these operators in a practical context, consider a simplification of the previous gate-level full-adder design. The power of using a more

abstract description is shown in Listing 3.12. Instead of a number of very primitive module instances connected together, we use a single, expressive continuous assignment. The RHS of the assignment is easy to understand: it simply adds together the two inputs `x` and `y` and the carry-in `ci`. The LHS uses a more cryptic concatenation of two wires `co` and `s`, the result is a 2-bit wire vector. Since the result of the RHS, i.e., adding together the three input wires, is a 2-bit result. This forces the top bit into the carry-out and the bottom bit into the sum: exactly what we want. By being more expressive we not only make the module easier to read, we also open the opportunity for the Verilog tool-chain to optimise the design for us. For example, if the implementation technology offers a particularly efficient hardware structure for an adder, the tool-chain can spot the operator in our code and capitalise on this fact. In this case the advantages are perhaps marginal, but for more complex designs it can make a real difference to the end result.

3.3.3 Reduction

The second class of operation which will be alien to C programmers are those of **reduction**. Each Verilog reduction operator takes one operand and produces a 1-bit result; essentially it applies a logical operator to all the bits in the operand, bit-by-bit from right to left. Again considering the wire definitions in Listing 3.3 and assuming `w1` is set to the value `4'b1010`, the reduction expression

```
|w1
```

is exactly equivalent to the expression

```
w1[0] | w1[1] | w1[2] | w1[3]
```

which evaluates to `1'b1`. Another example could be

```
&w1
```

which is exactly equivalent to the expression

```
w1[0] & w1[1] & w1[2] & w1[3]
```

and evaluates to `1'b0`. This sort of operation does not exist in C because it is word-oriented with the programmer seldom combining bits from the same word. In Verilog however, we work more freely and can easily extract and utilise bits from any value since they are always accessible as 1-bit wires.

3.3.4 Timing and Delays

To model issues of **timing** in a model, Verilog offers several ways to specify **delay**. In the context of continuous assignment, one can utilise **regular delay** to specify the time between changes to the RHS expression and the assignment to the LHS. One

```
reg               r0;       // 1-bit unsigned register
reg        [3:0] r1;       // 4-bit unsigned register vector
reg        [0:3] r2;       // 4-bit unsigned register vector
reg        [4:1] r3;       // 4-bit unsigned register vector
reg signed [3:0] r4;       // 4-bit signed register vector
reg               r5[7:0]; // 1-bit, 8-entry register array
reg        [3:0] r6[7:0]; // 4-bit, 8-entry register vector array
```

Listing 3.13 Several different examples of register definition.

can think of this as the analogy of propagation delay whereby it takes some length of time for values to propagate through the circuit representing the RHS expression before they result in a value on the LHS. For example, the assignment

```
assign #10 r = ( x | y );
```

uses a regular delay of ten time units; the value of `r` is assigned ten time units after any change to `x` or `y`. One caveat to this simple rule is that the change in `x` and `y` must persist for at least as long as the time delay for `r` to actually see the result. If `x` changes from 1 to 0 and then back again in less than ten time units, `r` will not change to represent this change.

3.4 State and Clocked Design

So far we have only considered combinatorial circuits which hold no state but simply compute some outputs from some input values. This is partly because wires cannot hold state, their value being maintained only as long as a signal is driven onto the wire. However, we will often want to maintain some state in a design so that computation is based not only on the input values but also on previous computation.

3.4.1 Registers

To maintain state, Verilog allows the definition of **registers** that retain their value even when not being continuously driven with that value; registers are more or less analogous to variables in a C program. While wires in Verilog are literally wires in real hardware, registers are implemented using flip-flops so that the stored value is retained as long as the flip-flop is powered. While a wire is continuously updated with whatever value drives it, a register is only updated when a new value is assigned into it.

Definition of registers is essentially the same as definition of wires, one simply replaces the `wire` keyword with `reg`; Listing 3.13 describes some examples. However, the introduction of registers adds some complication in terms of how one uses

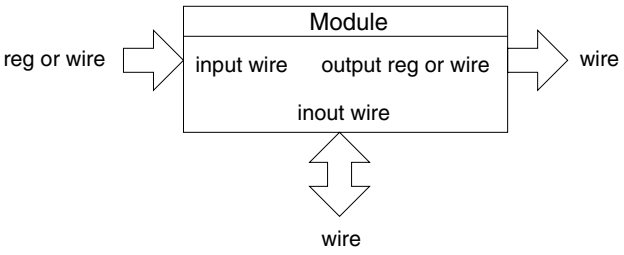

Figure 3.2 Connection rules for registers and wires.

inputs and outputs to and from modules. In short, only certain types can be used in each circumstance; Figure 3.2 describes these rules. For example, if some register `r` is defined within a module `x`, it can be named in the input port list to some module instance `y` within `x`. However, within `y` the input must be viewed as a wire: we cannot be sure that the input will always be a register (and hence retain state) so we must be pessimistic about its behaviour. In this description, we have introduced a new type of port, marked with the `inout` keyword, which can act as both an input and output.

Although one might describe a register vector above as like being an array of registers, this is not a great analogy since an array is usually a connection of separate values whereas a register vector typically represents one value. Verilog does offer "real" arrays as well, which are somewhat confusingly termed memories. The range of indices which address the memory is declared after the memory identifier as described in Line 6 and Line 7 in Listing 3.13. In these declarations we declare two memories: `r5` has eight entries numbered $7,6,\ldots,0$, each entry is a 1-bit register; `r6` also has eight entries indexed in the same way but each entry in the memory is a 4-bit register vector.

