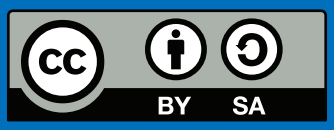
A Practical Introduction to Computer Architecture

Daniel Page ⟨dan@phoo.org⟩

git # b4055dd3 @ 2019-05-13



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.*
>
> – Wozniak

When reading the content in Chapter 2, at least two features may have been frustrating: first, the concepts and techniques were presented in a fairly theoretical way meaning that, second, any examples were limited in their scope. This Chapter attempts to resolve such issues by allowing a more practical, accessible way to experiment with hardware design. Specifically, it introduces Verilog, an industry standard Hardware Description Language (HDL). By simulating a Verilog description of some (potentially large) hardware component, one can for example quickly and definitively examine behaviour for given inputs (perhaps identifying an error in the design).

The aim is introduce just enough of the Verilog language that a reader can understand and implement concepts from Chapter 2 and elsewhere; the strategy throughout is to contrast hardware design using Verilog with the more familiar setting of software development using C. This limited remit contrasts, by design, with dedicated guides to Verilog which can (and do) fill books in their own right. In particular, we omit the study of concepts such as transistor-level design, auxiliary and related topics such as formal verification, and advanced topics such as the Programming Language Interface (PLI).

1 Introduction

1.1 The problem of design complexity

As recently as fifty years ago the components used to construct a given circuit was limited in number (in the range of 10 to 100) and in diversity (i.e., were selected from a relatively 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 documents one extreme example of what could be achieved [2, Pages 148–152]. Development of a floppy disk controller for the Apple II was, in early 1977, viewed as a vital task so 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 however, and lacking a Wozniak to deploy in every company, this model fails to scale; a modern computer processor, for example, will typically consist of many millions of components (e.g., transistors). In common with the analogous task in the context of software, producing such circuits is difficult. The first issues to bite are related to design. For example, the sheer number of inputs, outputs and states demands careful decomposition into a large number of smaller subsystems, the interface between which must be carefully managed. Even then, coping with constraints such as propagation delay becomes a major issue: the optimisation, layout and connection of components quickly becomes too difficult to perform (without error) by hand. Next, physically constructing the circuit from the design presents further challenges. For example, the size of components rules out doing this by hand: one would need

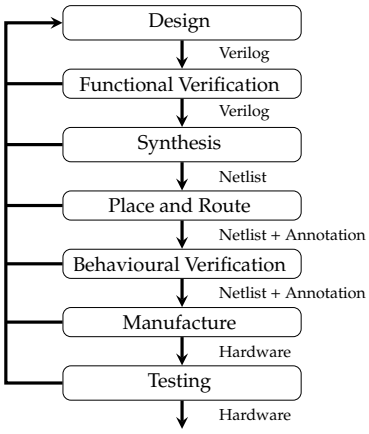


Figure 1: *A waterfall-style hardware development cycle using Verilog.*

an exceptionally tiny soldering iron to wire connections together! Finally, one is faced with the challenge of verifying the system as a whole functions correctly *and* satisfies behavioural requirements such as power consumption and heat dissipation; as circuit complexity increase, the impact of both issues are magnified.

Several key advances have helped tame 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.

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 initially 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 implementation and concentrate on design; the language makes design modularisation and reuse easier, while associated tools such as simulators make test and verification easier and more efficient. There are now several iterations of the standard language: for example, an initial version termed Verilog-95 was improved and resubmitted to the IEEE for standardisation as Verilog-2001 or IEEE 1364-2001. Standardisation, along with the production of software to produce 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 concrete 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 models 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 logic gates and the connections between them; this is somewhat equivalent to the level of detail in Chapter 2.

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 specify the required behaviour itself rather than how that behaviour is implemented.

Roughly speaking, production of physical hardware from a Verilog model is a three-phase process 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 behaviour. On one hand, the simulation is highly accurate; it may track how individual gates and signals change as time progresses for example. On the other hand, it is typically idealised in the sense that some physical constraints (e.g., the impact of propagation delay) may be ignored.

Use of simulation typically accelerates the design process versus working "on paper" alone. For example, evaluating the model in given situations (e.g., for particular inputs) can be automated, and each development cycle can therefore be relatively quick. As a result, the simulation phase is typically used to ensure a level of functional correctness before the model is processed further: if the model simulates correctly, then one can start to translate it into concrete hardware with a greater confidence of success.

Synthesise The second phase of development is similar to the compilation step in which 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 hardware components which together implement the required behaviour; this is often called a **netlist**.

Optimisation steps may be applied automatically by the synthesiser to improve the time or space characteristics of the result. Such optimisations are usually conceptually simple (e.g., eliminating redundant parts of a circuit) but repetitive: ideal fodder for a computer, but not so for a human engineer.

Implement The synthesis phase produces a list of components and a description of how they are connected together to form an implementation of the model. However, two further steps are required to translate this into concrete hardware.

First, 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. Second, 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, both steps are well suited to solution by a computer rather than a human 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. The goal of the repeated backward-facing paths is to avoid getting to the end with a defective result: the manufacture step in particular is very expensive, so fixing errors early by revisiting initial design and verification steps is vital.

2 Structural modelling

2.1 Comments

Like most other programming languages, Verilog allows **comments** in the source code description of a model. Comments have 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 be equally as complicated and hard to read as a C program, comments are a vital way to convey meaning which might not be present in the source code itself.

2.2 Wires

In Verilog, the term **net** is (roughly) used to describe a connection between components; we will cover the definition of components later but for now focus on a specific type of net. A **wire** is a specific type of net, and the exact analogy of the same thing in real life: it acts as a conducting medium that allows a signal (or value) to be carried between the two end-points. We say that a **value** is **driven** onto one end before propagating along the wire so it can be read at the other end. We define Verilog wires using a similar syntax to variable declaration in C, as shown in Figure 2: each case includes the keyword `wire`, a type and an identifier. A case-by-case description follows.

An aside: describing Verilog wires using C variables as a rough analogy.

C	Verilog
• The definition `char u` means 1. there are 8 separate 1-bit elements in `u`, but 2. `u` is typically used as 1 *single* 8-bit object rather than 8 *separate* 1-bit elements. • The definition `char v[ 32 ]` means 1. there are 32 separate 8-bit elements in `v`, and 2. `v` is typically used as 32 *separate* 8-bit elements rather than 1 *single* 256-bit object.	• The definition `wire x` means there is 1 *single* 1-bit wire in `x`. • The definition `wire [3:0] y` means there are 4 *separate* 1-bit wires in `y` and 1. `z` can be used as 1 *single* 4-bit object, *or* 2. `z` can be used as 4 *separate* 1-bit elements.

Wire definition	Meaning
`wire w0;`	A 1-bit unsigned wire called `w0`.
`wire [ 3 : 0 ] w1;`	A 4-bit unsigned wire vector called `w1`.
`wire [ 0 : 3 ] w2;`	A 4-bit unsigned wire vector called `w2`.
`wire [ 4 : 1 ] w3;`	A 4-bit unsigned wire vector called `w3`.
`wire signed [ 3 : 0 ] w4;`	A 4-bit signed wire vector called `w4`.

Figure 2: *Several different examples of wire definition.*

The first case defines a wire called `w0` capable of carrying 1-bit values. Although this is already enough for some situations, we will often want to communicate larger values. So-called **wire vectors** allow us to do exactly this by "bundling" several 1-bit wires into larger groups. The second case demonstrates this by defining a wire vector called `w1` that can carry 4-bit values. The part of the definition which reads `[ 3 : 0 ]` specifies the size of the wire vector as well as the order and labels of wires within it. More generally, `[ i : j ]` can be interpreted as "the elements in this vector are labelled i to j". Since $i > j$ this implies there are $i - j + 1$ elements in the wire vector; in this case, `w1` has $3 - 0 + 1 = 4$ elements labelled 3, 2, 1, 0. Since we can select the individual elements (i.e., 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.

The third case of Figure 2 defines another 4-bit wire vector but the order of the elements is reversed: the left-most, most-significant element of `w2` is 0 while the right-most, least-significant is 3. In the previous, second case the left-most, most-significant element was 3 while the right-most, least-significant was 0; here, since $j > i$, there are $j - i + 1$ elements in the wire vector meaning `w2` has $3 - 0 + 1 = 4$ elements labelled 0, 1, 2, 3. The fourth case uses another variation whereby the order of wires in the wire vector is the same as `w1`, but their index is changed. This time the left-most element of `w3` is numbered 4 rather than 3, while the right-most is numbered 1 rather than 0. Again, since $i > j$, there are $i - j + 1$ elements in the wire vector meaning `w3` has $4 - 1 + 1 = 4$ elements labelled 4, 3, 2, 1. These variations might seem confusing but it is crucial to see that they simply give different labels to the constituent elements. Typically the original approach (i.e., that for `w1`) is enough: the variations are useful mainly in niche situations such as compliance with the endianness requirements of some specification or third-party component.

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, two's-complement values rather than unsigned values; this is the direct analogy of so-called type modifiers in C. For example the next, fifth case in Figure 2 declares a 4-bit signed wire vector called `w4`; it can carry values between -8 and 7 inclusive. This differs from the second cases, where `w1` is declared as a 4-bit unsigned wire vector capable of carrying 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 (per Chapter 1) regardless of the wire vector type. However, adding 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`	0 (or logical low, or **false**).
`1'b1`	1 (or logical high, or **true**).
`1'bX`	Undefined (or "unknown").
`1'bZ`	High impedance (or "disconnected").

(a) *Verilog literal values.*

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

(b) *Verilog literal strengths.*

Figure 3: *Verilog literal values and their strength.*

2.3 Values and literals

In idealised hardware, a wire should only ever take the values 0 or 1 (i.e., low and high, or **false** and **true**). These values are written as the Verilog literals `1'b0` or `1'b1`. However, physical reality dictates they are not ideal; as a result Verilog supports four different values as detailed by Figure 3a.

The **undefined** 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 should be. 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: it 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 prefixes the value with one of the strength types shown in Figure 3b. However, this sort of assumption should be used with care: generally, if you rely on an undefined 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 by Chapter 2 in the context of 3-state logic. If high impedance conflicts with any other value, it instantly resolves to the other 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.

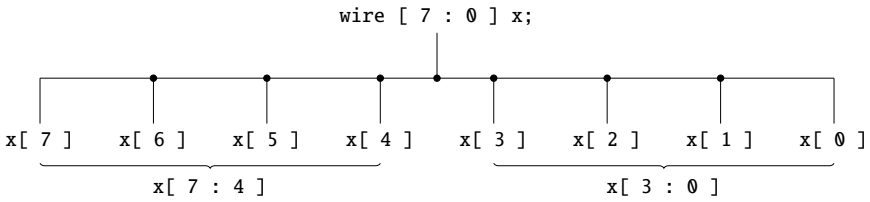
So far we have not described *why* the syntax for Verilog literals is this way. For example, what are the `1` and `b` characters in the literal `1'b0`? The general syntax can be informally described as the combination of four parts, namely

1. an optional sign, i.e., `-`, which denotes that the literal is negative (or positive if omitted),
2. an optional size field which specifies the number of bits required to represent the literal,
3. an optional base field which specifies the radix the literal is is expressed in (for example `H` or `h` for hexadecimal, `D` or `d` for decimal and `B` or `b` for binary), and
4. the literal itself

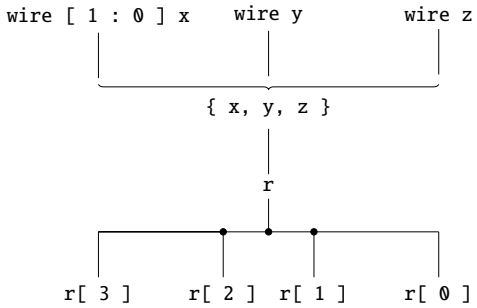
where an apostrophe character is used to separate the size and base fields. So, for example we have

- `2'b10` is a 2-bit binary value $10_{(2)}$, $2_{(10)}$ 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 $17_{(10)}$, $00010001_{(2)}$ or $11_{(16)}$.
- `-8'd10` is an 8-bit decimal value $-10_{(10)}$, represented in two's-complement.
- `3'bXXX` is a 3-bit undefined binary value (all three bits are undefined).
- `8'hZZ` is an 8-bit high impedance hexadecimal value (all eight bits are undefined).
- `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 0 and bit three is 1. 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.

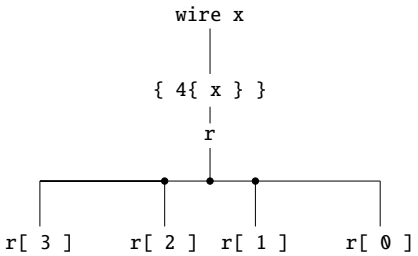
Where size is not important, one can omit the size field and accept a default (undefined bits, whether specified or not, are assumed to be zero); likewise one can omit the base field to accept the default of decimal. These features mean, for example, that the literal `10` is the same as `32'd10`, i.e., the 32-bit decimal value ten.



