# Chapter 2
# Simplifying RTL Design

*Confusion and clutter are the failure of design, not the attributes of information.*

—Edward R. Tufte

This chapter gives an overview of the challenges in RTL designs, and some of the basic techniques we can use to simplify them.

## Challenges

The basic challenge in RTL design is that there are a lot of things going on at the same time. The design of hardware involves dealing with concurrency. And currency is inherently a difficult problem.

In addition, in RTL we describe both the function of the design and a great deal of the implementation details. For instance, we define the basic clocking structure and whether reset is synchronous or asynchronous. By the way we write the RTL we determine whether latches or flip-flops will be used.

Historically, we have used code structure and coding style to develop code that is synthesis friendly, easy to achieve timing closure, and meets our power and gate count constraints. Clarity of the code has often been a secondary concern.

As designs become more complex, the challenge of describing both function and implementation at the same time becomes even more difficult. For instance, interface protocols such as USB 3.0 involve a number of complex algorithms. Although we think about these algorithms as operating on packets, these are serial interfaces; we must implement the algorithms serially, operating on one bit or one word at a time. Developing the correct algorithm and at the same time defining its serial



M. Keating, *The Simple Art of SoC Design: Closing the Gap between RTL and ESL*,
DOI 10.1007/978-1-4419-8586-6_2, © Synopsys, Inc. 2011

implementation is a complex task. As in any complex task, at some point it becomes easier to divide it into two separate tasks, and solve them separately.

One of the byproducts of designing both the function and the implementation details simultaneously is that the code size tends to become quite large. Source code file sizes can often run into the tens of pages. The code tends to be structured to be friendly to the compilers not necessarily to the humans who read and debug the code. All this results in code that is difficult to analyze, review, and debug.

## Syntactic Fluff

Another byproduct of trying to write synthesis friendly code is that we end up with a lot of syntactic fluff. For example, describing a simple flop might consist of the following code:

```
always @(posedge clk or negedge reset) begin
  if (!reset) foo <= 0;
  else foo <= foo + 1;
end
```

In this case, the only part of the code that is algorithmically significant is the line:

```
  foo <= foo + 1;
```

The rest of the code is syntactic fluff. That is, it is required in order to convince the synthesis tool that a flip-flop should be used and tell it the nature of the clock and the reset signal as well as the reset value of *foo* (which is zero for most flops).

Another example of writing synthesis friendly code is the practice of separating the code into combinational and sequential sections. In the early days of synthesis, we could get better results by putting all the combinational code at the beginning of the file and all the sequential code at the end of the file. So code might look something like the following:

```
assign a = b;

always @(c or d) begin
  e = c && d;
  f = c || d;
```

(continued)

(continued)

```
end
always @(posedge clk or negedge resetn) begin
  if (!resetn) foo <= 0;
  else foo <= a;
end

always @(posedge clk or negedge resetn) begin
  if (!resetn) bar <= 0;
  else bar <= e + f;
end
```

This structure, of course, makes no logical sense. Logically, the combinational code that defines the value of *a* should be right next to the sequential code where *a* is used.

With today's synthesis tools, this kind of partitioning provides no value at all. The synthesis tools can optimize all the code across a very large module regardless of how the code is organized or structured.

One of the themes of this book is that we need to migrate our coding style from being synthesis friendly to being human friendly. The synthesis tools have become much more sophisticated over the last 10 years, but at the same time the designs have become much more complex. As a result, we have an opportunity to rethink how we code digital designs make them easier to understand and analyze. The power of modern synthesis tools gives us a lot of leeway to modify how we write code in order to make the design process faster and more robust.

## Concurrency and State Space

There are several problems in RTL design that are simply the result of how hardware description languages and synthesis tools evolved. This category includes syntactic fluff and the fact that we describe function and implementation in the same file.

But there are two major challenges in RTL design that are fundamental to the problem of digital design: concurrency and state space. These two issues are closely related.

When we design a digital system, we are really specifying how that system evolves over time. That is, we are specifying the state space of the system and how it changes over time. The problem is that the state space may be very complex, consisting of multiple subsystems that are evolving simultaneously.

Consider, for example, a cell phone. The main digital chip in a cell phone may be simultaneously controlling the user interface, the audio and video services, network access, and the radio subsystem.

We can demonstrate the challenge of such complex systems from a very simple example. Consider the state machine in Figure 2-1.