There are some constraints on where and when one can reference elements in a memory, but roughly speaking the syntax follows that of register vectors so that the expression

```
r5[2]
```

selects element 2, a 1-bit register, from memory `r5` while the expression

```
r6[1]
```

selects element 1, a 4-bit register, from memory `r6`. We can combine the two forms to select individual wires from this. The expression

```
r6[1][1:0]
```

selects element 1 from `r6` in the same way as above, then extracts bits 1 and 0 from the result to finally form a 2-bit vector.

```
begin
  x = 4'b0000;
  y = 4'b1111;
end
```

(a) Basic sequential block.

```
begin:name
  x = 4'b0000;
  y = 4'b1111;
end
```

(b) Named sequential block.

```
fork
  x = 4'b0000;
  y = 4'b1111;
join
```

(c) Basic parallel block.

```
fork:name
  x = 4'b0000;
  y = 4'b1111;
join
```

(d) Named parallel block.

Table 3.5 Examples of different types of behavioural code block.

3.4.2 Processes and Triggers

Blocks of statements are marked before and after using keywords that act in a similar way to the { and } braces in C programs. The `begin` and `end` keywords denote a **sequential block** of statements; each statement in the block is executed in the order they are listed, any timing control for a statement is relative to the previous statement. The `fork` and `join` keywords denote a **parallel block** of statements; each statement in the block is executed at the same time as the others, each timing control is relative to the start of the block. We can attach a hierarchical name to each block of code by appending a colon and identifier to the start of block keyword. Examples of each of these block definitions can be found in Table 3.5. Note that in a similar way to how { and } braces work in C programs, we can omit the start and end of block keywords for one line blocks.

All blocks must exist within a Verilog **process**. A process can be thought of as a parallel thread of execution. To rationalise this, consider using an RTL-style approach to drive values onto two wires via two continuous assignments. These assignments are always being evaluated; they are executing at the same time. How would one cope with this using a behavioural approach ? The `fork` and `join` type blocks could come in handy but more generally we should split our behavioural statements into two separate groups, or processes, that each deal with updating one of the values. The processes will execute in parallel with each other just like the continuous assignments, they simply use behavioural code rather than RTL-style constructs to implement the required functionality.

There are two types of Verilog process, one marked with the `always` keyword and one with the `initial` keyword. An `always` process is like an infinite loop in that execution of the content starts at the top, continues through the body and then restarts again at the top. In contrast, an `initial` process only executes once. Table 3.6a demonstrates the definition of an `always` process; we simply decorate

```
always
begin
  ...
end
```

(a) Untriggered `always` processes are possible but poor design practice.

```
always @ ( x )
begin
  ...
end
```

(b) Triggering by value change.

```
always @ ( posedge x )
begin
  ...
end
```

(c) Triggering by positive edge.

```
always @ ( negedge x )
begin
  ...
end
```

(d) Triggering by negative edge.

Table 3.6 Examples of different process definitions and triggers.

a normal block with the `always` keyword. Basic processes are somewhat simple in that they either execute just once or loop forever; there is no way to specify when they should be executed. We can add a **trigger**, or **sensitivity list**, to a process to specify this sort of information. Having looked at how digital logic works, examples are easy to come by: we might want a block of code to be triggered by a positive or negative signal edge or level. Table 3.6b-d demonstrate the addition of triggers which cause the associated block to execute when the value of `x` changes from anything to anything; when a positive edge in `x` is encountered, that is a change in value from 0 to 1; and when a negative edge in `x` is encountered, that is a change in value from 1 to 0. Here we have included just one signal `x` in the sensitivity list, more can be added by separating the extra signals with commas; in this case we might specify, for example, that a process is triggered when either `x` *or* `y` changes value.

Note that Table 3.6a which represents an untriggered `always` block is considered bad design practice: without some form of delay, we are saying that every iteration of the block happens at the same time. Since this is clearly impossible, we need to ensure that there is some delay within the process or, more preferably, change the design so it is clear when each iteration of the process is triggered.

3.4.3 Procedural Assignments

Within a block of code, we can place a sequence of statements which are executed when the encompassing process is triggered. The most simple example of a state-

```
begin
  x  = 10;
  y  = 20 + x;
end
```

(a) Blocking procedural assignments.

```
begin
  x <= 10;
  y <= 20 + x;
end
```

(b) Non-blocking procedural assignments.

Table 3.7 Examples of different procedural assignments.

ment is the **procedural assignment** which evaluates a RHS expression and assigns the value to an LHS expression. Such assignments come in two flavours, **blocking assignments** and **non-blocking assignments**. A blocking assignment is said to block subsequent statements in the sense that they only execute after it has completed. In contrast, a non-blocking assignments allow subsequent statements to execute at the same time as it. For example, consider the two blocks in Table 3.7. The left-hand example uses blocking assignments; the assignment to `y` only executes once the assignment to `x` is finished. This means that when the assignment to `y` executes, `x` will definitely have been assigned the value $10_{(10)}$. However, the right-hand process uses non-blocking assignments; the assignments to `x` and `y` execute at the same time. Here, this causes a problem because we can no longer guarantee that `x` will definitely have been assigned the value $10_{(10)}$ before the assignment to `y` is executed; this means `y` potentially gets the wrong result. Problems like this are avoided by considering a data-flow graph and avoiding dependent non-blocking assignments. Additionally, we should not mix blocking and non-blocking assignments within a given block of code so that it is clear when a particular assignment will occur.

Procedural assignments can incorporate timing and delay specifiers in the same way as continuous assignments. However, it is important to again note the difference between the two: continuous assignments are continually being executed while procedural assignments are only executed when their process block is triggered. Additionally, note that procedural assignments can only use registers as the LHS while continuous assignments can only use wires. This makes sense since the execution of a procedural assignment is time dependent and hence requires the LHS to retain state.