(a) *The Verilog subscript (or "grab") operator: given an* 8*-bit wire vector* `x` *we can access each individual* 1*-bit wire using a subscript operator (middle). Note that ranges are allowed, so we can also access contiguous groups of bits; for example, we might access the two* 4*-bit halves of* `x` *(bottom).*



(b) *The Verilog concatenate operator: we can group wires (top), or even wire vectors, into larger wire vectors (middle). Note that the effect is simply to "merge" all the wires together into one result. For example (bottom) the* 3*-rd or most-significant bit of the result* `r`*, which is a* 4*-bit wire vector, is simply the* 1*-st bit of* `x`*, and the* 0*-th or least-significant bit of* `r` *is* `z`*.*



(c) *The Verilog replicate operator: as a short-hand for specific uses of concatenation, we can make* n *"copies" of an input wire or wire vector. For example (top) the* 1*-bit wire* `x` *is replicated four times (middle) to form the* 4*-bit wire vector* `r`*: each bit of* `r` *is equal to* `x`*.*

Figure 4: *Diagrammatic descriptions of simple Verilog operators on wires and wire vectors.*

2.4 Simple operators on wires and wire vectors

Having defined some wire or wire vector, using it is fairly straightforward: we simply refer to it via the identifier we gave it, in a similar way to use of a variable in a C program. As well as this direct means of using the wire or wire vector, we can apply several operators which allow slightly more complicated uses. Crucially however, such operators simply relabel wires rather than implying any computation: the aim is simply to split a larger wire vector into smaller wires or wire vectors, or group a smaller wire or wire vectors into a larger wire vector. Examples of these two cases are shown in Figure 4a and Figure 4b respectively, with a third example in Figure 4c.

Notice that the Verilog **subscript** (or "grab") operator resembles the C array subscript operator: the aim is to specify an element within a wire vector. Verilog allows ranges in the subscript itself, so in fact we can specify more than one element. Either way, the example in Figure 4a shows that we can split the large wire vector `x` into parts, accessing the individual wires within it. If `x` has the value `8'b11110000` then we have that

- `x[ 7 ]`, `x[ 6 ]`, `x[ 5 ]` and `x[ 4 ]`, are all 1-bit wires with value `1'b1`,
- `x[ 3 ]`, `x[ 2 ]`, `x[ 1 ]` and `x[ 0 ]`, are all 1-bit wires with value `1'b0`,
- `x[ 7 : 4 ]` is a 4-bit wire vector with value `4'b1111`, and
- `x[ 3 : 0 ]` is a 4-bit wire vector with value `4'b0000`.

In contrast, the Verilog **concatenate** operator performs roughly the opposite role by grouping together wires or wire vectors; there is no clear analogy for this in C. In Figure 4b, if `x`, `y` and `z` have the values `2'b10`, `1'b1` and `1'b0` then we have that

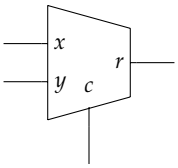
- `{ x, y, z }` is a 4-bit wire vector with value `4'b1010`,
- `r[ 3 ]` is a 1-bit wire with value `1'b1` (matching `x[ 1 ]`),
- `r[ 2 ]` is a 1-bit wire with value `1'b0` (matching `x[ 0 ]`),
- `r[ 1 ]` is a 1-bit wire with value `1'b1` (matching `y`), and
- `r[ 0 ]` is a 1-bit wire with value `1'b0` (matching `z`).

Finally, Figure 4c illustrates the **replicate** operator, which makes a specified number of "copies" of wires or wire vectors. If we say `x` has the value `1'b1`, then `{ 4{ x } }`, which is a 4-bit wire vector, has the value `4'b1111`: the number of copies is the first thing within the braces, and the thing being copied is the second. This is basically the same as if we had written `{ x, x, x, x }`, i.e., used the concatenation operator to group four copies of `x` together.

To reiterate, and as the diagrams illustrate, all any operator does is relabel elements in things we have defined into more convenient forms: in each case we can relate the value of the things we produce directly to the things we use.

2.5 Modules

Programs written in C can be thought of as collections of functions that use each other in order to perform computation: the exact 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**. In common with a function, a modules provide us with an abstraction mechanism: 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:



From a functional point of view, what the component does (e.g., how we interact with it via the inputs and outputs) is more important than how it does it (e.g., what is inside the component). The interface to the component focused more on the former: it describes only the inputs (i.e., x, y and c) and outputs (i.e., r).
A module definition is a static description, or model, of some component that we can later use; definitions consist of three parts, namely

An aside: describing Verilog modules using C functions as a rough analogy.

C	Verilog
• A program is described using static **function** definitions. • Each function has a **body** that describes how it behaves, i.e., how it computes outputs from inputs. • The functions reference each other via **calls**; a function call implies an active use. • Values are carried by **variables**, on which computation is performed by functions.	• A model is described using static **module** definitions. • Each module has a **body** that describes how it behaves, i.e., how it computes outputs from inputs. • The modules reference each other via **instantiations**; a module instantiation implies an active use. • Values are carried by **nets**, on which computation is performed by modules. • Each module instance represents some physical hardware component, and the model describes a hardware system as a whole.

```
module mux2_1bit( output wire r,
                  input  wire c,
                  input  wire x,
                  input  wire y );

  ...

endmodule
```

(a) *The "inline" module definition style.*

```
module mux2_1bit( r, c, x, y );

  output wire r;
  input  wire c;
  input  wire x;
  input  wire y;

  ...

endmodule
```

(b) *The "inside" module definition style.*

Figure 5: *Two styles of module interface for a* 1*-bit,* 2*-way multiplexer.*

1. a name or **identifier** which we can refer to the module by,
2. an **interface**, meaning the module inputs and outputs, which we call **ports** and which form the module **port list**, and
3. the **body** which determines the module behaviour, i.e., how the component does computation to produce the outputs from the inputs and any stored state.

Using a Verilog module, we can describe the interface of the multiplexer discussed above. In fact, two different styles of definition are possible (in both cases we replace the module body with continuation dots to be filled in later):

1. Figure 5a describes the port list "inline" (roughly corresponding to newer ANSI-style C function definitions), whereas
2. Figure 5b describes the port list "inside" the module (roughly corresponding to older K&R-style C function definitions).

Although the two styles are largely equivalent, for simple modules the first method is more compact and so is preferred; there are some advantages to the second style however, which we discuss later.

2.5.1 Module 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 static description. Equally, a module does not do anything until it is

instantiated. If you think of a module definition as a template, each time we instantiate the module we take the template and reproduce the component it describes. In a sense we are including a copy of it down in our model; if we instantiate a multiplexer module, this is the analogy of taking a physical multiplexer component from a box and placing it on a circuit. This implies a crucial difference between function calls and module instances: whereas a function call is transient (we call the function and once we get the result, the call is "gone"), a module instance is permanent (the components it implies always exists).

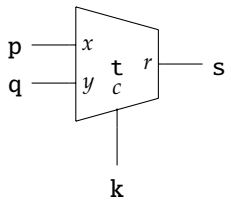
Put like this, module instantiation is how we create an active instance of some component; instantiations consist of three parts, namely

1. a **type** which specifies the module we want to instantiate,
2. an **identifier** that names the module **instance**, and
3. a list of **external wires**, i.e., connections to the internal port list of the module instance.

Again using the multiplexer example, the Verilog module instantiation on the left-hand side does basically the same thing as the diagrammatic description on the right-hand side:

```
mux2_1bit t( s, k, p, q );
```

↦



That is, we create an instance of the `mux2_1bit` module and call it `t`; the internal ports of the multiplexer, i.e., r, c, x and y are connected to the external ports `s`, `k`, `p` and `q`. Having performed the instantiation, any input we drive onto `k` can be "seen" by `t` on c, and any output `t` drives onto r can be "seen" by us on `s`.

2.5.2 Primitive, or built-in modules

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 instantiate and connect together primitive modules to describe the behaviour of more complex components (e.g., the body of our multiplexer module). The syntax is the same as modules *we* define, meaning the following correspondences hold:

`buf t0( r, x );`	↦	`r = x`
`not t1( r, x );`	↦	`r = ¬x`
`nand t2( r, x, y );`	↦	`r = x` $\overline{\wedge}$ `y`
`nor t3( r, x, y );`	↦	`r = x` $\overline{\vee}$ `y`
`and t4( r, x, y );`	↦	`r = x` $\wedge$ `y`
`or t5( r, x, y );`	↦	`r = x` $\vee$ `y`
`xor t6( r, x, y );`	↦	`r = x` $\oplus$ `y`

For example in the last case we instantiate a 2-input XOR gate, whose type is `xor`, and name it `t6`; the inputs to the gate are connected to the external wires `x` and `y`, the the output from the gate to `r`. This means we are effectively computing `r` = `x` $\oplus$ `y`.

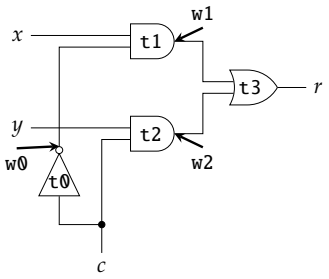
In fact, Verilog is nicer to us than simply providing a 2-way version of XOR. It provides multi-input versions of the primitive modules as well, allowing the inclusion of more inputs to the same primitive gate types as follows:

`xor t8( r, w, x, y );`	↦	`r = w` $\oplus$ `x` $\oplus$ `y`
`xor t9( r, w, x, y, z );`	↦	`r = w` $\oplus$ `x` $\oplus$ `y` $\oplus$ `z`

Again in the last case, we instantiate a 4-input XOR gate: the inputs are connected to the external wires `w`, `x`, `y` and `z`, and the output to `r`. This means we are effectively computing `r` = `w` $\oplus$ `x` $\oplus$ `y` $\oplus$ `z`.

Using primitive gates we can start filling in, i.e., implementing the behaviour of, the 1-bit, 2-way multiplexer module whose interface is described by Figure 5. We encountered the truth table and SoP-based Boolean expression in Chapter 2; the corresponding circuit is reproduced in Figure 6a, which is annotated to make some features clear by naming them. There is a one-to-one correspondence between these features and lines in Figure 6b which describes the Verilog implementation. For example,

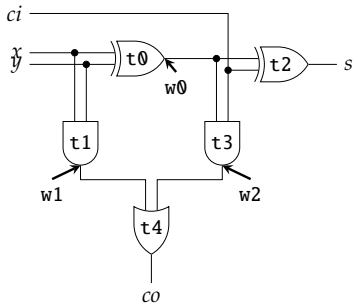
1. the first line declares three internal wires called `w0`, `w1` and `w2`,



(a) *A 1-bit, 2-way multiplexer described using a circuit diagram.*

```
module mux2_1bit( output wire r,
                  input  wire c,
                  input  wire x,
                  input  wire y );

  wire w0, w1, w2;

  not t0( w0,  c );

  and t1( w1,  x, w0 );
  and t2( w2,  y, c  );

  or  t3(  r, w1, w2 );

endmodule
```

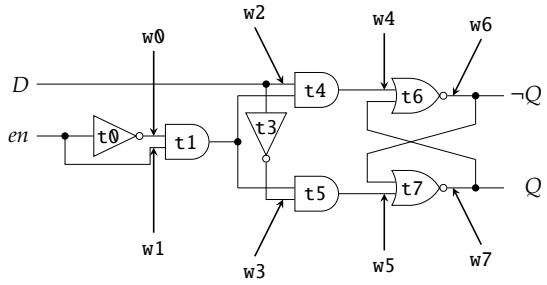
(b) *A 1-bit, 2-way multiplexer described using gate-level Verilog.*



(c) *A full-adder cell described using a circuit diagram.*

```
module fa( 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
```

(d) *A full-adder cell described using gate-level Verilog.*



(e) *A D-type flip-flop described using a circuit diagram.*

```
module dff( input  wire  en,

            input  wire   D,
            output wire   Q );

  wire w0, w1, w2, w3, w4, w5, w6, w7;

  not t0( w0,      en );
  and t1( w1, w0, en );

  buf t2( w2, D       );
  not t3( w3, D       );

  and t4( w4, w2, w1 );
  and t5( w5, w3, w1 );

  nor t6( w6, w4, w7 );
  nor t7( w7, w5, w6 );

  buf t8(  Q, w7     );

endmodule
```

(f) *A D-type flip-flop described using gate-level Verilog.*

Figure 6: *Gate-level implementations, and their diagrammatic analogues, of several building block components using primitive modules.*

2. the next line instantiates a NOT gate called `t0`, whose input is `c` and output is `w0`,
3. the next line instantiates an AND gate called `t1`, whose inputs are `x` and `w0` and output is `w1`,
4. the next line instantiates an AND gate called `t2`, whose inputs are `y` and `c` and output is `w2`,
5. the last line instantiates an OR gate called `t3`, whose inputs are `w1` and `w2` and output is `r`.

Basically all we have done is instantiate modules to represent the gates, and then connect them up per the circuit with internal wires (as well as the module inputs and outputs). Figure 6c and Figure 6d offer another example, this time for a full-adder cell. Exactly the same principles apply for the implementation in Figure 6d as with the multiplexer.

Stated in this way, Verilog module instantiation *appears* very similar to a C function call. For example, it looks a little like we call the `not` module with `c` as input to produce the output `w0`. However, each primitive gate instance executes in parallel with every other; they can be described as **continuous** in the sense that they do not wait for their inputs to be ready before computing their output. That is, they are *always* computing an output. Put another way, there are some crucial differences to keep in mind which make this analogy inaccurate:

- 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* instructions. With module instantiation, rather than being shared, *each* instance is a *separate* physical component.

  For example, the instances `t1` and `t2` in Figure 6b are both AND gates, i.e., described by the same module `and`, but represent separate physical components.

- Within a typical C program, each function call in the program is executed in *sequence* so that at any given time we know where in the program execution is: it cannot be in two functions at the same time for example. With a Verilog model, every module instance executes in *parallel* with every other instance so that one module can be doing things at the exact same time as another.

  For example, the instances `t0` and `t1` in Figure 6b are working continuously and at the same time (even though their order in the source code may imply `t0` is before `t1` somehow). We could swap the two lines over and still get a valid result: in C `w0` would be incorrectly used by `t1` before it is assigned a value by `t0`, but in Verilog the connection between `t0` and `t1` is still valid and so the wire still connects them correctly.

The continuous operation of Verilog module instances can be confusing if you are more used to the sequential nature of C. For example, what happens if some inputs to an instance have values driven onto them while others do not? The instance is always generating an output, so the output must be *something*, but it will typically be resolved as the undefined value. This also implies a challenge in terms of timing. Specifically, we already saw that propagation delay (within the gates that implement the instance) can impact on when the output becomes valid; as the size and complexity of the components we look at increases, these small delays can accumulate and present more significant delays. So even when we do drive values onto all the inputs, we need to take care the output is only used once it is valid.