*Note: In this book, we use a mix of styles in state machine diagrams. For very simple diagrams, we use traditional bubble diagrams. For state machine drawings where we show some code, we use State Chart notation. This format (using rectangles instead of circles for states) gives room for including more information about the state. For an explanation of this format, see [11].*

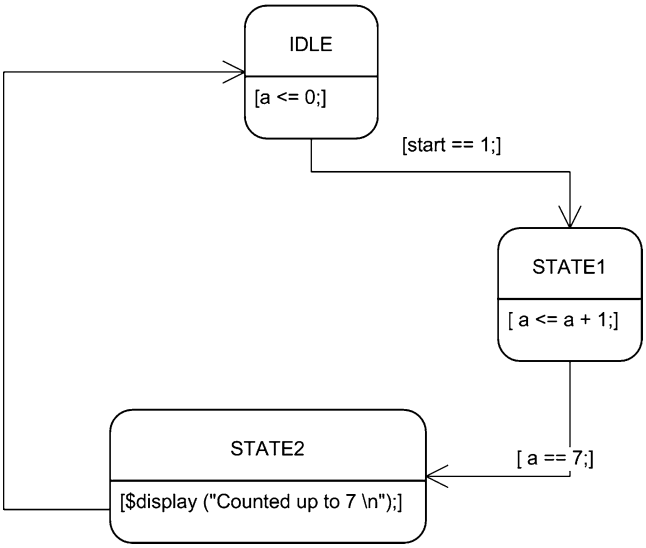


**Figure 2-1** A simple state machine.

Analyzing the state machine is quite simple. We just have a counter that counts up to seven once the start signal is asserted.

If we have two state machines that are decoupled, as in Figure 2-2, the analysis is again simple:

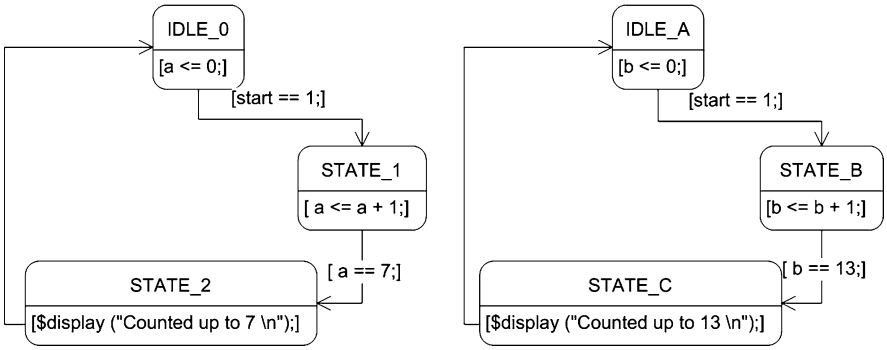


**Figure 2-2** Two decoupled state machines.

Now we have two state machines that count up to some terminal value, starting when the start signal is asserted. Note that because the two terminal counts are relatively prime, there is no way to predict the value of *b* given the value of *a*. After a hundred clock cycles or so, the relationships between the values of *a*and *b* will appear to be completely random. Thus, while it is easy to analyze each state machine independently, analyzing and predicting the values of both states at any particular time starts to get a bit tricky.

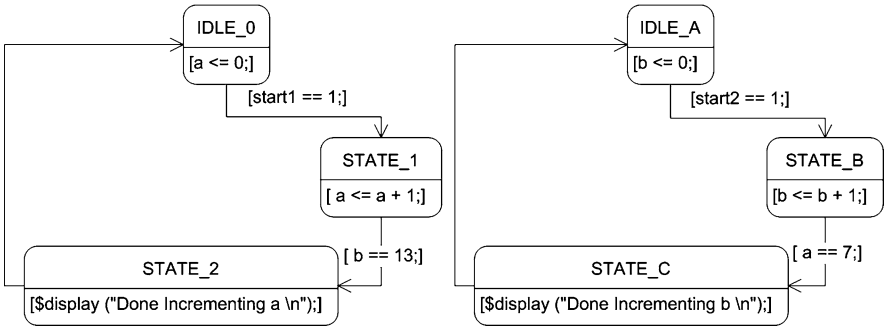


**Figure 2-3** Two coupled state machines.

In Figure 2-3 things are getting dicey. In the above design, the two counters have separate start signals. Also, we halt incrementing *a* based on the value of *b*, and vice versa. The two state machines are now tightly coupled, and the combined behavior depends heavily on when the two start signals are asserted. The behavior of this circuit is a lot more complex than the behavior of the previous two circuits.