Either way, we are now in a position to actually design a meaningful module using behavioural-level Verilog. Recall that a D-type flip-flop is a component that can store 1-bit values; it has an output Q which represents the currently stored value an input D which is used to set the value when a positive edge is driven onto the enable signal *en*. Listing 3.14 replicates this interface and implements the required behaviour using two processes. Firstly, we define a 1-bit register called `t` which is used to represent the value stored within the flip-flop; we use a continuous assignment to set the flip-flop output `Q` to match the value `t`. Using an `initial` process, we ensure that `t` is set to 0 to start with. Then, we use an `always` process which is triggered whenever a positive edge is detected on the input `en`. Whenever this

```
module dtype_ff( input  wire  D,
                 output wire  Q,
                 input  wire en );
  reg t;

  assign Q = t;

  initial
  begin
    t = 0;
  end

  always @ ( posedge en )
  begin
    t = D;
  end

endmodule
```

Listing 3.14 An implementation of a D-type flip-flop using behavioural Verilog.

occurs the code within is executed; this means the value of input `D` is assigned to our register `t` and Q is updated in turn by the continuous assignment.

3.4.4 Timing and Delays

We previously saw how one can specify regular delays on continuous assignments to model propagation delay. This is also true of procedural assignment, although the semantics are slightly different. Using the following

```
#10 r = ( x | y );
```

simply specifies that the assignment should be executed 10 time units after the previous statement completed. This essentially defers execution of the entire assignment statement for some period of time. Trying to work out how timing delays interact with non-blocking forms of assignment is difficult and generally it is better not to mix the two forms.

In contrast to regular delays, **intra-assignment delay** simply defers update of the LHS. For example, by moving the timing specifier to the RHS, the assignment

```
r = #10 ( x | y );
```

executes immediately after the previous statement completes. The RHS is evaluated straight away but the assignment of the result to the LHS is only performed 10 time units later.

```
module dtype_ff( input  wire   D,
                 output wire   Q,
                 input  wire  en,
                 input  wire rst );
  reg t;

  assign Q = t;

  initial
  begin
    t = 0;
  end

  always @ ( posedge en )
  begin
    if( rst == 1 )
      t = 0;
    else
      t = D;
  end

endmodule
```

Listing 3.15 An implementation of a D-type flip-flop with reset.

3.4.5 Further Behavioural Statements

3.4.5.1 Conditional Statements

Basic conditional behaviour can be implemented almost identically to in C by using the `if` statement; this takes the form

```
if      ( x )
  statement0;
else if( y )
  statement1;
else
  statement2;
```

To decide which of `statement0`, `statement1` or `statement2` gets executed we first evaluate the expression `x` and, if the value is non-zero, `statement0` is executed. Otherwise, we carry on to the next case, evaluate `y` and test if it is non-zero; if it is, then `statement1` is executed. If neither `x` or `y` evaluates to a non-zero value then `statement2` is executed as the default case. A slight problem is introduced by considering whether the unknown value `1'bX` "means" zero or non-zero. For an `if` statement, this is resolved by defining the unknown value as zero. In this example, if `x` and `y` both evaluate to the unknown value then we end up in the default case where `statement2` is executed. Note that each single statement associated with each test can in fact be a block of statements surrounded with `begin` and `end` keywords.

As an example of the `if` statement in action, consider upgrading our D-type flip-flop design from Listing 3.14 to include a signal which zeros or resets the stored value. This alteration is shown in Listing 3.15 and simply performs a test on the

rst signal each time a positive clock edge triggers the process: if rst is equal to 1, then we set the stored value t to 0, otherwise we set it to the input D.

Long or nested if statements can quickly become hard to maintain as the number of alternatives and complexity of conditions grows. To combat this, Verilog includes a construct called case which is very similar to the switch statement in C. A case statement takes an expression which can evaluate to one of several options; depending on which of these options it does actually evaluate to, one of several statements is executed. It essentially acts like a multi-way or computed branch and takes the form

```
case( x )
  value0  : statement0;
  value1  : statement1;
  value2  : statement2;
  ...
  default : statementN;
endcase
```

The expression x, which can include operators and so on rather than being simply a wire or register, is first evaluated. If the value equals value0, then statement0 is executed; if it equals value1, then statement1 is executed and so on. At most one statement is executed, and there is no need for a break statement as is needed in C. If the value of x does not equal any of the options then the statement corresponding to the default value is executed; if there is no default option then no statements are executed. Note that we can never have more than one default options and that, like an if statement, each single statement associated with a value can be a block of statements surrounded with begin and end keywords.

A similar problem to usage of if statements is presented when considering what happens if x evaluates to the unknown or high impedance value; since the case statement performs a bit-by-bit comparison even a single bit within x which is not 0 or 1 can result in unexpected behaviour. One method to deal with such options is to explicitly list them so that, for example, if we use

```
case( x )
  2'b00   : y = 1'b0;
  2'b01   : y = 1'b1;
  2'b10   : y = 1'b1;
  2'b11   : y = 1'b0;

  2'b0x   : y = 1'bX;
  2'b1x   : y = 1'bX;
  2'bx0   : y = 1'bX;
  2'bx1   : y = 1'bX;

  default : y = 1'b0;
endcase
```

then when x evaluates to the value 2'b0X we execute the fifth statement rather than the default statement as would have happened before. The other way to deal with this problem is to use the casex and casez variations of the basic case construct. A casez statement treats all occurrences of the high impedance value as

don't care; a `casex` treats both the unknown and high impedance values as don't care. This applies to both the case expression (i.e., `x` in the above) or one of the case options (e.g., `2'b0X` in the above). By treating these values as don't care we exclude them from comparison so that, for example, the case value `2'b0X` would match the case option `2'b00` (if not previously matched) when using a `casex` statement instead of a standard `case` statement. Likewise, the case value `2'b00` would match the case option `2'bX0` (if not previously matched) when using `casex` instead of `case`.