2.5.3 Gate behaviours for unknown values

A somewhat subtle wrinkle exists when directly equating primitive modules to logic gates. That is, since Verilog allows wires to carry the unknown and high impedance values, i.e., `1'bX` and `1'bZ`, said modules need to act accordingly.

Fortunately, the implications of this fact are easy to reason about. First, if an input is `1'bZ`, it is treated as `1'bX`. Then, the rules in Figure 7 are applied to describe the output based on the inputs. Note the additional entries versus traditional truth tables for NOT, AND, OR and XOR; they show, for example, that if we AND together `1'b0` and `1'bX` then the result would be `1'b0`. Intuitively, and indeed in physical terms, this might seem an odd claim: how can we decide on a result if one of the inputs is unknown? In theory at least, the properties of AND mean this *is* sane because when one input is 0, it does not matter what the other input is: 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.

2.5.4 User-defined modules

Primitive modules are just like user-defined modules (except the fact they are provided by Verilog) so the syntax for instantiation is the same as well. As a result, we can consider implementing the behaviour of modules using our own user-defined modules in the same way we did above using primitive modules.

NOT1	
x	r
0	1
1	0
X	X

(a) *NOT.*

AND2		
x	y	r
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.*

OR2		
x	y	r
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.*

XOR2		
x	y	r
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.*

Figure 7: *Gate behaviours where one or more inputs are the unknown value.*

Consider for example the 1-bit, 2-way multiplexer implemented using primitive gates in Figure 6b. We can extend this in (at least) two ways by altering either the size or number of inputs: Figure 8b describes the implementation of a 4-bit, 2-way multiplexer while Figure 8d describes a 1-bit, 4-way multiplexer. In both cases, the module behaviour is implemented using instances of `mux2_1bit`: exactly the same rules apply when instantiating this user-defined module as primitive modules such as `or`. That is, we specify a type, an identifier and a port list; each instance of `mux2_1bit` operates continuously and in parallel with every other. Of course we could have implemented both modules directly using primitive modules. Ignoring which is the most efficient, reuse of the `mux2_1bit` module clearly leans on one of the advantages of Verilog: we can easily build more complex behaviour from more simple behaviour, capitalising in this case on the design strategies of replication and cascading.

One of the goals of this example is to highlight the fact that instantiation creates a **hierarchy** of instances; each instance contains instances of modules it in turn instantiates. To keep track of this hierarchy, Verilog outlines a simple naming scheme. For example, the 4-bit, 2-way multiplexer implemented in Figure 8b contains four instances of `mux2_1bit` (the 1-bit, 2-way multiplexer) named `t0`, `t1`, `t2` and `t3`. Looking back at Figure 6b, each one of these instances contains an instance of `not` called `t0`, two instances of `and` called `t1` and `t2`, and an instance of `or` called `t3`. Hierarchical naming means each primitive gate has a unique name: the `not` instance within the `mux2_1bit` instance `t1` is called `t1.t0` while the one within instance `t2` is called `t2.t0`. This naming scheme should be somewhat familiar: we use the same sort of scheme when referring to fields within C structure instances for example. In Verilog it is useful because 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 for a specific instance.

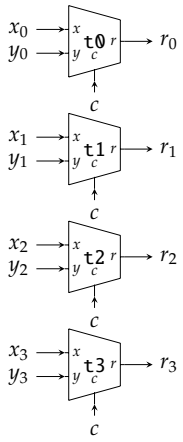
2.6 User-Defined Primitives

In some cases, describing the behaviour of a module using primitive gates can feel like hard work. In particular, this approach contradicts the goal of reducing our work by handing off as much work to the Verilog tool-chain as we can. The **User-Defined Primitive (UDP)** offers a more convenient way to implement (some) modules: we specify the the truth table and leave the implementation (i.e., how the behaviour is realised using primitive gates) to the tool-chain.

UDPs have a similar looking interface to modules. For example, we might reimplement our 2-way multiplexer as shown in Figure 9b. The body of the UDP *is* the corresponding truth table (for reference shown in Figure 9a) stating, given some combination of input, what output should be produced. Each line lists the values of the inputs (in the same order as the port list) followed by a colon and then the value required on the output. 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 `x` and `c` are 0 and `y` is any value, set the output `r` to 0".

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

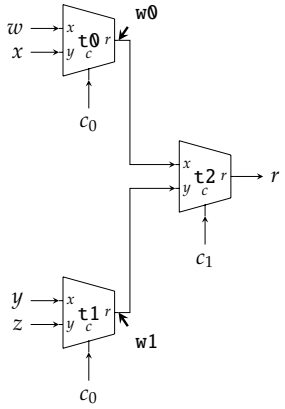
1. There can only be one output port, although there can be as many inputs as required; the output port must be specified first in the port list.



(a) *A 4-bit, 2-way multiplexer described using a circuit diagram.*

```
module mux2_4bit( output wire [ 3 : 0 ] r,
                  input  wire           c,
                  input  wire [ 3 : 0 ] x,
                  input  wire [ 3 : 0 ] y );

  mux2_1bit t0( r[ 0 ], c, x[ 0 ], y[ 0 ] );
  mux2_1bit t1( r[ 1 ], c, x[ 1 ], y[ 1 ] );
  mux2_1bit t2( r[ 2 ], c, x[ 2 ], y[ 2 ] );
  mux2_1bit t3( r[ 3 ], c, x[ 3 ], y[ 3 ] );

endmodule
```

(b) *A 4-bit, 2-way multiplexer described using gate-level Verilog.*



(c) *A 1-bit, 4-way multiplexer described using a circuit diagram.*

```
module mux4_1bit( output wire  r,
                  input  wire c0,
                  input  wire c1,
                  input  wire  w,
                  input  wire  x,
                  input  wire  y,
                  input  wire  z );

  wire w0, w1;

  mux2_1bit t0( w0, c0,  w,  x );
  mux2_1bit t1( w1, c0,  y,  z );
  mux2_1bit t2(  r, c1, w0, w1 );

endmodule
```

(d) *A 1-bit, 4-way multiplexer described using gate-level Verilog.*

Figure 8: *Gate-level implementations, and their diagrammatic analogues, of two components using user-defined modules.*

MUX2			
c	x	y	r
0	0	?	0
0	1	?	1
1	?	0	0
1	?	1	1

(a) *A 1-bit, 2-way multiplexer described using a truth table.*

```
primitive mux2_1bit( output r,
                     input  c,
                     input  x,
                     input  y );
  table
    0 0 ? : 0;
    0 1 ? : 1;
    1 ? 0 : 0;
    1 ? 1 : 1;
  endtable

endprimitive
```

(b) *A 1-bit, 2-way multiplexer described using a Verilog UDP.*

Figure 9: *A 1-bit, 2-way multiplexer described using a UDP.*

2. Both input and output ports can only be single wires; wire vectors are not allowed.

2.7 RTL-based constructs

Expressing a Verilog model purely in terms of logic gates is analogous to writing software directly in a low-level machine language: it permits a lot of control over the result, but requires a significant amount of effort by the programmer. To combat this we can use RTL, which allows us to work at a higher-level of abstraction. Clearly there is some overlap between RTL and gate-level design; a rough way to think of RTL is simply as a short-hand for combinatorial logic we have previously described long-hand using gates.

2.7.1 Continuous assignment

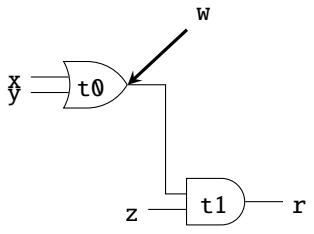
In Section 2 we described the operation of gates, realised using primitive modules, as continuous in the sense that they continuously compute outputs from their inputs rather than waiting for the inputs to be available. We can use an RTL **continuous assignment** to do the same thing. Such a statement 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 driven with (or assigned) a value produced by evaluating the RHS.

```
assign r = ( x | y ) & z;
```

↦

```
or t0( w, x, y );
and t1( r, w, z );
```

↦



Similarly to C where the LHS would be a variable, in a Verilog continuous assignment the LHS *must* be a wire or wire vector. Furthermore, but now unlike C, a Verilog continuous assignment is continuously being evaluated rather than only being executed when control-flow reaches the assignment. This means a Verilog continuous assignment is perhaps better described as (permanently) *connecting* the LHS to the output of a circuit described by the RHS.

There are two key benefits relating to the use of a continuous assignment over, for example, an equivalent gate-level description:

1. The RHS can be constructed using a rich set of high-level, C-style logical and arithmetic operators; Figure 10 includes a comprehensive list.

   The goal is to describe the RHS expression in a natural form which focuses on the required behaviour. Of course, there is still an underlying requirement for gate instances and connections, but implementation of the expression in these low-level terms is deferred to (i.e., automated by) the Verilog tool-chain.

2. Consider an example where the goal is to OR corresponding bits of two 4-bit wire vectors called `x` and `y`. Using a gate-level description, we might instantiate four gates as follows

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

Figure 10: *A list of computationally-oriented Verilog operators.*

```
module mux2_1bit( output wire r,
                  input  wire c,
                  input  wire x,
                  input  wire y );

  assign r = c ? y : x;

endmodule
```

(a) *A 1-bit, 2-way multiplexer described using a Verilog continuous assignment.*

```
module mux2_4bit( output wire [ 3 : 0 ] r,
                  input  wire           c,
                  input  wire [ 3 : 0 ] x,
                  input  wire [ 3 : 0 ] y );

  assign r = c ? y : x;

endmodule
```

(b) *A 4-bit, 2-way multiplexer described using a Verilog continuous assignment.*

Figure 11: *Two (re)implementations of user-defined multiplexer modules using RTL-based continuous assignments.*

```
or t0( r[ 0 ], x[ 0 ], y[ 0 ] );
or t1( r[ 1 ], x[ 1 ], y[ 1 ] );
or t2( r[ 2 ], x[ 2 ], y[ 2 ] );
or t3( r[ 3 ], x[ 3 ], y[ 3 ] );
```

with the i-th gate OR'ing the i-th bits of `x` and `y` together to form the i-th bit of `r`. One could easily argue this is already quite verbose, and for larger `r`, `x` and `y` increasingly so.

In contrast, using a continuous assignment we could instead simply write

```
assign r = x | y;
```

Since the Verilog tool-chain knows the types of `x` and `y`, the result is what we might intuitively expect. That is, it matches the above wrt. element-wise application of OR. Furthermore, if we change the types of `x` and `y`, say to 1-bit wire vectors, the same continuous assignment *still* gives the result expected. In fact, it *even* works if the types are mismatched: if `x` is a 4-bit wire vector and `y` is a 2-bit wire vector, `y` is padded (or extended) with two more-significant 0 bits.

Hopefully the implication is clear: using RTL, we as the Verilog developer do less work and focus on more creative tasks while avoiding tasks which are mechanical and easy to automate. In certain cases we sacrifice some control over the result, but we almost always get a trade-off in terms of increase productivity.

2.7.2 Expressions with conditional behaviour

Verilog includes a so-called ternary (or "choice") operator, which has a direct analogue in C. The operator is used with three operands (say `c`, `x` and `y`) with the first selecting between the second and third, i.e.,

$$\texttt{assign r = c ? y : x;} \quad \mapsto \quad \texttt{r} = \begin{cases} \texttt{x} & \text{if } \texttt{c} = 0 \\ \texttt{y} & \text{if } \texttt{c} = 1 \end{cases}$$

This *should* look familiar: the operator is essentially describing the same behaviour as a multiplexer, with `c` acting as the control signal. That is, used as above the RHS will be continuously evaluated meaning any change in `c`, `x` or `y` will update the LHS `r` in exactly the same was as a multiplexer we described previously using gate-level Verilog.

We can direct reimplement the 1-bit, 2-way multiplexer previously described using gates in Figure 8b and using a UDP in Figure 9b: Figure 11a shows a third, RTL-based implementation. In terms of the claimed benefit of an RTL-based approach, Figure 11b offers some neat evidence: it reimplements the 4-bit, 2-way multiplexer originally described using gates in Figure 8b. Notice that the two implementations are the same: they both use a single line continuous assignment to describe the required behaviour, and the Verilog tool-chain understands how to interpret this based on the types of, in this case, `x`, `y` and `r`.

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

  wire [ 1 : 0 ] t;

  assign t  = ci + x + y;

  assign s  = t[ 0 ];
  assign co = t[ 1 ];

endmodule
```

(a) *The full-adder described long-hand, with an extra intermediate wire* `t`.

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

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

endmodule
```

(b) *The full-adder described naturally using an LHS concatenation operator.*

Figure 12: *A full-adder cell described using a Verilog continuous assignment.*

2.7.3 Creative use of concatenation

Figure 12 describes a reimplementation of the gate-level full-adder in Figure 6d; the two different versions are intended to make it easier to understand.

1. Figure 12a makes use of a single continuous assignment to describe the required behaviour. The RHS has a fairly intuitive meaning: via the built-in Verilog plus operator, it adds together the 1-bit inputs `ci`, `x` and `y`. The result is driven onto the LHS, namely the 2-bit wire `t`.

   The assignment is valid since the type of the RHS is a 2-bit wire vector (since we are computing the sum of three 1-bit wires) and this matches the type of the LHS which is a 2-bit wire vector by definition.

   Finally, the 1-bit outputs `co` and `s`, are driven with the most- and least-significant bits of `t` respectively.

2. Figure 12b simplifies things further. Essentially the same approach is taken, but now the continuous assignment uses a more cryptic LHS formed from the concatenation of `co` and `s`.