As we can see, the concurrent behavior of two tightly coupled state machines can become very complex to analyze, even when each state machine is simple.

## Techniques

The previous sections described three problems in RTL design:

- Syntactic fluff
- The order/structure of RTL code
- The problems of state space size and complexity, and the problem of concurrency

We now give a brief overview of some of the techniques we can use to address these problems. These techniques will be explored in more detail in the rest of the book.

As mentioned earlier, the key technique for managing complexity is to divide and conquer. In terms of RTL design, and in fact in any code based design, the key mechanism is encapsulation. We want to partition the design – and the code – so that each piece can be designed and analyzed separately from the other pieces. To the degree possible, we would like to encapsulate functionality, hide local information so that external pieces of the design don't see it, and present a simple interface to the rest of the system.

Even with today's languages and tools, we can use encapsulation techniques to raise the level of abstraction above the traditional RTL level. In doing so, we can make the function of the design more obvious and make the implementation less obtrusive.

In this section, we will examine four areas for encapsulation and raising the abstraction level of design:

- Combinational code
- Sequential code
- Interfaces
- Data Types

## *Encapsulating Combinational Code*

Consider the following piece of SystemVerilog code:

```
input bit a;
input bit b;
input bit control;

bit temp;
bit [7:0] foo;

always_comb begin
  if (control == 1) temp = a;
  else temp = b;
end

always_comb foo = temp * 3;
```

In this case, the signal *temp* has global scope. That means that when we are analyzing this design, we need to worry about the value of *temp* at all times. But in fact, the signal is used only as a temporary or intermediate value in calculating *foo*.

Compare the previous counter to the following code:

```
function automatic bit [7:0] foo (input bit a, b,
control);
  bit temp;
  if (control == 1) temp = a;
  else temp = b;
  foo = temp * 3;
endfunction
```

This code is slightly shorter than the previous code. But it also has several additional advantages:

1. It makes it completely explicit that the value of *foo* depends only on the inputs *a*, *b* and *control*. This relationship is not at all obvious from the statement *always_comb foo = temp * 3.* In fact, if the two *always_comb* blocks in the previous example are separated by significant amounts of code, it may not be easy at all to see the relationship between *foo* and the inputs *a, b*, and *control***.**
2. The signal *temp* is local within the function. It is completely obvious that it is not used by any other piece of code.
3. All of the code required to calculate *foo* is grouped together within the function. There is no possibility of scattering this code throughout the file. This means that the analysis of how *foo* is calculated becomes a local rather than a global activity.
4. The function *foo* must now be called explicitly whenever it is needed. This makes coding slightly more burdensome, but it makes analysis significantly easier. Typically, the function will be called in one or perhaps a few states. That means whenever the module is in the other states, we can completely ignore *foo*.

Thus, functions provide an effective encapsulation mechanism for combinational code.

## *Structuring Sequential Code*

Unfortunately, modern hardware description languages do not provide an equivalent encapsulation mechanism for sequential code. There is no structure that allows us to group pieces of sequential code together, define explicitly the inputs, or to hide local or temporary signals. The *task* construct allows some degree of encapsulation, since (unlike *function*) it allows some timing and sequential constructs. And we will use it in a later chapter. But we are not allowed to have an *always @ (posedge clk)* block in a task. As a result, we really do not have an equivalent to the *function* for sequential code.

Instead, we are left to group sequential code arbitrarily within *always @ (posedge clk)* blocks. These sequential blocks can be scattered throughout a file. To analyze the module then, it is necessary to read and memorize virtually the entire file

Consider the following code:

```
always @ (posedge clk or negedge resetn) begin
  if (!resetn) begin
    bar <= 0;
    bar_p1 <= 0;
  end else begin
    bar_p1 <= bar;
    bar <= a + b;
  end
end

always @ (posedge clk or negedge resetn) begin
  if (!resetn) begin
    foo <= 0;
  end else begin
    foo <= bar_p1 + bar;
  end
end
```

Here it is not obvious that *foo* depends on the inputs *a* and *b*. If the two sequential blocks are separated by significant amount of code, it may be nontrivial to sort out exactly what the relationship is between *foo* and *bar*.