As an example of the `case` statement in action, recall the traffic light controller we developed previously in the form of a state machine. The idea was that we stored some state and had a function which moved us to a next state on each positive clock edge; we also had an associated function to set the value of various traffic lights based on the state. Listing 3.16 sketches an implementation of this design, omitting some of the cases for brevity. The module holds the 3-bit current state in the register `S` and initialises this to zero using an `initial` block. On every positive clock edge the first `always` process is triggered; we evaluate `S` and update the value to match the required next state. Also, each time `S` changes we trigger a second `always` block which again evaluates `S` but this time sets the value of output wires corresponding to the traffic lights.

3.4.5.2 Iteration Statements

It is often useful to iterate or loop over a block of statements a given number of times. Although the `if` and `case` statements allow us to control the flow of execution, they are unsuitable for this task. Likewise, an `always` process offers some form of iteration but it is also unsuitable in the sense that it is too coarse grain: we require a set of statements similar to `do`, `while` and `for` loops available in C programs.

The `forever` loop simply performs an infinite loop over the body of statements associated with it. Although in software this is generally considered a bad idea, in hardware it can make perfect sense: for example, a keyboard waits in an infinite loop for keys to be pressed until it is eventually powered off. A `forever` loop takes the form

```
forever
begin
  ...
end
```

where the continuation dots represent the body of code which is looped over. In common with all iteration statements in Verilog, there is no way to exit from the loop prematurely. In a C loop this is performed using the `break` statement; there is no equivalent in Verilog.

The next step up in terms of complexity is a loop which instead of iterating forever, iterates a fixed number of times. The `repeat` statement, not seen in languages

```
module traffic( input  wire clk,

                output reg   Mr,
                output reg   Ma,
                output reg   Mg,
                output reg   Ar,
                output reg   Aa,
                output reg   Ag );

  reg [2:0] S;

  initial
  begin
    S = 0;
  end

  always @ ( posedge clk )
  begin
    case( S )
      0       : S = 1;
      1       : S = 2;
      2       : S = 3;
      3       : S = 4;
      4       : S = 5;
      5       : S = 6;
      default : S = 0;
    endcase
  end

  always @ ( S )
  begin
    case( S )
      0       : begin
                  Mr = 0; Ma = 0; Mg = 1;
                  Ar = 1; Aa = 0; Ag = 0;
                end

      ...

      5       : begin
                  Mr = 0; Ma = 1; Mg = 0;
                  Ar = 1; Aa = 0; Ag = 0;
                end
      default : begin
                  Mr = 0; Ma = 0; Mg = 0;
                  Ar = 0; Aa = 0; Ag = 0;
                end
    endcase
  end

endmodule
```

Listing 3.16 An implementation of the traffic light controller state machine.

like C, offers exactly this behaviour. For example, if we want to execute a body of statements thirty two times we would use

```
repeat( 32 )
begin
  ...
end
```

Note that the value that determines the number of iterations must be a constant rather than an arbitrary expression. Where an arbitrary number of iterations is required, we use a `while` loop such as

```
while( x )
begin
  ...
end
```

In this case the expression `x` is evaluated and tested before each loop iteration; if the resulting value is non-zero, then iteration continues, if it is zero then the loop terminates.

Finally, we have a `for` loop which offers a powerful but simple way to iterate over the loop body. Definition of a `for` loop uses three expressions: an initialiser, a conditional, and an updater. The idea is that the initialiser expression sets loop counters to an initial value. Before each time the loop body is executed the conditional expression is evaluated: if the resulting value is non-zero the loop is executed, otherwise it terminates. After each loop execution the update expression is evaluated to update the loop counters. An an example which executes a body of statements 32 times we could write

```
for( i = 0; i < 32; i = i + 1 )
begin
  ...
end
```

In this case we have `i = 0` as the initialiser which sets the loop counter `i` to zero; `i < 32` is the conditional which allows the loop to continue as long as `i` is less than 32; the update expression `i = i + 1` increments the loop counter each time the loop is executed. Note that unlike the equivalent statement in C, we cannot use the increment or decrement operators: we are forced to use the expression `i = i + 1`. Also note that in some sense a `for` loop is just a shorthand; we could write the above using just a `while` loop as

```
i = 0;
while( i < 32 )
begin
  ...
  i = i + 1;
end
```

As in the case of `always` blocks, there is a problem if the body of statements one iterates over has no delays in it: again we are saying that every iteration happens at the same time.

3.4.6 Tasks and Functions

We have already drawn analogies between modules in Verilog and functions in C, but have tried hard to show that they are different. Even so, it is useful to have some means of defining shared or common blocks of code rather than replicating that code wherever required. Two methods of achieving this can be used in Verilog; they are called **tasks** and **functions** respectively. In a rough sense, functions can only be combinatorial results while tasks can be fairly arbitrary blocks of code. More exactly, there are some constraints on how tasks and functions can be constructed:

- Tasks can have any number of inputs and outputs; functions must have at least one input and no (explicit) outputs.
- Tasks can contain fairly arbitrary statements including timing control; functions cannot contain timing control statements.
- Tasks can call other tasks and other functions; functions can only call other functions.

All tasks and functions are local to the module in which they are defined and can access all wires and registers defined within that module; neither can contain locally defined wires or registers.

Consider the following example function which calculates the parity of an input, i.e., whether there are an odd or even number of 1 bits set in the value:

```
function parity;
  input [7:0] x;
begin
  parity = ^x;
end
endfunction
```

After giving the function a name or identifier, the 8-bit input `x` is defined. The body of the function simply applies the XOR reduction operator to `x` and forms the return value by assigning this to a pseudo-value with the same name of the function. The function can then be called much like one would in C. The following example task demonstrates this by using the `parity` function to perform a parity check of an input `x` against a parity bit `b`:

```
task parity_check
  input  [7:0]   x;
  input        bit;
  output       err;
begin
  if( parity( x ) == bit )
    err = 0;
  else
    err = 1;
end
endtask
```

If the calculated parity of input `x` is equal to the input check bit called `bit`, then we set the output `err` to 0, otherwise we set it to 1 to mark an error has occurred.