   The types of the RHS and LHS still match: the only change is the RHS, which is the concatenation of two 1-bit wires meaning it is a 2-bit wire vector.

   By connecting the wires of the LHS and RHS together, which is all a continuous assignment does, we again get the result what we wanted but without the need of `t`: the most-significant bit of the RHS is connected to the most-significant of the LHS (i.e., to `co`); the least-significant bit of the RHS is connected to the least-significant of the LHS (i.e., to `s`).

This simple example demonstrates a few points worth making. First, there is a clear conceptual difference between Verilog and C, where the RHS of an assignment *must* be a single variable: here we are allowed to include operators in Section 2.4 since they simply relabel wires rather than perform computation. Second, it lends evidence to the claimed advantages of an RTL-based approach. In this case, the RHS expression is literally *the* most natural description of the behaviour we want.

As an aside, there is a further benefit worthy of note. If the technology we use to implement this module supports particularly efficient structures for adder cells, the Verilog tool-chain can spot the plus operator and capitalise on this fact; the same is not necessarily true if we obfuscate the behaviour we want by implementing it via a gate-level description. Such a scenario might occur, for example, if we use an FPGA (which often house dedicated blocks of logic for operations such as addition and multiplication) rather than a transistor-based standard cell library. Of course in this case the benefit is perhaps marginal, but for more complex modules it can make a real difference to the end result.

2.7.4 Reduction operators

One class of Verilog operator which will be unfamiliar to most C programmers are those which perform **reduction**. Each Verilog such operator takes an operand and produces a 1-bit result; essentially it applies this logical operator to each bit in the single operand, bit-by-bit from right-to-left. Those familiar with functional programming languages (Haskell for example) may recognise this style of operation as similar to `foldr` and `foldl`.

Consider a 4-bit wire vector called `x` which has the value `4'b1010`. The reduction expression

```
|x
```

takes each wire in `x` and places a OR operator between them. The result is is exactly equivalent to the long-hand expression

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

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

```
&x
```

which is exactly equivalent to the expression

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

and evaluates to `1'b0`. In part, this sort of operation does not exist nor make as much sense in C because the programmer less often combines bits from the same word: once some integer `x` is defined, it is typically used as a single object rather than inspecting individual bits. With Verilog however, a wire vector can be viewed simply as a collection of wires (per Section 2.2, and rather than a single object) so it is useful to extract and operate on each wire individually.

2.7.5 Timing and delays

To model the issue of **timing**, Verilog offers several ways to specify **delay**. For example, in the context of continuous assignment we can use a **regular delay** to control the time between changes to the RHS expression and the assignment to (or update of) the LHS. One can think of this as a way of modelling propagation delay, where it takes some time for values to propagate through the circuit represented by the RHS before provoking a change in the LHS. In contrast to the example at the start of this Section, the continuous assignment

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

uses a regular delay of 10 time units: a value is assigned to the LHS `r` 10 time units *after* any change to `x`, `y` or `z` in the RHS. One caveat to this simple rule is that any change to the RHS must persist for at least as long as the delay for it to be "seen" on the LHS. For example if `x` changes from `1'b1` to `1'b0` and then back again in less than 10 time units, `r` will not change correspondingly.

3 Behavioural modelling

The basic premise behind behavioural-level Verilog is description of a module body *purely* in terms of behaviour, effectively taking another step forward from RTL. Three concepts are important:

1. The behaviour of each module is at least partly described using **processes**, each of which
   (a) can be triggered to perform computation, and
   (b) operates in parallel with the others (much like a module instance does).
2. Each process is composed of **blocks** of **statements** (and only *valid* behavioural statements).
3. The statements in a block are "executed" when the containing process is triggered, i.e., they represent steps in some larger computation specified by the process.

The final concept has an implicit dependence on time and state. That is, if statements are executed in a sequence of steps, we need to maintain state between those steps: behavioural Verilog allows declaration of **registers** to satisfy this requirement.

Crucially, it is possible to "mix and match" styles of Verilog in the same description of module behaviour. For example, we might describe the behaviour of some module partly using a behavioural process and then partly using some module instances or RTL continuous assignments. Per the goals of using Verilog at all, this is attractive: it enables us to use the right level of abstraction for whatever we are doing. However, this approach must follow the rules above: since a behavioural process (or block) must be composed of behavioural statements *only*, it is not legal to mix other styles *inside* them. Any module instanciation, for example, must exist *outside* a behavioural process since this is not a behavioural statement. Performing an instanciation inside a process makes no sense: this treats module instanciation like a C function call. Rather, the instance *always* exists, rather than existing only when the process is triggered.

Register definition	Meaning
`reg r0;`	A 1-bit unsigned register called `r0`.
`reg [ 3 : 0 ] r1;`	A 4-bit unsigned register vector called `r1`.
`reg [ 0 : 3 ] r2;`	A 4-bit unsigned register vector called `r2`.
`reg [ 4 : 1 ] r3;`	A 4-bit unsigned register vector called `r3`.
`reg signed [ 3 : 0 ] r4;`	A 4-bit signed register vector called `r4`.
`reg r5 [ 7 : 0 ];`	A 1-bit, 8-entry register memory called `r5`.
`reg [ 3 : 0 ] r6 [ 7 : 0 ];`	A 4-bit, 8-entry register vector memory called `r6`.

Figure 13: *Several different examples of register definition.*

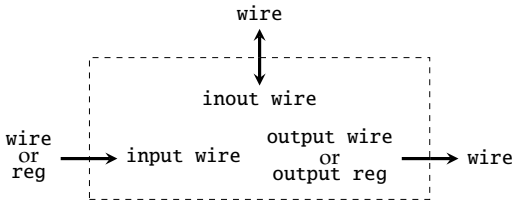


Figure 14: *Connection rules for internal and external registers and wires.*

3.1 Registers

Verilog allows the definition of **registers** which, in contrast to wires, retain their value even when not being continuously driven; while a wire is continuously updated with whatever value drives it, a register is only updated when a new value is assigned to it. In this sense, registers are more directly analogous to variables in C than wires are. Definition of registers is essentially the same as definition of wires: one simply replaces the `wire` keyword with `reg`.

Figure 13 describes several examples, with the first five acting as register versions of those previously explained in relation to wires in Figure 2.

3.1.1 Interfacing rules

The introduction of registers adds some complication in terms of inputs and outputs to and from modules. In short, only certain types can be used in each circumstance. Figure 14 describes these rules, which one can think of as analogous to rules for casting and coercion in C. The diagram tries to capture two rules, which in more human-readable terms are as follows:

1. Any input port must be a wire internally, even though externally it can be a wire or register.

   Put another way, the module must be pessimistic about how it is used: the module cannot rely on whatever instantiates it using a register as input, which means there can never be a problem with it falsely assuming the input will maintain state.

2. Any output port must be a wire externally, even though internally it can be a wire or register.

   Put another way, whatever instantiates the module must be pessimistic about how it is implemented. It cannot rely on use of a register in said implementation, which means there can never be a problem with it falsely assuming the input will maintain state.

Within this description, we have introduced a new type of port, marked with the `inout` keyword, which can act as *both* an input and output; use of this feature is not discussed further.

3.1.2 Register arrays, or memories

Although one might describe a register vector (i.e., the register version of a wire vector) as an *array* of registers, this is not a perfect analogy. In particular, this is because to access elements in a register (or wire) vector via the subscript operator, we need to supply a static index. For example, given a register vector `r` we might access it via `r[ 0 ]` where the index `0` is static (i.e., is constant, so does not change). Notice the same is not true in C array subscripts for example: we might define an array `A` and then access it via `A[ i ]` with `i` perhaps representing a loop counter.

The reason Verilog is restrictive in this case is down to efficiency: using static indexes we only every relabel individual registers (or wires) via the subscript, concatenation and replication operators. However, it also supports the use of components which act like more traditional arrays; it terms them **memories**. The reason

for this nomenclature is that their concrete implementation typically requires an actual RAM as described in Chapter **??**: this lifts the restriction wrt. static indexes, but implies a larger, heavy-weight component.

The number of elements in (resp. range of indexes used to access) the memory is declared *after* the memory identifier: this is illustrated by the final two cases in Figure 13 which define the memories `r5` and `r6`. The former, for example, has eight entries each of which is a 1-bit register; the latter also has either entries but each is now a 4-bit register vector. There are some constraints on where and when one can reference elements in such memories, 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[ 2 ]
```

selects element 2, a 4-bit register, from memory `r6`. If we have some `x` whose value is `3'b2` then we are now allowed to write

```
r6[ x ]
```

and get the same result as in the first case: we again select element 2, a 4-bit register, from memory `r6` due to the value provided by `x`. Finally, we can also combine the two forms to select individual wires from the result. For example, the expression

```
r6[ 2 ][ 1 : 0 ]
```

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

3.2 Processes

Within a module definition, behavioural statements *must* exist within a Verilog **process**: they cannot be used outside a process, and other forms of Verilog cannot be used inside a 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, and also executing at the same time. How would one cope with this using a behavioural approach? In short, we 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 statements rather than RTL-style constructs to implement the required functionality.

Figure 15 illustrates two types of Verilog process marked with the keywords `always` and `initial`; in both cases the process are named with the identifier `id`, but this is optional (for reference, it allows the process to exist within the naming hierarchy outlined in Section 2.5.4). An `always` process is like infinite loop: execution of the content starts at the top, continues through the body and then restarts again at the top. An `initial` process is executed just once; the goal is to initialise the module when powered on.

However, this behaviour is limited in the sense that we cannot control when a process executes. That is, these processes either execute just once or continually; we might ideally wish for more control, having them execute as the result of some event for example. In particular, having studied sequential logic and state machines in Chapter 2, it seems sane to trigger execution of a process using a positive or negative clock edge or level; the idea is that the process waits until such an event occurs, then executes.

This is achieved via specification of a **sensitivity list**, which can be added to an `always` process; the concept makes no sense for an `initial` process since it only ever executes once.

1. Figure 16a has no sensitivity list; in a sense it is untriggered because it simply executes without waiting for an event. There are situations where this can be useful, but usually it is considered bad design practice: we expand on this issue later when discussing timing and delay.

2. Figure 16b has a sensitivity list that includes the unannotated signal `x`. Whenever `x` changes, from anything to anything, the process is triggered and subsequently executes.

   Note that this is the *only* mechanism which makes sense if `x` is not a 1-bit wire or register: positive and negative edges only make sense for single wires or register, so if `x` is a wire or register vector we can only reason about overall (rather than per-element) change.

3. Figure 16c has a sensitivity list that includes the signal `x` annotated with the `posedge` keyword. Whenever there is a positive edge on `x` (i.e., it changes from 0 to 1), the process is triggered and subsequently executes.

4. Figure 16d has a sensitivity list that includes the signal `x` annotated with the `negedge` keyword. Whenever there is a negative edge on `x` (i.e., it changes from 1 to 0), the process is triggered and subsequently executes.

```
always begin:id
  ...
end
```

(a) *An* `always` *process.*

```
initial begin:id
  ...
end
```

(b) *An* `initial` *process.*

Figure 15: *(Incomplete) examples of Verilog process types.*

```
always begin:id
  ...
end
```

(a) *An "untriggered"* `always` *process: normally bad practise!*

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

(b) *An* `always` *process triggered by any change in* `x`*.*

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

(c) *An* `always` *process triggered by a positive edge (i.e., change from* 0 *to* 1*) on* `x`*.*

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

(d) *An* `always` *process triggered by a negative edge (i.e., change from* 1 *to* 0*) on* `x`*.*

Figure 16: *(Incomplete) examples of Verilog* `always` *processes with associated sensitivity lists.*

```
begin:id
  ...
end
```

(a) *A sequential block.*

```
fork:id
  ...
join
```

(b) *A parallel block.*

Figure 17: *(Incomplete) examples of Verilog block types.*

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

(a) *A blocking assignment.*

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

(b) *A non-blocking assignment.*

Figure 18: *(Incomplete) examples of Verilog procedural assignments.*

```
module dff( input  wire  en,

            input  wire   D,
            output wire   Q );

  reg t;

  assign Q = t;

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

endmodule
```

Figure 19: *A D-type flip-flop, implemented using behavioural Verilog.*

Note that these example include only one signal in the sensitivity list (i.e., `x`), but more generally can include a comma separated list of signals; the module is triggered by associated changes in any signal in the list. For example, we might specify that a process is triggered when *either* some `x` *or* `y` changes.

3.3 Statement blocks (or groups)

Within a process, we specify blocks of statements delineated using keywords that act in a similar way to the `{` and `}` braces in C. Figure 17 illustrates two types of block:

- Figure 17a describes a **sequential block** of statements using the `begin` and `end` keywords. Each statement in the block is executed in sequence, any specification of delay is relative to the previous statement; execution of the block is complete when the last statement in the block completes execution.
- Figure 17b describes a **parallel block** of statements using the `fork` and `join` keywords. Each statement in the block is executed in parallel, any specification of delay is relative to the start of the block; execution of the block is complete when the statement that takes the longest to execute competes execution.

As with the processes in Figure 15 both cases the process are named with the identifier `id`, but this is optional. Also note that like C, where we can omit `{` and `}` when they surround single-line statements, we can omit the start and end of block keywords in certain circumstances. From here on we try to avoid this where possible, even though the result may be less aesthetically pleasing.

3.4 Statements

Within a block, which *must* in turn exist within a process, we can write a sequence of **statements**: the statements are executed, in sequence, when encompassing process is triggered.

The syntax and semantics of behavioural Verilog statements starts to get more complicated; specific examples are not really enough to explain their general form and function. As such, we first try to explain each statement generally, using **syntax placeholders** where necessary, and *then* give some specific examples. Each placeholder shows where a particular syntactic construct *should* go; we are then free to fill them in with *any* construct of that particular type. Specifically,

- REG$_i$ is the i-th register, for example `x` or `y`,
- WIRE$_i$ is the i-th wire, for example `x` or `y`,
- LITERAL$_i$ is the i-th literal value, for example `1'b0` or `2'd3`,
- EXPRESSION$_i$ is the i-th expression composed from wire or register operands (and vectors thereof) and operators per Figure 10, for example `x | y` or `x + y`,
- STATEMENT$_i$ is the i-th statement, for example `x = y + z;`,
- the continuation dots . . . are intended to save space by indicating repetition of a particular construct.

Note the alternate font use to typeset placeholders versus actual Verilog source code, and that we omit the subscript i when only one instance of the associated type exists.

```
module slb_8bit( output wire [ 7 : 0 ]   r,
                 input  wire [ 7 : 0 ]   x,
                 input  wire [ 2 : 0 ]   y );

  wire [ 7 : 0 ] x0 = x;

  wire [ 7 : 0 ] x1 =  y[ 0 ] ? { x0[ 6 : 0 ], 1'b0 } : x0;
  wire [ 7 : 0 ] x2 =  y[ 1 ] ? { x1[ 5 : 0 ], 2'b0 } : x1;
  wire [ 7 : 0 ] x3 =  y[ 2 ] ? { x2[ 3 : 0 ], 4'b0 } : x2;

  assign r = x3;

endmodule
```

(a) *A combinatorial logarithmic shifter.*

```
module slb_8bit( input  wire            adv,

                 output wire [ 7 : 0 ]   r,
                 input  wire [ 7 : 0 ]   x,
                 input  wire [ 2 : 0 ]   y );

  reg  [ 7 : 0 ] r1a, r2a, r3a;
  reg  [ 2 : 0 ] r1b, r2b, r3b;

  wire [ 7 : 0 ] x0 = x;
  wire [ 2 : 0 ] y0 = y;

  wire [ 7 : 0 ] t1 =  y0[ 0 ] ? {  x0[ 6 : 0 ], 1'b0 } :  x0;
  wire [ 7 : 0 ] t2 = r1b[ 1 ] ? { r1a[ 5 : 0 ], 2'b0 } : r1a;
  wire [ 7 : 0 ] t3 = r2b[ 2 ] ? { r2a[ 3 : 0 ], 4'b0 } : r2a;

  assign r = r3a;

  always @ ( posedge adv ) begin
    r3a = t3;
    r2a = t2; r2b = r1b;
    r1a = t1; r1b =  y0;
  end

endmodule
```

(b) *A pipelined logarithmic shifter with* 3 *stages.*

Figure 20: *A logarithmic shifter, implemented using behavioural Verilog.*

3.4.1 Procedural assignment

The most simple, but perhaps most useful statement is the **procedural assignment**: as the name suggests, it evaluates some RHS expression and assigns the value to an LHS expression. However, and despite their name and syntactic appearance, it is crucial to keep in mind that procedural assignments are fundamentally different from continuous assignments. A simple example will illustrate this fact:

```
assign WIRE = EXPRESSION ;
```

```
always @ ( posedge clk ) begin
  REGISTER = EXPRESSION ;
end
```

The (left-hand) RTL continuous assignment is being continuously executed; any change in the RHS provokes an update to the LHS. Although the (right-hand) procedural assignment *looks* similar, there are some key differences. First, notice that the assignment is within an encompassing block which is, in turn, within an encompassing process. This means the assignment is only executed when the processes is triggered (i.e.,there is a positive edge on `clk`) and control-flow reaches the assignment. The implication is that the RHS might change at various points in time, but the LHS is only ever updated when the assignment is executed. Second, while continuous assignments must use a a wire or wire vector as their LHS, the LHS of a procedural assignment *must* be a register or register vector. Set within the description above, this should make sense: if statements are executed in sequence, the LHS must retain the value assigned to it so that value can be used later.

Even this brief introduction to procedural assignment goes quite a way toward allowing useful examples of behavioural 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*. Figure 19 replicates this interface and implements the required behaviour using two processes. First, we define a 1-bit register called `t` which is used to represent the value stored within the flip-flop; a continuous assignment connects this to the output `Q`, meaning whatever value is assigned to `t`

can be "seen" externally on `Q`. The value of `t` is initialised using an `initial` process: this executes just once, setting `t` to zero. The behaviour of the flip-flop is modelled by an `always` process triggered by `en`, the enable signal: whenever there is a positive edge one `en`, statements in the process execute in sequence. In fact, there is only one statement which updates `t` with the input `D`.

Further, Figure 20 shows two Verilog implementations of the logarithmic shifter design introduced in Chapter 2, more specifically in Figure 2.46. Recall that the goal is to left-shift some input x by a distance of y bits; Chapter 2 used this as an example of how one can form a pipeline, by describing a design that had three stages (one for each pair of shift and multiplexer). While Figure 20a shows an unpiplined implementation as a combinatorial module, Figure 20b uses the behavioural Verilog covered to pipeline the design as described previously. The implementation can be viewed roughly in three parts:

- The first part is a series of register and wire definitions; the registers `r1`, `r2` and `r3` and similar act as the pipeline registers. `r1` holds the intermediate value being shifted for example.
- The second part represents the pipeline stages, in the sense that each one of the continuous assignments takes input from one of the (sets of) previous pipeline register(s), and computes an output.
- The third part is an `always` process: when a positive edge is detected on `adv`, this triggers the block of procedural assignments which act to advance the pipeline. This is achieved by taking the output of a given stage and storing it into the subsequent pipeline register(s).

Moving on from these examples, more generally there are two flavours of procedural assignment termed **blocking** and **non-blocking**. A blocking assignment is said to block subsequent statements in the same block: they only execute once it has. In contrast, a non-blocking assignments allow subsequent statements to execute at the same time as it. For example, consider Figure 18.

- Figure 18a uses two blocking assignments; the second assignment to `y` only executes once the assignment to `x` is complete. This means that when the assignment to `y` executes, `z` will definitely have been assigned the value $10_{(10)}$.
- Figure 18b uses two non-blocking assignments; the assignments to `x` and `y` execute at the same time.

Their use requires some care. In Figure 18b for example, 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 assigned a different (or in this case wrong) value.

3.4.2 Conditional statements

Execution of a statement only occurs if control-flow reaches it; we can alter control-flow, for example skipping a statement, by utilising a similar set of conditional statements as exist in C.

`if` statements The `if` statement is the most obvious example. It takes the form of a sequence of clauses, each including a condition expression and an associated statement. For example, in

```
if ( EXPRESSION ) begin
  STATEMENT ;
end
```

a single clause associates EXPRESSION with STATEMENT. Note that for single-line statements there is no need to use the block keywords `begin` and `end`. To decide whether the statement in a clause is executed, we need evaluate the associated condition. Verilog interprets *any* non-zero result to be **true**, and zero to be **false**; although not formally zero or non-zero, the unknown and high impedance values are treated as zero. This means, for example, that

- `1'b0` and `4'b0000` are interpreted as **false** since both are zero, i.e., all bits are equal to zero,
- `1'b1` and `4'b0010` are interpreted as **true** since both are non-zero, i.e., at least one bit is not equal to zero,
- `1'bX` and `4'b00X0` are interpreted as **false** since the unknown bit is interpreted as zero,
- `4'b01X0` is interpreted as **true** since although the unknown bit is interpreted as zero, there is still at least one bit not equal to zero,
- `1'bZ` and `4'b00Z0` are interpreted as **false** since the high impedance bit is interpreted as zero,
- `4'b01Z0` is interpreted as **true** since although the high impedance bit is interpreted as zero, there is still at least one bit not equal to zero.

Armed with this, execution of the `if` statement itself simply means considering each clause in turn. If evaluation of the condition produces **true** as a result, the associated statement is executed and the `if` statement as a whole is complete; otherwise, the statement is not executed (or skipped) and we move onto the next clause. This means that although more than one condition *may* evaluate to **true**, the fact we stop considering clauses once one is satisfied means at most one statement is executed. In the example above therefore, since there is only one clause, we evaluate CONDITION and if the result is non-zero we execute STATEMENT, otherwise we skip it.

We can add at most one, but optionally none as above, default clauses as per the following example:

```
if ( EXPRESSION ) begin
  STATEMENT0;
end else begin
  STATEMENT1;
end
```

Now we have two clauses, i.e.,

1. an explicit condition EXPRESSION associated with STATEMENT_0, and
2. a default condition associated with STATEMENT_1

but essentially the same approach applies in terms of execution. That is, EXPRESSION is first evaluated: if the result is non-zero we execute STATEMENT_0. Otherwise, i.e., if evaluation of EXPRESSION produced a zero result, we continue with the next clause: since the default condition is the same as having an explicit condition that always evaluates to **true**, STATEMENT_1 is executed.

Clearly one of the clause statements can, in turn, be an `if` statement; this is sometimes called a **nested `if`**. For example we might write

```
if ( EXPRESSION0 ) begin
  if ( EXPRESSION1 ) begin
    STATEMENT0;
  end
end else begin
  STATEMENT1;
end
```

to specify an "outer" `if` statement with two clauses, and an "inner" `if` statement with one. Likewise, we can extend the number of causes in a single `if` statement; this is sometimes called a **cascaded `if`**:

```
if          ( EXPRESSION0 ) begin
  STATEMENT0;
end else if ( EXPRESSION1 ) begin
  STATEMENT1;
end else begin
  STATEMENT2;
end
```

There are three clauses:

1. an explicit condition EXPRESSION_0 associated with STATEMENT_0,
2. an explicit condition EXPRESSION_1 associated with STATEMENT_1, and
3. a default condition associated with STATEMENT_2.

As above, EXPRESSION_0 is evaluated first: if the result is non-zero, STATEMENT_0 is executed. Otherwise we continue with the next clause by evaluating EXPRESSION_1: if the result is non-zero, STATEMENT_1 is executed. Finally, if neither EXPRESSION_0 nor EXPRESSION_1 evaluate to non-zero results then we reach the default condition, which is always satisfied, and therefore STATEMENT_2 is executed.

`case` statements `if` statements can become hard to understand and maintain when the number and complexity of clauses grows. To combat this, Verilog includes a `case` statement which is similar in form and function to a C `switch` statement.

A single expression selects between one of several clauses. The condition is first evaluated, and then the result is matched against literal values provided in each clause; the matching process is applied to the clauses in turn, and performed element-wise on the result and values. For example,

- `1'b0` matches `1'b0` but not `1'b1`,
- `1'b1` matches `1'b1` but not `1'b0`,
- `2'b01` matches `2'b01` because all (i.e., both) the elements match, but not `2'b00`, `2'b10`, or `2'b11` because there is a mismatch in at least one element for each case.

```
module traffic( input  wire clk,

                input  wire rst,

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

  reg [ 2 : 0 ] Q;

  initial begin
    Q = 0;
  end

  always @ ( posedge clk ) begin
    case( Q )
      3'd0    : Q = rst ? 3'd6 : 3'd1;
      3'd1    : Q = rst ? 3'd6 : 3'd2;
      3'd2    : Q = rst ? 3'd6 : 3'd3;
      3'd3    : Q = rst ? 3'd6 : 3'd4;
      3'd4    : Q = rst ? 3'd6 : 3'd5;
      3'd5    : Q = rst ? 3'd6 : 3'd0;
      3'd6    : Q = rst ? 3'd6 : 3'd0;
      default : Q =               3'd0;
    endcase
  end

  always @ ( Q ) begin
    case( Q )
      3'd0    : begin Mr = 0; Ma = 0; Mg = 1; Ar = 1; Aa = 0; Ag = 0; end
      3'd1    : begin Mr = 0; Ma = 1; Mg = 0; Ar = 1; Aa = 0; Ag = 0; end
      3'd2    : begin Mr = 1; Ma = 0; Mg = 0; Ar = 0; Aa = 1; Ag = 0; end
      3'd3    : begin Mr = 1; Ma = 0; Mg = 0; Ar = 0; Aa = 0; Ag = 1; end
      3'd4    : begin Mr = 1; Ma = 0; Mg = 0; Ar = 0; Aa = 1; Ag = 0; end
      3'd5    : begin Mr = 0; Ma = 1; Mg = 0; Ar = 1; Aa = 0; Ag = 0; end
      3'd6    : begin Mr = 1; Ma = 0; 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
```

Figure 21: *An implementation of the traffic light controller from Chapter 2.*

Equal		
x	y	r
0	0	1
0	1	0
1	0	0
1	1	1
X	0	0
X	1	0
X	**X**	1
X	**Z**	0
Z	0	0
Z	1	0
Z	**X**	0
Z	**Z**	1

(a) *A* `case` *statement.*

Equal		
x	y	r
0	0	1
0	1	0
1	0	0
1	1	1
X	0	1
X	1	1
X	**X**	1
X	**Z**	1
Z	0	1
Z	1	1
Z	**X**	1
Z	**Z**	1

(b) *A* `casex` *statement.*

Equal		
x	y	r
0	0	1
0	1	0
1	0	0
1	1	1
X	0	0
X	1	0
X	**X**	1
X	**Z**	1
Z	0	1
Z	1	1
Z	**X**	1
Z	**Z**	1

(c) *A* `casez` *statement.*

Figure 22: *Matching (or equality comparison) where one or more inputs are the unknown or high impedance values.*

If there is a match between result and literal, the associated statement is executed, and the `case` statement as a whole is complete. It is invalid to include the same value in different clauses of the same `case` statement; this implies that at most one statement is executed, because only one clause can be satisfied.

For example, in the following

```
case( EXPRESSION )
  LITERAL0 : begin
    STATEMENT0;
 end
  LITERAL1 : begin
    STATEMENT1;
 end
endcase
```

two clauses are selected between by EXPRESSION:

1. an explicit value LITERAL_0 associated with STATEMENT_0, and
2. an explicit value LITERAL_1 associated with STATEMENT_1.

Note that for single-line statements there is no need to use the block keywords `begin` and `end`, and that unlike C there is no need to "break" out of a clause. EXPRESSION is first evaluated: if the result matches LITERAL_0 then STATEMENT_0 is executed, if it matches LITERAL_1 then STATEMENT_1 is executed, otherwise no statement is executed.

We can add at most one, but optionally none as above, default clauses as per the following example:

```
case( EXPRESSION )
  LITERAL0 : begin
    STATEMENT0;
  end
  LITERAL1 : begin
    STATEMENT1;
  end
  default : begin
    STATEMENT2;
  end
endcase
```

three clauses are selected between by EXPRESSION:

1. an explicit value LITERAL_0 associated with STATEMENT_0,
2. an explicit value LITERAL_1 associated with STATEMENT_1, and
3. a default value associated with STATEMENT_2.

Again, EXPRESSION is first evaluated: if the result matches LITERAL_0 then STATEMENT_0 is executed, if it matches LITERAL_1 then STATEMENT_1 is executed, otherwise STATEMENT_2 is executed because the default value is like a "wildcard" that matches anything.

As with `if`, the unknown and high impedance values need to be considered when matching. In short these values match themselves only, but variants of `case` alter this behaviours. In particular, `casex` treats

the unknown and high impedance values as don't care, meaning a match with them is **true**; `casez` does the same thing but for high impedance only. Figure 22 expands on this in detail; the rules are still performed element-wise, so for example

- using `case`, `4'b0110` matches `4'b0110` but not `4'b0100`, `4'b00X0` or `4'b00Z0`,
- using `casex`, `4'b0110` matches `4'b0110`, `4'b00X0` and `4'b00Z0` but not `4'b0100`,
- using `casez`, `4'b0110` matches `4'b0110`, and `4'b00Z0` but not `4'b0100` or `4'b00X0`.

As an example of the `case` statement in action, recall the traffic light controller developed previously in Chapter 2. Figure 21 sketches an implementation in roughly three parts:

1. a 3-bit register vector called `Q` which represents the state of the FSM and which is initialised to zero using an `initial` process,
2. an `always` process which models δ, the transition function, and
3. an `always` process which models ω, the output function.

The `clk` signal is used to update the FSM: every positive edge of `clk` triggers the first `always` process. It updates the state, using a `case` statement, by assigning a new value to `Q` based on the current value. For example, if the current value of `Q` matches `3'd1` then the new value will be `3'd2`. Every time `Q` changes, the second `always` process is triggered, and the outputs (i.e., the registers representing the traffic lights) and set accordingly; again a `case` statement is used to do this.

3.4.3 Iteration statements

A loop statement is associated with a body statement; the aim is basically to repeatedly execute (or iterate over) the body, with the rule which determines how many repetitions (or iterations) depending on the loop type. In a sense, an `always` process offers some form of iteration but it is unsuitable for the same task in that it is too coarse-grain: we want a set of loop statements similar to the `do`, `while` and `for` loops available in C.

`forever` statements The most basic Verilog iteration statement is an infinite loop which simply iterates forever. In software, this behaviour might be considered undesirable; usually when a loop fails to terminate, a bug is the likely cause. In hardware on the other hand, components often behave like this: for example a keyboard operates in an infinite loop, processing key presses until it is eventually powered-off.

The statement is written using the `forever` keyword followed a single-line statement or block of statements as follows:

```
forever begin
  STATEMENT;
end
```

In common with all iteration statements in Verilog, there is no way to exit from the loop prematurely; in C one might "break" out of the loop, but there is no equivalent in Verilog.

`repeat` statements The next step, represented by the `repeat` statement, allows a bounded, fixed number of iterations. For example, the loop

```
repeat ( LITERAL ) begin
  STATEMENT;
end
```

is controlled by LITERAL: this is the fixed number of iterations performed, so if we write

```
repeat ( 32 ) begin
  STATEMENT;
end
```

then the loop body, i.e., STATEMENT, is executed thirty two times.

`while` statements In a sense, the iteration count used by a `repeat` statement is an expression: we simply constrain what form the expression can take, forcing it to be a literal value. If we relax this constrain and allow the expression to be more general, we get a loop which can iterate an unbounded number of times.

For example, execution of the `while` loop

```
while ( EXPRESSION ) begin
  STATEMENT;
end
```

proceeds as follows:

- at the start of each iteration, EXPRESSION is evaluated; if the result is zero then the loop terminates, i.e., execution of the loop is complete,
- otherwise the loop body, in this case STATEMENT, is executed and we start again for another iteration.

`for` statements Finally, the most complicated but also most expressive loop mirrors the C `for` loop. In addition to the loop body, use of a Verilog `for` loop requires an initialiser statement (e.g., to set a loop counter to zero), a condition expression (e.g., to test whether the counter is below some limit), and an update statement (e.g., to increment the counter). These are illustrated by

```
for ( STATEMENT0; EXPRESSION; STATEMENT1 ) begin
  STATEMENT2;
end
```

which, in a sense, is just a short-hand for

```
begin
  STATEMENT0;

  while ( EXPRESSION ) begin
    STATEMENT2;
    STATEMENT1;
  end
end
```

As such, execution of the loop proceeds as follows:

- before execution, STATEMENT_0 (the initialiser statement) is executed,
- at the start of each iteration, EXPRESSION (the condition expression) is evaluated; if the result is zero then the loop terminates, i.e., execution of the loop is complete,
- otherwise the loop body, in this case STATEMENT_2, is executed followed by STATEMENT_1 (the update statement) and we start again for another iteration.

Returning to a previous example, the following

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

executes STATEMENT thirty two times. We start by initailising a counter `i` (a previously defined register) to zero; at the start of each iteration we check whether `i` is less than thirty two and execute STATEMENT if so. At the end of each iteration, we update `i` by adding one to it. This means `i` ranges from zero to thirty one, making thirty two iterations in total.

Note that unlike the equivalent statement in C, we cannot use the pre- or post-increment or decrement operators in Verilog: we are forced to update `i` via the assignment `i = i + 1` instead.

3.4.4 Timing and delays

Timing control We saw previously how attaching regular delays to continuous assignments modelled propagation delay; the basic idea was to specify the length of time a change in the RHS took to affect a change in the LHS. A similar concept can be applied to behavioural statements; the syntax is similar, but the semantics are slightly different. Consider the following two cases:

- Within a sequential block such as

  ```
  begin
    #10 STATEMENT0;
    #20 STATEMENT1;
    #30 STATEMENT2;
  end
  ```

associating a regular delay associated with a statement specifies it should execute a prescribed number of time units after the previous one finishes. Here for example,

  - STATEMENT$_0$ executes 10 time units after the start of the block,
  - STATEMENT$_1$ executes 20 time units after STATEMENT$_0$ finishes (i.e., 30 time units after the start of the block),
  - STATEMENT$_2$ executes 30 time units after STATEMENT$_1$ finishes (i.e., 50 time units after the start of the block).

- Within a parallel block such as

```
fork
  #10 STATEMENT0;
  #20 STATEMENT1;
  #30 STATEMENT2;
join
```

associating a regular delay associated with a statement specifies it should execute a prescribed number of time units after the start of the block. Here for example,

  - STATEMENT$_0$ executes 10 time units after the start of the block,
  - STATEMENT$_1$ executes 20 time units after the start of the block, (i.e., 10 times units after STATEMENT$_0$ finishes)
  - STATEMENT$_2$ executes 30 time units after the start of the block, (i.e., 10 times units after STATEMENT$_1$ finishes).

In common with the semantics of regular delay of continuous assignments, 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 procedural assignment is difficult, and generally it is better not to mix the two forms (without good reason).

In contrast to regular delays, **intra-assignment delay** defers update of the LHS. Taking the previous assignment and moving the timing control to the RHS, we get

```
begin
  REG0 = #10 EXPRESSION0;
  REG1 = #20 EXPRESSION1;
  REG2 = #30 EXPRESSION2;
end
```

Each assignment executes *immediately* after the previous statement completes: each RHS is evaluated straight away. However, the corresponding LHS is only updated after the corresponding delay; focusing on the first assignment for example, the LHS `a` is assigned to 10 time units *after* the statement is executed (and the RHS is evaluated).

`wait` statements In Section 3.2 we introduced the notion of a sensitivity list whereby execution of a process could be triggered based on some event. However, this is a coarse-grained mechanism: it allows us to trigger entire processes only, rather than individual statements for example. Verilog does include a more fine-grained analogue however; a `wait` statement

```
wait ( EXPRESSION ) begin
  STATEMENT;
end
```

basically blocks (or pauses) execution until EXPRESSION evaluates to a non-zero result, at which point STATEMENT is executed. In a sense therefore, a `wait` statement triggers execution of STATEMENT based on an event captured by EXPRESSION.

Preventing uncontrolled iteration In Section 3.2, we described an `always` process as an infinite loop: the associated block of statements was described as being executed (or iterated over) again and again. Now we have introduced enough to be more specific about the claim that an "untriggered" `always` process is normally bad practise: if there is no trigger and also no delay within the associated block, we are sort of saying that every iteration happens at the *same* time (since the process itself does not wait). The same is true of all the iteration statements (infinite loop or otherwise) described in Section 3.4.3: if there is no delay within the loop body, we are sort of saying that every iteration happens at the *same* time.

In both cases, this behaviour is usually undesirable. Fortunately, the solution can be implemented by following some simple rules:

- for an `always` process, make sure there is either an associated sensitivity list (so the process waits until triggered rather than iterating immediately) or some form of delay within the associated block (so execution does not complete instantaneously), and
- for an iteration statement, make sure there is some form of delay within the associated loop body (so execution does not complete instantaneously).

3.5 Tasks and functions

We have already drawn analogies between modules in Verilog and functions in C, but also tried to show how they are fundamentally different. Even so, it *is* attractive to have something like a function or sub-routine so that we can define and reuse common or shared functionality rather than replicating it wherever required; in Verilog, **tasks** and **functions** fill this role.

Tasks and functions are both defined using a similar syntax, which in turn is similar to a module definition (if one replaces `module` and `endmodule` with the appropriate keywords) and includes

1. a name **identifier** which we can refer to it by,
2. an **interface**, meaning the inputs and outputs, and
3. the **body** which determines the behaviour.

Both are local to (i.e., can only be used by) the module they are defined in, both can access all wires and registers defined within said module and can define their own wires and registers (and vectors thereof). Despite these similarities, some major differences then constrain when and how each construct can be used:

Tasks	Functions
Functions are pure in the sense that they cannot have any side effects; they are typically used to represent combinatorial circuits. Specifically, they • are defined using the `function` and `endfunction` keywords, • can only invoke other functions, • can be invoked as part of an expression, • must have at least one input and no (explicit) outputs because there is always one implicit output, • cannot contain delays.	Tasks represent more arbitrary functionality; they are typically used to represent combinatorial or sequential circuits, in a sense acting as local modules that capture some behaviour. Specifically, they • are defined using the `task` and `endtask` keywords, • can invoke other tasks and other functions, • cannot be invoked as part of an expression (rather invocation is more like a statement), • any number of inputs and outputs, • can contain fairly arbitrary statements including delays.

3.5.1 Functions

Figure 23 includes three example Verilog functions; keep in mind that really, these would need to be defined *within* some module.

A good example is Figure 23c: for each i-th bit for $0 \leq i < 8$, the function computes the majority of inputs `x`, `y` and `z`. That is, in an element-wise manner it decides whether two or more of `x`, `y` and `z` have their i-th bit set to one (for each i). Notice that the function interface explicitly names the inputs `x`, `y` and `z` and their type: these are each 8-bit inputs. However the output is implicitly called `majority` (to match the function identifier); the type of this output is specified inline with the identifier. Contrast this, where `majority` produces an 8-bit output, with Figure 23a where there is no type, meaning that `parity` produces a 1-bit output. The function body in this case includes just one statement: an assignment to `majority`, the output.

After definition, a function can be invoked much like one would in C, e.g., as the RHS of a continuous assignment

```
assign r = parity( x );
```

or to form part of an expression as per Figure 24a; in either case, the function represents a combinatorial circuit.