We can then use the task within some block of code. As a slightly more contrived example, consider reading an 8-bit value and an associated check bit from some input source; having obtained all the bits we could use the task to check the parity and perform some operation as a result:

```
reg err;

always @ ( posedge clk )
begin
  x[0] = read_bit();
  x[1] = read_bit();
  x[2] = read_bit();
  x[3] = read_bit();
  x[4] = read_bit();
  x[5] = read_bit();
  x[6] = read_bit();
  x[7] = read_bit();

  bit  = read_bit();

  parity_check( x, bit, err );

  if( err )
    ...
  else
    ...
end
```

If the `parity_check` task is used often enough within the encompassing module, then we will have reduced the amount of replicated code (although not necessarily the logic that implements it) and associated risk of introducing errors. Notice here that the inputs and outputs of the task must be valid in the context of their use. For example, the procedural assignment to `err` in the task means the argument passed in must be a register.

3.5 Effective Development

Like most forms of programming, hardware design using a HDL like Verilog is more than simply knowing the language syntax and semantics. Without some experience actually using the language to produce functioning models, many of the pitfalls and intricacies will remain hidden. With this in mind, the following section attempts to convey some good design practises and features in the Verilog language that can improve the quality of code in terms of maintainability and functional correctness.

`$display( <format>, <args> ... )` Print output to the console.
`$monitor( <format>, <args> ... )` Print output whenever values change.
`$monitoron` Enable all monitor output.
`$monitoroff` Disable all monitor output.
`$stop` Halt the simulator.
`$finish` Exit the simulator.
`<value> = $random( <seed> )` Produce a random value.
`<handle> = $fopen( <filename> )` Open a file for input.
`$fclose( <handle> )` Close an open file.
`$fdisplay( <format>, <args> ... )` Direct `$display` output to file.
`$fmonitor( <format>, <args> ... )` Direct `$monitor` output to file.
`$readmemb( <filename>, <memory>, <start>)` Read ASCII binary file into memory.
`$readmemh( <filename>, <memory>, <start>)` Read ASCII hexadecimal file into memory.

Table 3.8 A list of common system tasks.

3.5.1 System Tasks

Since during simulation a Verilog model is, by definition, running on some form of controlled platform, it makes sense to allow it access to some of the resources on that platform. For example, it is useful to allow a model to issue messages to the console or access the file system to load test data or store results. Although such a facility will not be present in real hardware, the use of a **system task** as a means of accessing functionality on the host platform is a massive aid to the development and debugging cycle.

A system task is similar to a normal Verilog task except that is it executed by the simulator rather than the model. As such, they are somewhat dependent on the exact simulator used although standardised Verilog includes a number of system tasks which should always be available. Table 3.8 lists some common system tasks;

perhaps the most useful are the `$display` and `$monitor` tasks which offer a simple means of debugging.

The `$display` task is similar to the `printf` function in C in that one gives it a format string and a list of values to fill into the template. Using this task, the translated string is printed to the console of the simulator so that it can be read by a user. For example, the following block

```
begin
  x = 10;
  y = 20;

  $display( "x is %b in binary, y is %d in decimal", x, y );
end
```

uses the `%b` formatting parameter to format value `x` as a binary number and the `%d` formatting parameter to format `y` as a decimal value. The resulting string, "x is 1010 in binary, y is 20 in decimal" is printed to the console so that one can verify that `x` and `y` have the values expected. However, the use of many `$display` tasks to track the value of a wire of register can be cumbersome; often we want to just print the value whenever it changes. The `$monitor` task offers a simple way to achieve this. The syntax is similar to a `$display` task in that it takes a format string and a list of values but rather than being executed just once, the monitor task is executed whenever one of the listed value changes. Thus, using the block

```
begin
  $monitor( "x is %b in binary, y is %d in decimal", x, y );

  $monitoron;
  x = 10;
  y = 20;
  $monitoroff;
end
```

we would expect two messages to be printed to the console: one after `x` is assigned to and one after `y` is assigned to.

3.5.2 Using the Pre-processor

Constant values without an obvious meaning in a C program or Verilog program are often called "magic": they make things work but it often is not clear why their specific value is chosen. As a solution to this problem, C allows the use of a **pre-processor** with which one can manipulate the source code before it is fed to the compiler. For example, instances of defined macros are replaced with an associated value; the program source code is hence more easily understood.

Although less sophisticated than the C pre-processor, Verilog allows similar functionality through directives to the tool-chain. Perhaps the most simple example is the `` `include `` directive which allows one to include the contents of a Verilog source file in another. This practise should be used with care so that cycles of inclusion are

```
1  `define WIDTH 4
2
3  module add_nbit( output wire              co,
4                   output wire [`WIDTH-1:0]  s,
5                   input  wire               ci,
6                   input  wire [`WIDTH-1:0]  x,
7                   input  wire [`WIDTH-1:0]  y );
8
9    assign { co, s } = x + y + ci;
10
11 endmodule
```

Listing 3.17 Using pre-processor directives to replace global constants.

```
1  `define GATES
2
3  module add_1bit( output wire co,
4                   output wire  s,
5                   input  wire ci,
6                   input  wire  x,
7                   input  wire  y );
8
9  `ifdef GATES
10   wire w0, w1, w2;
11
12   xor t0( w0, x,  y  );
13   and t1( w1, x,  y  );
14
15   xor t2(  s, w0, ci );
16   and t3( w2, w0, ci );
17
18   or t4( co, w1, w2 );
19 `else
20   assign { co, s } = x + y + ci;
21 `endif
22
23 endmodule
```

Listing 3.18 Using pre-processor directives for conditional source code inclusion.

avoided and is best used, for example, to include a single file of macro definitions that are common to the entire model.

Continuing the similarity with the C pre-processor, the `` `define `` directive allows the definition of a macro which is replaced the associated value when used subsequently. Listing 3.17 shows the use of `` `define `` in action by implementing an adder whose input and output ports are wider than 1-bit. By altering the definition of the macro `WIDTH`, we can change how wide these ports are without fiddling with the main module definition. Note that when using a macro in C, one would only need to use the macro identifier `WIDTH` while in Verilog we are required to prefix the identifier with a quote to get `` `WIDTH ``.

Conditional inclusion or exclusion of Verilog code can be achieved using the `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `elsif `` and `` `endif `` directives. This sort of construct can alter the source code fed to the Verilog tool-chain by simply changing some defined constants rather than major alterations to the source code itself. List-

ing 3.18 demonstrates an example; we have two different implementations of the adder module and select which one is used in the design by simply defining or not defining the `GATES` macro. Code for whichever branch is taken is fed to the tool-chain, the branch which is not taken is omitted as if it were never there or simply commented. Since macros can be defined by the tool-chain as well as in the source code, conditional inclusion and exclusion of code can offer a neat way to configure a design for different purposes and situations.

The `‘timescale` directive is a means of changing the unit of time during simulation. This alters the units used to define timing delays without the need to scale them all by hand, a daunting task in a large design. Use of `‘timescale` follows the general format:

```
‘timescale <reference> / <precision>;
```

where the `<reference>` field determines the unit of measurement while the `<precision>` field specifies the precision to which delays are rounded. There are some constraints as to what values can be used in both fields but these are largely dependent on the simulator.

3.5.3 Parameters

Although using pre-processor directives to ease maintainability is a useful idea, often they are not powerful enough to achieve what we want. For example, consider the definition of an adder module with variable sized input and output ports as described in Listing 3.17. By using the macro `WIDTH` we can easily change the size of the ports but this is a global change: *all* `add_nbit` modules will have the *same* sized ports, what if we want one with 4-bit ports and another with 2-bit ports ?

Module **parametrisation** solves this problem by allowing the declaration of parameters for each module which can be overridden by each instance. The `parameter` keyword is used to declare a parameter with a default value. Essentially similar to a macro definition, the parameter symbol can then be used freely within the module body, although it cannot be assigned to since it is not a variable. The major difference between using a parameter is that each instance can set it to a different value. This is shown in Listing 3.19 where we parametrise the adder module and then form two instances. The instance named `t0` has the `WIDTH` parameter set to four so that the ports are four bits wide; The instance named `t1` has the `WIDTH` parameter set to two so that the ports are two bits wide. This is performed using the `defparam` keyword and utilises the Verilog hierarchical naming scheme to specify exactly which instance and parameter are being referred to.

Note that we mentioned in Section 3.2 that the form of module definition where the ports are typed inside the module body is sometimes useful: parameters are one reason why. Since the parameter is defined inside the body, one cannot use the symbol before it is defined and hence the parameter symbol cannot exist in the module header.

```
1  module add_nbit( co, s, ci, x, y );
2
3    parameter WIDTH = 0;
4
5    output wire             co;
6    output wire [WIDTH-1:0] s;
7    input  wire             ci;
8    input  wire [WIDTH-1:0] x;
9    input  wire [WIDTH-1:0] y;
10
11   assign { co, s } = x + y + ci;
12
13 endmodule
14
15 module top();
16
17   wire       t0_co, t0_ci;
18   wire       t1_co, t1_ci;
19
20   wire [3:0] t0_s, t0_x, t0_y;
21   wire [1:0] t1_s, t1_x, t1_y;
22
23   add_nbit t0( t0_co, t0_s, t0_ci, t0_x, t0_y );
24   add_nbit t1( t1_co, t1_s, t1_ci, t1_x, t1_y );
25
26   defparam t0.WIDTH=4;
27   defparam t1.WIDTH=2;
28
29 endmodule
```

Listing 3.19 A full-adder implementation whose port width is parametrised.

```
1  add_1bit t0(     w0 ,    w1 ,     w2 ,    w3 ,    w4  );
2  add_1bit t1( .co(w0), .s(w1), .ci(w2), .x(w3), .y(w4) );
3  add_1bit t2( .co(w0), .s(w1), .ci(w2), .y(w4), .x(w3) );
4  add_1bit t3(          .s(w1),          .x(w3), .y(w4) );
```

Listing 3.20 Using named port lists to instantiate a full-adder.

3.5.4 Named Port Lists

As modules get more complex, the number of input and output ports typically grows quite large. This presents a number of problems. Firstly, it becomes difficult to remember the order the ports are declared in and one can easily wrongly connect two modules as a result. Secondly, it is quite common to want to omit specific connections to complex modules because one does not use the functionality they provide. For example, we might ignore the write port of a memory module if the memory is used in a read-only mode. In this case, we would like to connect the port to null somehow.