```
function parity;
  input [ 7 : 0 ] x;

  begin
    parity = x[ 0 ] ^ x[ 1 ] ^ x[ 2 ] ^ x[ 3 ] ^
             x[ 4 ] ^ x[ 5 ] ^ x[ 6 ] ^ x[ 7 ] ;
  end
endfunction
```

(a) *An implementation of the* `parity` *function which computes the number of bits in* `x` *set to one.*

```
function [ 31 : 0 ] byteswap;
  input [ 31 : 0 ] x;

  begin
    byteswap = { x[  7 :  0 ], x[ 15 :  8 ],
                 x[ 23 : 16 ], x[ 31 : 24 ] };
  end
endfunction
```

(b) *An implementation of the* `byteswap` *function which swaps the order of the four bytes in* `x`*.*

```
function [ 7 : 0 ] majority;
  input [ 7 : 0 ] x;
  input [ 7 : 0 ] y;
  input [ 7 : 0 ] z;

  begin
    majority = ( x & y ) | ( y & z ) | ( x & z );
  end
endfunction
```

(c) *An implementation of the* `majority` *function which computes, in an element-wise manner, whether two or bits of* `x`*,* `y` *and* `z` *are set to one.*

Figure 23: *Some example Verilog functions.*

```
task parity_check;
  output           r;
  input  [ 7 : 0 ] x;
  input            y;

  begin
    if ( parity( x ) == y ) begin
      r = 1;
    end else begin
      r = 0;
    end
  end
endtask
```

(a) *An implementation of the* `parity_check` *task which uses the* `parity` *function to test whether the parity of* `x` *is* `y`*.*

```
task apply_test;
  input ci;
  input  x;
  input  y;

  begin
    t_ci = ci;
    t_x  =  x;
    t_y  =  y;

    #10 $display( "%b %b %b %b %b", t_co, t_s, t_ci, t_x, t_y );
  end
endtask
```

(b) *An implementation of the* `apply_test` *task which takes several inputs, copies their value into some global registers and then displays said global registers.*

Figure 24: *Some example Verilog tasks.*

3.5.2 Tasks

Figure 24 includes two example Verilog tasks; keep in mind that really, these would need to be defined *within* some module.

In contrast to the functions above, when there is an output (e.g., the `parity_check` task in Figure 24a) this is named explicitly in the interface. Unlike a function, this means a given task *must* be invoked as a statement rather than form part of an expression. For example, a (contrived) use of `parity_check` within some block might read as follows (assuming definitions of `r0` through to `r3`):

```
begin
  parity_check( r0, 8'b00000000, 1'b0 );
  parity_check( r1, 8'b00000001, 1'b0 );
  parity_check( r2, 8'b00000011, 1'b0 );
  parity_check( r3, 8'b00000111, 1'b0 );
end
```

In each line, the shared description of behaviour is reused: first to check whether the parity of `8'b00000000` is `1'b0`, then whether the parity of `8'b00000001` is `1'b0` and so on.

4 Effective development

Like most forms of programming, effective hardware design using Verilog demands more than just knowledge of the language syntax and semantics. An often quoted maxim is that to learn programming one has to *do* programming rather than have it taught; the implication is that without practical experience, many pitfalls and intricacies will be masked.

With this in mind the following Section attempts to cover useful features in the Verilog language, and more general techniques, that can help bridge the gap towards writing practical and maintainable models.

4.1 A rough guide to simulation

A Verilog simulator is simply a software tool; it "executes" an abstract HDL model for us *before* the costly process of producing a concrete implementation. That is, the simulator takes a HDL model and, given some input, tries to work out what happens in the corresponding circuit at each point in time. Using it, we can inspect the outputs (and also any intermediate values within the circuit) computed, and work out whether our model is in fact correct, i.e., whether it has functional "bugs" that would prevent the concrete hardware working as expected.

In a very rough sense, simulators can be classified in terms of how they achieve their goal:

Cycle-based simulators do not work out behaviour in a detailed way, but rather just try to work out what the state is at regular intervals in time. For example, if a value changes numerous times in one interval this is not visible: only the value at the end of the interval is worked out.

Event-based simulators maintain an internal clock that keeps track of time, but works out the state each time an event is triggered. Each time a value is changed somewhere (e.g., an input), the simulator works out what impact this has; this means that every intermediate change is visible.

Both classes can be subdivided into so-called compiler-based simulators that first translate the HDL model into a platform dependent internal format, or interpreter-based simulators that look at the HDL model line-by-line; usually the former is more efficient, i.e., completes a given simulation faster, than the latter.

4.2 System tasks and functions

Since a Verilog simulator is simply some software, it executes on some host platform, e.g., your workstation. It is often attractive to allow access to resources on said platform by the Verilog model being simulated. For example it might be useful to allow a model to display messages, accept input from the user, or access the file system to load or store data used for testing. Of course the resource will not necessarily be present when the model is implemented in concrete hardware, but until then the facility can be of massive benefit to the underlying goals of development and functional verification.

The concept of a **system task** (resp. **system function**) acts as an interface between the model and the simulator. You can think of a system task (resp. function) as being like a user defined task (resp. function) in terms of how it is invoked within the model, but the implementation is provided "behind the scenes" by the simulator rather than the model itself.

On one hand, their availability and function depends somewhat on the exact simulator used; a reasonable analogy is that of an operating system that manages access to some resource, via a system call, on behalf of

a user process. On the other hand, Verilog standards includes a number of system tasks and functions which should *always* be available. A useful selection of these standard system tasks are discussed below; note that the identifiers of each system task or function is prefixed by a dollar character to distinguish them from user defined identifiers.

4.2.1 The `display` task

The `display` system task is similar to `printf` in C; the idea is that a programmer invokes the task by passing it a list of arguments and a "format string". Inside the format string are a number of format specifiers which are filled, in order, by the arguments; each specifier details how the corresponding argument should be formatted (e.g. in decimal, with leading zeros and so on). The format string is therefore translated by `display` into the result that is written to the simulator console.

Although there are several format specifiers, three are most useful:

- `%s`, translates an argument into a string,
- `%b`, translates an argument into a binary string,
- `%d`, translates an argument into a decimal string, and
- `%h`, translates an argument into a hexadecimal string.

For example, the following (contrived) block of statements

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

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

uses the `%b` format specifier to translate the argument `x` into a binary string, and `%d` to translate `y` into a decimal string; the result "x is 1010 in binary, y is 20 in decimal" is printed to the simulator console meaning we can verify `x` and `y` have the expected values.

4.2.2 The `monitor` task

Use of `display` can quickly get cumbersome; often we want to just write the value whenever it changes rather than have to manually embed invocations of `display` where we think it *might* have changed. The `monitor` task offers a simple way to achieve this. The syntax is similar to a `display` task, but rather than being executed just once it is executed *whenever* one of the arguments 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;

  x = 30;
  y = 40;
end
```

we would expect two messages to be printed to the console: one each after the first assignments to `x` and `y`. The second assignments to `x` and `y` do not trigger anything to be printed because the `monitoroff` system task disables the mechanism (having initially been enabled by `monitoron`).

4.2.3 The `random` function

The `random` system function provides a source of (pseudo-) random numbers by returning 32 bits of random output each time it is invoked; it takes an optional argument (a "seed") that initialises this process. For example, we might set some register `x` to a random value as follows:

```
begin
  x = $random;
end
```

Instantiation	Meaning
`fa t0( w0, w1, w2, w3, w4 );`	Unnamed ports.
`fa t1( .co( w0 ), .s( w1 ), .ci( w2 ), .x( w3 ), .y( w4 ) );`	Named ports, normal order.
`fa t2( .co( w0 ), .s( w1 ), .ci( w2 ), .y( w4 ), .x( w3 ) );`	Named ports, alternate order.
`fa t3( .s( w1 ), .ci( w2 ), .x( w3 ), .y( w4 ) );`	Named ports, `co` omitted.

Figure 25: *Using named port lists to instantiate a full-adder cell.*

4.2.4 The `stop` and `finish` tasks

It can be useful for the model being simulated to control the simulation. The `finish` and `stop` system tasks do this in two similar but complementary ways:

- `stop` instructs the simulator to pause in order to accept input from the user. For example, if an error condition occurs during simulation this is useful since it allows the user to manually inspect (and potentially restart) the simulation.
- `finish` terminates the simulation (and probably the simulator itself).

4.3 Named port lists

As modules get more complex, the number of input and output ports will typically grow; this can present problems that extend beyond finding a meaningful name for each port. First, it will start to become difficult to reason about which port is which. This means connecting an external wire or register to the wrong port is more likely, and changes to the module (e.g., adding an extra port) are harder. Second, it is quite common to want to omit specific connections to a module because they are irrelevant somehow functionality they provide. For example, if we have a 1-bit, 4-way multiplexer but only use three of the four inputs then we might want to omit one of them (i.e., simply leave it unconnected).

The concept of a **named port list** can solve both of problems at the same time. Consider having to instantiate the full-adder module named `fa` and as described in Section 2. Figure 25 shows four different instanciations:

- In the first case, we adopt the standard approach of specifying connections via their order: each external wire is connected to the corresponding entry in the port list defined by the module being instanciated.

  As such, `w0` is connected to the first entry in the module port list (which is `co`), `w0` to the second entry (which is `s`), and so on.

- The second case is equivalent to the first, although the syntax now differs. The "dot" notation now specifies connections by name: each external wire (in the braces) is connected to a named entry (after the dot) in the port list.

  For example, `.co( w0 )` specifies that the external wire `w0` should be connected to the port named `co`.

- The third case demonstrates that by specifying connections by name, we can reorder the connections but still specify the same result.

  Specifically, entries for the ports named `x` and `y` are in a different order than the second case. The instanciation is still equivalent however: each external wire is still connected to the same port as before.

- The fourth case demonstrates that by specifying connections by name, we can omit one (or more) without impact on the rest.

  In this case, a connection to the `co`, or carry-out, port is omitted; you can think of this as simply ignoring the output since the result is not used. We could not do the same thing using the ordering approach however: there *must* be a first entry in our list of external wires, so this will always be connected to `co` by default.

4.4 The Verilog pre-processor

Compilation of C programs typically includes some **pre-processing**; this may be invoked implicitly by the compiler, e.g., `gcc`, rather than explicitly, e.g., as `cpp`. The idea is that directives in the source code control the pre-processor; in a sense the pre-processor "executes" the directives in order to manipulate or translate the source code before it is fed to the compiler.

Although less sophisticated than the C pre-processor, Verilog supports a similar idea through **directives**; note that unlike C where each directive is prefixed by the hash character, in Verilog a quote character is used for the same purpose.

```
`define N 8

module mux2_nbit( output wire [ `N - 1 : 0 ] r,
                  input  wire                c,
                  input  wire [ `N - 1 : 0 ] x,
                  input  wire [ `N - 1 : 0 ] y );

  assign r = c ? y : x;

endmodule
```

Figure 26: *Implementation of an* `N`*-bit,* 2*-way multiplexer where the pre-processor defines a symbol* `N`*.*

```
module mux2_1bit( output wire r,
                  input  wire c,
                  input  wire x,
                  input  wire y );

`ifdef GATES
  wire w0, w1, w2;

  not t0( w0,  c );

  and t1( w1,  x, w0 );
  and t2( w2,  y, c  );

  or  t3(  r, w1, w2 );
`else
  assign r = c ? y : x;
`endif

endmodule
```

Figure 27: *Implementation of a* 1*-bit,* 2*-way multiplexer in* either *gate-level or RTL Verilog depending on whether the symbol* `GATE` *is defined or not.*

4.4.1 The `include` directive

Perhaps the most simple example is the `include` directive which allows inclusion of one Verilog source code file in another. Usage is simple: we simple name the file which should be included, and the pre-processor injects the content at that point before feeding the overall result to the simulator. For example we might use

```
`include "constants.v"
```

to take the contents of the source code file `constants.v`, some globally defined constant values say, and inject it at this point in the current file. Note that this approach should be used with care: cyclic inclusion, which forms an infinite loop, needs to be avoided for example.

4.4.2 The `define` directive

Literal values without an obvious meaning are sometimes termed "magic": when used in some source code, it can be unclear why their specific value is chosen or what they actually *mean*. The `define` directive (partly) solves this problem by associating a symbolic name with some literal value; this works nearly the same way as in C in the sense that every use of the symbol is instead translated into the literal. The result can potentially be much easier to understand; for example the symbol `PI` is arguably more meaningful than the value 3.142.

Figure 26 presents an example: the symbol `N` is associated, in the first line, with the decimal literal `8`. Note that one can omit the literal value entirely, and simply specify that the symbol is defined: this can be useful later when considering `ifdef`.

Unlike C, where we could use the symbol simply as `N`, in Verilog each use *must* be prefixed by a quote mark character, i.e., written as `` `N ``. In this case, the symbol is used within the module interface to dictate the width of ports `r`, `x` and `y`: for example, `wire [ `N - 1 : 0] r` describes an 8-bit wire called `r` because `N` is translated into `8`. With this approach, we more or less implement an `N`-bit, 2-way multiplexer in the sense that the number of bits can be changed by simply changing the definition (rather than the module itself).

Note that there is convention in C by which pre-processor symbols are given upper-case names to avoid clashes with other variables; in Verilog it can make sense to follow the same convention, as above, so that people can understand the source code more easily.

4.4.3 The `ifdef` directive (and friends)

Conditional inclusion or exclusion of Verilog source code can be achieved using the `ifdef` and `endif` directives (and a range or relations such as `ifndef`, `else` and `elseif`). These work much like their counterparts in C, allowing sections of source code to be conditionally included for or excluded from use; a single source code description can be manipulated automatically into the exact source code used by the simulator, rather than force the developer to (un)comment sections by hand.

Figure 27 offers an example of this concept. In this case, the implementation style used for a 1-bit, 2-way multiplexer module is controlled using `GATE`: if the symbol is defined then gate-level Verilog is used, otherwise RTL Verilog is used. That is, if the symbol is defined (resp. not defined), then the pre-processor discards the bottom (resp. top) section of source code and retains the top (resp. bottom) section before feeding the result to the simulator.

4.5 The `timescale` directive

Until now, any mention of time has been carefully described in terms of abstract time units. The `timescale` directive is a means of specifying the concrete units of time used during simulation; this impacts on any specification of delay for example. For example,

```
`timescale 10ns / 1ns;
```

sets the **reference time unit** to 10ns, and the **precision** to 1ns. This mean that 1 time unit now has the concrete meaning 10ns, and 1ns is the smallest specifiable quantity of time to which any smaller are rounded. Thus, after the directive above, the previous example

```
begin
  #10 STATEMENT0;
  #20 STATEMENT1;
  #30 STATEMENT2;
end
```

means

- STATEMENT_0 executes $10 \cdot 10 = 100$ns after the start of the block,
- STATEMENT_1 executes $20 \cdot 10 = 200$ns after STATEMENT_0 finishes,
- STATEMENT_2 executes $30 \cdot 10 = 300$ns after STATEMENT_1 finishes.

4.6 Module parameters

Although pre-processor directives are sufficient in many situations, in others they are not powerful enough to achieve the desired result. For example, reconsider Figure 26 where we controlled the size of inputs to 2-input multiplexer using the symbol `N`: by redefining `N` via the pre-processor, we could resize the inputs without altering the module definition itself. Although this is convenient up to a point, the disadvantage is that the definition is global: we cannot have one multiplexer with 4-bit inputs and one with 8-bit inputs because there is only one `N`, i.e., *all* multiplexers of this type have the same sized inputs.

The concept of module **parameters** solves this problem by allowing the declaration of a local parameter which can be redefined independently for each instance. Inside the module definition, the `parameter` keyword is used to declare a parameter with a default value; the parameter symbol can then be used freely within the module itself (although clearly it cannot be assigned to). This is highlighted by Figure 28, which defines a module called `mux2_nbit` using a parameters called `N`. Notice that the "inside" module definition style is used here so that the value of `N` can be used to control the size of `r`, `x` and `y`.

The major difference between using a parameter is that each instance can set it to a different value. This is again shown in Figure 28 where two modules called `mux2_4bit` and `mux2_8bit` are defined: each case instantiates `mux2_nbit`, naming the instance `t`, then sets the parameter `N` within `t` using the `defparam` keyword. In the former case `N` is set to four meaning that `r`, `x` and `y` have a 4-bit size; in the latter case `N` is set to eight meaning that `r`, `x` and `y` have a 8-bit size.

4.7 Generate statements

Consider (yet again) the task of writing a 4-bit, 2-way multiplexer. Having increased the size of `r`, `x` and `y` (either directly or using a parameter as above), the next task is to implement the required module behaviour. In this case, one approach would be to use RTL Verilog as in Figure 11b. Another approach would be to simply use gate-level Verilog and instantiate four instances of a 1-bit, 2-way multiplexer in a similar way to Figure 8b.

```
module mux2_nbit( r, c, x, y );

  parameter N = 1;

  output wire [ N - 1 : 0 ] r;
  input  wire               c;
  input  wire [ N - 1 : 0 ] x;
  input  wire [ N - 1 : 0 ] y;

  assign r = c ? y : x;

endmodule

module mux2_4bit( output wire [ 3 : 0 ] r,
                  input  wire           c,
                  input  wire [ 3 : 0 ] x,
                  input  wire [ 3 : 0 ] y );

  mux2_nbit t( r, c, x, y );

  defparam t.N = 4;

endmodule

module mux2_8bit( output wire [ 7 : 0 ] r,
                  input  wire           c,
                  input  wire [ 7 : 0 ] x,
                  input  wire [ 7 : 0 ] y );

  mux2_nbit t( r, c, x, y );

  defparam t.N = 8;

endmodule
```

Figure 28: *Implementation of an* N*-bit,* 2*-way multiplexer using module parameters, and instanciation as* 4*-bit,* 2*-way and* 8*-bit,* 2*-way variants.*

```
module mux2_4bit( output wire [ 3 : 0 ] r,
                  input  wire           c,
                  input  wire [ 3 : 0 ] x,
                  input  wire [ 3 : 0 ] y );
  genvar i;

  generate
    for( i = 0; i < 4; i = i + 1 ) begin:id
      mux2_1bit t( r[ i ], c, x[ i ], y[ i ] );
    end
  endgenerate

endmodule
```

Figure 29: *Implementation of an* 8*-bit,* 2*-way multiplexer using a generate statement to instanciate four* 1*-bit,* 2*-way multiplexer instances.*

Of course the clear disadvantage is that as the size of inputs and output grows, this becomes increasingly laborious and prone to mistakes.

A Verilog **generate** statement is designed to automate this sort of task, programatically generating hardware instances for is. In a conceptually similar way to the pre-processor, the idea is that the generate statement is "executed" before the model is simulated: it expands into the components which are simulated, rather than implying components itself.

Here, the idea is that instead of writing four instantiations of the 1-bit, 2-way multiplexer by hand we let the Verilog tool-chain do the work for us. Figure 29 provides an example where the `generate` and `endgenerate` keywords enclose a `for` statement. The tool-chain executes the `for` statement *before* simulation: the loop body is replicated with `i`, the loop counter defined using the `genvar` keyword, replaced by the associated value for each iteration. So since `i` ranges from zero through to three, we get four copies of

```
mux2_1bit t( r[ i ], c, x[ i ], y[ i ] );
```

where `i` is replaced by zero through to three. The end result is that we get what we want: four instances of the 1-bit, 2-way multiplexer where the `i`-th instance takes the `i`-th bit of `x` and `y` as input and produces the `i`-th bit of `r` as output. Put another way, this generate statement is equivalent to writing

```
mux2_1bit t( r[ 0 ], c, x[ 0 ], y[ 0 ] );
mux2_1bit t( r[ 1 ], c, x[ 1 ], y[ 1 ] );
mux2_1bit t( r[ 2 ], c, x[ 2 ], y[ 2 ] );
mux2_1bit t( r[ 3 ], c, x[ 3 ], y[ 3 ] );
```

One caveat to this equivalence is that in the above, we have named all the instances `t`: this would not be allowed were we to write out the instantiations by hand, but the generate statement *does* allow it by automatically constructing unique identifiers for each instance. More specifically, if as here the generate loop is labelled `id`, the instances are identified by `id[ 0 ].t`, `id[ 1 ].t` and so on. That is, the instances generated can be indexed, which is useful when constructing more complicated generated structure.

This is perhaps better explained by example: consider the 1-bit, 2-way multiplexer example again. In isolation, matching the fragment above, one way to utilise the generate loop is as follows:

```
wire           c

wire [ 3 : 0 ] r;
wire [ 3 : 0 ] x;
wire [ 3 : 0 ] y;

genvar j;

generate
  for( int j = 0; j < 4; j = j + 1 ) begin:g0
    mux2_1bit t( r[ j ], c, x[ j ], y[ j ] );
  end
endgenerate
```

That is, we first define connecting wire vectors and use `j` to index individual wires. An alternative would be as follows:

```
wire c;

genvar i, j;

generate
  for( int i = 0; i < 4; i = i + 1 ) begin:g0
    wire r, x, y;
  end
endgenerate

generate
  for( int j = 0; j < 4; j = j + 1 ) begin:g1
    mux2_1bit t( g0[ j ].r, c, g0[ j ].x, g0[ j ].y );
  end
endgenerate
```

This time, we use an initial generate loop to define wires `r`, `x` and `y`; the difference is that each definition has a unique identifier `g0[ 0 ].x`, `g0[ 1 ].x` and so on. This means in a second generate loop we can index the connecting wires via this identifier using `j`, rather than as an index into what was previously a wire vector. Of course the utility of one approach over the other depends on the context, but the key point is that either way, we are simply statically indexing named wires and hence specifying connections between components: once the pre-processed output is fed to the simulator, there is no fundamental difference between either of these approaches *or* the hand-written alternative.

4.8 Developing test stimuli

None of the Verilog modules presented thus far have been self-contained: they all include inputs and outputs, with the implication that they will be used by some external component. In a sense, any useful module will

```
module fa_test();

  wire t_co,      t_s;
  reg  t_ci; t_x, t_y;

  fa t( .co( t_co ), .s( t_s ), .ci( t_ci ), .x( t_x ), .y( t_y ) );

  initial begin
    #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b0;
    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
    #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b1;
    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b0;
    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );
    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b1;
    #10 $display( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );

    #10 $finish;
  end

endmodule
```

(a) *A test stimulus for* `fa` *that uses the* `display` *system task to print results explicitly.*

```
module fa_test();

  wire t_co,      t_s;
  reg  t_ci; t_x, t_y;

  fa t( .co( t_co ), .s( t_s ), .ci( t_ci ), .x( t_x ), .y( t_y ) );

  initial begin
        $monitor( "co=%b s=%b ci=%b x=%b y=%b", t_co, t_s, t_ci, t_x, t_y );

        $monitoron;

    #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b0;
    #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b1;
    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b0;
    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b1;

    #10 $monitoroff;
        $finish;
  end

endmodule
```

(b) *A test stimulus for* `fa` *that uses the* `monitor` *system task to print results implicitly.*

Figure 30: *Two styles of test stimulus for a full-adder cell.*

```
module fa_test();

  wire t_co,      t_s;
  reg  t_ci; t_x, t_y;

  fa t( .co( t_co ), .s( t_s ), .ci( t_ci ), .x( t_x ), .y( t_y ) );

  wire o_co,      o_s;

  assign { o_co, o_s } = t_ci + t_x + t_y;

  wire f = ( o_co == t_co ) &
           ( o_s  == t_s  ) ;

  initial begin
        $monitor( "f=%s ci=%b x=%b y=%b", f ? "pass" : "fail", t_ci, t_x, t_y );

        $monitoron;

    #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b0;
    #10 t_ci = 1'b0; t_x = 1'b0; t_y = 1'b1;
    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b0;
    #10 t_ci = 1'b0; t_x = 1'b1; t_y = 1'b1;

    #10 $monitoroff;
        $finish;
  end

endmodule
```

Figure 31: *A test stimulus for* `fa` *with integrated oracle (using the built-in Verilog plus operator).*

```
module traffic_test();

  reg  t_clk;
  reg  t_rst;

  wire t_Mr, t_Ma, t_Mg;
  wire t_Ar, t_Aa, t_Ag;

  traffic t( .clk( t_clk ), .rst( t_rst ), .Mr( t_Mr ), .Ar( t_Ar ),
                                           .Ma( t_Ma ), .Aa( t_Aa ),
                                           .Mg( t_Mg ), .Ag( t_Ag ) );

  initial begin
       t_clk =   1'b0;
  end

  always  begin
    #1 t_clk = ~t_clk;
  end

endmodule
```

Figure 32: *A (partial) test stimulus for the traffic light FSM (which requires a clock signal)*

look like this: if it has no inputs and produces no output as a side effect, we can replace it with an empty module and get the same result!

However modules without inputs or outputs, which are often termed **top level modules** in reference to their location in a hierarchy of instances, *are* useful. Specifically, in order to describe a test stimulus as outlined in Chapter **??** we need to write a module which acts as a container for an instance of whatever we are testing. In short, the role of the stimulus module is as a surrogate for the external component mentioned above: it provides inputs to the DUT, and inspects the outputs to check they are as expected (e.g., match an oracle). Note that the remit of the stimulus is usually limited to testing our Verilog model; we typically do not manufacture it for example, so there is usually some flexibility in terms of how we implement it. For example even if we use a very detailed, gate-level description of the DUT, it usually makes sense to use a high-level approach to describing the stimulus.

4.8.1 Providing input to and inspecting output from a DUT

Figure 30 includes two approaches to writing a Verilog test stimulus for a full-adder. Both take a similar approach in the sense that they comprise roughly four parts:

1. definition of a register (or vector thereof) for each input to the DUT,
2. definition of a wire (or vector thereof) for each output from the DUT,

3. an instantiation of the DUT, connecting the input and output ports to the registers and wires above, and
4. a set of behavioural processes that stimulate the inputs to the DUT and inspect the outputs.

In Figure 30a for example, we define

1. registers `t_ci`, `t_x` and `t_y`,
2. wires `t_co` and `t_s`
3. define an instance of `fa` called `t` whose inputs `ci`, `x` and `y` and outputs `co` and `s` are connected to `t_ci`, `t_x` and `t_y` and `t_co` and `t_s` respectively, and
4. an `initial` process comprised of procedural assignments to registers `t_ci`, `t_x` and `t_y`, and invocations of the `display` system task to inspect wires `t_co` and `t_s`.

Of course this is a reasonable starting point (for combinatorial circuits at least), but one can easily point at ways to improve it. For example, Figure 30b resolves the need for explicit calls to `display` by using the `monitor` system task: now, every time one of `t_co`, `t_s`, `t_ci`, `t_x` or `t_y` changes their values are displayed.

4.8.2 Using an oracle to automatically check for correct DUT behaviour

Another way to improve Figure 30a is to consider the use of an oracle as introduced in Chapter **??**. The idea is to include a mechanism that automatically checks whether the output from a given test is correct or incorrect rather than forcing the user do this by inspection (and potentially make mistakes). In the case of our full-adder module, this is reasonably straight-forward: we can use the built-in Verilog addition operator as an oracle.

Figure 31 demonstrates on way to achieve this. Notice that `o_co` and `o_s` are driven with the result of adding `t_ci`, `t_x` and `t_y` using the built-in Verilog plus operator. These are compared with `t_co` and `t_s` (the outputs from the full-adder instance), to form `f`; the value of this wire therefore indicates whether the full-adder is producing the output we expect for each test, since

$$\texttt{f} = \begin{cases} 1 & \text{if } \texttt{t\_co} = \texttt{o\_co} \text{ and } \texttt{t\_s} = \texttt{o\_s}, \text{ i.e., the test passed} \\ 0 & \text{if } \texttt{t\_co} \neq \texttt{o\_co} \text{ or } \quad \texttt{t\_s} \neq \texttt{o\_s}, \text{ i.e., the test failed} \end{cases}$$

As such, the `monitor` output now highlights whether or not a given test passed or failed, rather than the outputs `t_co` and `t_s` (although clearly these could be included as well, for example to assist debugging if a given test fails).

4.8.3 Generating a clock signal for use by a DUT

As with all other inputs, the stimulus must provide a clock signal for any module that requires one. This can be achieved automatically by some simulators by "marking" a wire as a clock signal somehow: the simulator then manages generation the clock signal with a given period. However, it is reasonably simply to generate one using vanilla Verilog; one approach is demonstrated in Figure 32. The idea is for the stimulus to maintain a register called `t_clk`: this is set to zero by the `initial` process. The (untriggered) `always` process loops, updating `t_clk`. To avoid timing issues with such untriggered processes as outlined earlier, the procedural assignment that updates `t_clk` includes a delay. This essentially means, in this case, `t_clk` is update (by inverting the value) every 10 time units: it toggles from 0 to 1 and back to 0 with a total clock period of 20 time units.

References

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

[2] M.S. Malone. *Infinite Loop: How Apple, the World's Most Insanely Great Computer Company, Went Insane*. 1999 (see p. 3).

[3] S. Palnitkar. *Verilog HDL: A Guide in Digital Design and Synthesis*. 2nd ed. Prentice-Hall, 2003.