Named port lists solve both of these problems simultaneously. Consider having to instantiate the full-adder module described in Section 3.2 and Section 3.3. Listing 3.20 shows four different ways to do this starting with the conventional method in Line 1. In fact, the first three lines are equivalent; even though the order or some

```
1  module mux2_nbit( r, i0, i1, s0 );
2
3    parameter WIDTH = 0;
4
5    output wire [WIDTH-1:0] r;
6    input  wire [WIDTH-1:0] i0;
7    input  wire [WIDTH-1:0] i1;
8    input  wire             s0;
9
10   assign r = ( s0 ? i1 : i0 );
11
12 endmodule
13
14 module top();
15
16   wire [31:0]  r;
17   wire [31:0] i0;
18   wire [31:0] i1;
19   wire        s0;
20
21   mux2_1bit t0( .r(r), .i0(i0), .i1(i1), .s0(s0) );
22
23   defparam t0.WIDTH = 32;
24
25 endmodule
```

Listing 3.21 A parametrised implementation of a 1-bit, 2-way multiplexer with 32-bit ports.

wires in Line 2 and Line 3 are different, the fact we name the ports we want to connect to ensures the instantiations are correct. The final instantiation in Line 4 omits the carry-in and carry-out ports. The carry-in input is therefore wired to zero for us by the Verilog tool-chain, while the carry-out output is simply ignored or left dangling.

3.5.5 Generate Statements

Consider the task of creating a 2-way multiplexer where the inputs and outputs are no longer 1-bit values as previous discussed, but 32-bit values. One way to go about this is clearly to simply utilise wire vectors and increase the width of the ports. We might even use parameters as shown in Listing 3.21.

Another way to do the same thing is to use a Verilog **generate statement**. A generate statement is designed to programatically and automatically generate hardware instances; the statement is a means of manipulating the design rather than something which is executed at run-time. As such, one can think of it as another form of directive. The idea is that instead of typing out thirty two instantiations of the 2-way multiplexer with 1-bit inputs, we let the Verilog tool-chain do the work for us. Listing 3.22 demonstrates the generate statement in action; the generate block encloses a `for` loop which the tool-chain executes to instantiate thirty two multiplexers, each one taking the *i*-th bit of the two inputs and producing the *i*-th bit of the result. Note that in this case, Verilog demands our loop counter is declared using the `genvar`

```
1  module mux2_1bit( output wire  r,
2                    input  wire i0,
3                    input  wire i1,
4                    input  wire s0 );
5
6    assign r = ( s0 ? i1 : i0 );
7
8  endmodule
9
10 module top();
11
12   wire [31:0]  r;
13   wire [31:0] i0;
14   wire [31:0] i1;
15   wire        s0;
16
17   genvar i;
18   generate
19     for( i = 0; i < 32; i = i + 1 )
20     begin:gen_mux
21       mux2_1bit t( .r(r[i]), .i0(i0[i]), .i1(i1[i]), .s0(s0) );
22     end
23   endgenerate
24
25 endmodule
```

Listing 3.22 A generate statement building a 1-bit, 2-way multiplexer with 32-bit ports.

type to distinguish it from types used for statements which need to be executed at run-time.

Looking at the two solutions, the parametrised version seems more sensible; certainly it does the right job and also promotes reuse of the module. However, there is a subtle design principle underlying the use of generate which goes beyond simply reducing work. The Verilog tool-chain is good at taking models and running simulation and synthesis processes on them. But like any software, it is finite in computational power and generally fallible in the face of complex inputs. In fact, a Verilog tool-chain will typically do a much better job of any optimisation when faced with simple, small constructs rather than large ones. With this in mind, creating models from lots of modules that operate on small inputs is often more desirable than using one module that operates on a single large input.

3.5.6 Simulation and Stimuli

One of the major advantages of using Verilog over raw hardware is that one can simulate and debug a circuit before it is sent through the expensive process of manufacturing. Roughly speaking, there are two types of Verilog simulator:

Cycle-based Simulator The most simple form of simulator assumes that the entire model is clocked and that change occurs synchronously at clock edges. The

simulator runs a clock and on each transition calculates the new state of the model; any changes in between the clock transitions are lost.

Event-based Simulator Roughly speaking an event-based simulator waits, watching for changes in any values within the model and then propagates those changes to elements they effect. Events occur relative to an internal clock which tracks the progress of simulation time; the current time is typically stored in a variable named `$time` which is accessible for debugging purposes.

For example, the simulator might watch the inputs to a module and, when they change, evaluate expressions to compute the output or trigger execution of a process. Complex scheduling methods determine the order such events are processed in so that simulation remains valid and to reduce the amount of computation required.

In some sense, the difference in approach represents a trade-off between simulation speed and accuracy although modern simulators of either type are effective. They can be further sub-divided into interpreted simulators versus those which perform some form of compilation on a model before simulation. Purely interpreter-based systems process the model line-by-line and as a result are slower. In contrast, compilation-based simulation translates the model into an internal, easier to deal with format before executing it. This has the advantage that the simulation can be much faster; optimisation of the internal format and a global view of the model can give significant performance benefits. An extreme example of a compiler-based simulator might translate the model into native executable code for the host machine. Obviously the drawback is that every time the model changes, it must be recompiled before simulation can restart so the internal representation matches the design.

None of the modules we have looked at thus far have been self-contained: they include input and output ports to external systems. In part it is obvious that any useful module will look like this since if it takes no input and produces no output as a side effect, we can optimise the design by replacing it with the empty module ! This sort of module with no inputs and no outputs is usually called a top-level module since it is self-contained, at the top of the hierarchy. To follow convention, we usually call the top-level module `top`; clearly it can be called anything.

No matter what simulation system is used, to simulate a model we typically need to feed it test signals and collect the outputs. This is the job of a top-level stimulus module. Such a module encompasses the design, instantiating an instance of it and stimulating the inputs in order to provoke execution. For example the stimulus might toggle the reset signal to jump-start the design, emulating the behaviour of the physical switch one would wire onto the design when implemented in real hardware.

Figure 3.3 describes diagrammatically how a stimulus for our full-adder interfaces with the module. The stimulus module instantiates the full-adder and connects the ports to nets internal to it. These nets can then be set to values in order to provoke the full-adder to produce output which can be inspected in the simulator. Listing 3.23 demonstrates an implementation of this: we instantiate the adder and then use a single process to toggle `ci`, `x` and `y` through all possible values to test the module. To verify that we are getting the right results on `co` and `s`, we use the

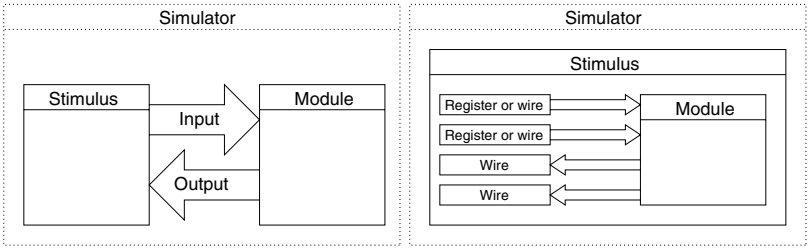

Figure 3.3 A stimulus module encompasses the design and feeds it test values.

```
1  module top();
2
3    wire co;
4    wire  s;
5    reg  ci;
6    reg   x;
7    reg   y;
8
9    add_1bit t0( .co(co), .s(s), .ci(ci), .x(x), .y(y) );
10
11   initial
12   begin
13     $monitor( "%b %b %b %b %b", co, s, ci, x, y );
14
15     #10 ci = 0; x = 0; y = 0;
16     #10 ci = 0; x = 0; y = 1;
17     #10 ci = 0; x = 1; y = 0;
18     #10 ci = 0; x = 1; y = 1;
19     #10 ci = 1; x = 0; y = 0;
20     #10 ci = 1; x = 0; y = 1;
21     #10 ci = 1; x = 1; y = 0;
22     #10 ci = 1; x = 1; y = 1;
23   end
24
25 endmodule
```

Listing 3.23 Implementation of a stimulus module for a full-adder.

`$monitor` system task to track their values as they change. The use of `$monitor` reduces the amount of typing we do but increases the verbosity of the output; we might also have used `$display` to inspect the values at specific points in execution.

3.6 Further Reading

- D. Harris and S. Harris.
  Digital Design and Computer Architecture: From Gates to Processors.
  Morgan-Kaufmann, 2007. ISBN: 0-123-70497-9.

- S. Palnitkar.
  Verilog HDL: A Guide in Digital Design and Synthesis.
  Prentice-Hall, 2003. ISBN: 0-130-44911-3.
- D.E. Thomas and P.R. Moorby.
  The Verilog Hardware Description Language.
  Springer-Verlag, 2002. ISBN: 1-402-07089-6.

3.7 Example Questions

12. Write the following as Verilog declarations:

a. An 8-bit little-endian wire vector called `a`.
b. A 5-bit big-endian wire vector called `b`.
c. A 32-bit register called `c`.
d. A signed 16-bit register called `d`.
e. A memory of 1024 elements, each 8 bits in size, called `e`.
f. A generate variable called `f`.

13. Given the declarations:

```
wire [3:0] a;
wire [3:0] b;
wire [1:0] c;
wire [3:0] d;
wire       e;

assign a = 4'b1101;
assign b = 4'b01XX;
```

what are the values resulting from the following assignments:

a. `assign c = a[1:0];`
b. `assign c = a[3:2];`
c. `assign d = a & b;`
d. `assign d = a ^ b;`
e. `assign d = { a[3:2], a[1:0] };`
f. `assign d = { a[1:0], a[3:2] };`
g. `assign c = {2{b[1]}};`
h. `assign c = {2{b[2]}};`
i. `assign e = &a;`
j. `assign e = ^a;`

14. a. Consider the following Verilog processes for appropriately defined 1-bit wires `a`, `b`, `x`, `y`, `p` and `q`:

```
always @ ( posedge p )
begin
  x <= a;
  y <= b;
end

always @ ( posedge q )
begin
  x <= b;
  y <= a;
end
```

Given that `p` and `q` are independent and may change at any time, write down **one** potential problem with this design and outline **one** potential solution.

b. Consider the following Verilog process, for appropriately defined 1-bit wire `clk` and 2-bit wire vector `state`, which implements a state machine with three states:

```
always @ ( posedge clk )
begin
  case( state )
    0 : begin
          do_0;
          state = 1;
        end
    1 : begin
          do_1;
          state = 2;
        end
    2 : begin
          do_2;
          state = 0;
        end
end
```

If this process constitutes the entirety of the design, write down **one** potential problem with it and outline **one** potential solution.

15. A Decimal Digit Detector (DDD) is a device that accepts a 1-bit input at each positive clock edge, and waits until four such bits have been received. At this point, it sets a 1-bit output signal to true if the 4-bit input (interpreted in little-endian form) is a valid Binary Coded Decimal (BCD) digit and false if it is not; it then repeats the process for the next set of four input bits.

Design a Verilog module to model the DDD device. Your design should incorporate a reset signal that is able to initialise the DDD.

16. A comparator C is a function which takes two unsigned n-bit integers x and y as input and produces $\max(x,y)$ and $\min(x,y)$ as outputs. One can think of C as sorting the 2-element sequence (x,y) into the resulting sequence $(\min(x,y),\max(x,y))$. Design a Verilog module to model a single comparator for n-bit numbers.

17. An n-bit Linear Feedback Shift Register (LFSR) called R is a register whose n-bit content is shifted right on each positive clock edge; if we use R_i to refer to the i-th bit of R, this means that R_0 is shifted out of the register then provided as output from the LFSR, and we need to create a new bit to replace R_{n-1}. The new bit is created by XORing together bits from the register according to a tap sequence; if the tap sequence is $0,2,3,5$ then the new value for R_{n-1} shifted in is $R_0 \oplus R_2 \oplus R_3 \oplus R_5$. Design a Verilog module to model an 8-bit LFSR with the tap sequence $3,5,7$. Your design should incorporate a reset signal that is able to initialise the LFSR with a seed value given as input.