One possible solution is to start grouping more and more sequential code into a single sequential process. The trouble with this solution is that this process becomes large and unwieldy.

The best mechanism for structuring sequential code is the state machine. In a state machine, we can create a single large sequential process that uses the case statement to structure the sequential code into separate states.

To address the problems of concurrency described earlier, we recommend using a single state machine per module. Effective decoupling of modules (described in Chapter 8) then helps manage concurrency between state machines.

The key challenge in grouping large amounts of sequential code into a single state machine is that this state machine can rapidly become large and unwieldy itself. In fact, we can easily violate the rule of seven: many interesting state machines have more than seven to nine states. The solution to this problem is to code the process as a hierarchical state machine. We discuss hierarchical state machines Chapter 4, and give an example in Appendix B.

### *Using High Level Data Types*

Functions and state machines are the two most important mechanisms for encapsulation in RTL design. But there are some additional techniques available in SystemVerilog that can be very helpful in raising the abstraction level of RTL design.

Enumerated types are helpful in defining exactly what values are legal for a given signal or collection of signals. For instance:

```
bit read;
bit write;
```

This code implies that there are four possible values for the combination of the read and write signals. Most importantly, it implies that it is possible to assert both read and write at the same time; at least nothing in the declaration implies that this is impossible.

Instead, we can define an enumerated type signal *rw* which makes it explicit that only one of the read or write operations can be active at one time:

```
enum (NOP, READ, WRITE) rw;
```

*Structs* in SystemVerilog are also very useful in providing an encapsulation mechanism for related signals. For instance:

```
bit [ADDR_WIDTH] foo_address;
bit [ADDR_WIDTH] bar_address;

enum (NOP, READ, WRITE) foo_rw, bar_rw;

bit [DATA_WIDTH] foo_data;
bit [DATA_WIDTH] bar_data;
```

As written, the code relies on the signal name to imply the relationship between the different signals.

```
typedef struct {
  bit [ADDR_WIDTH] address;
  bit [DATA_WIDTH] data;
rw_type rw;} my_data_type;

my_data_type foo, bar;
```

Using a *struct* data type, we can make it explicit that both foo and bar are exactly the same data type, with exactly the same type of address, data and control signals. The relationship between the address, data, and control signals is much more explicit as well.

The SystemVerilog *interface* construct provides an encapsulation mechanism at the interface level. A module definition with 30 or 40 inputs and outputs clearly violates the rule of seven. Using the interface construct, we can reduce this to seven to nine interface declarations.

The following is an example of how a simple memory interface can be defined using interfaces:

```
interface mem_intf ; // interface for i_mem and d_mem
  bit [ADDR_WIDTH-1:0] addr;
  bit [WORD_SIZE-1:0] write_data;
  bit [WORD_SIZE-1:0] read_data;
  bit read;
  bit write;

  modport master (output addr, write_data, read, write,
                  input read_data);
  modport slave (input addr, write_data, read, write,
                 output read_data, exc );

endinterface: mem_intf
```

Then in the top level module, we instantiate an interface and connect it to the memory. Note how simple the code for the instantiating the memory has become, since only the interface, and not five different ports, needs to be connected.

```
module top ;
    …
mem_intf d_mem_intf();
    …
mem d_mem (.ifc(d_mem_intf), .clk(clk));
    …
endmodule
```

Then our behavioral model for the memory might look something like this. Note how simple the port declaration has become, since we declare the interface instead of five different ports.

```
module mem (input bit clk, mem_intf ifc);

  bit [`WORD_SIZE-1:0] mem_array [`MEM_DEPTH-1:0] ;

  always @(posedge clk) begin
    if (ifc.read) ifc.read_data <= mem_array[ifc.addr];
    if (ifc.write)mem_array[ifc.addr] <= ifc.write_data;
  end
endmodule
```

For an extensive discussion of how to use the *interface* construct, see [8]. For a brief discussion of how extensions to the synthesizable subset of SystemVerilog could make the interface construct even more useful, see the first section of Appendix D.

Finally, even the *for* loop now has a small opportunity for encapsulation:

```
    for(int index = 0; index < max_val; index++)
```

By declaring the loop index inside the for loop, we hide it from the rest of the code.

## *Thinking High-level*

Most important of all, raising the level of abstraction of RTL code requires us to think high-level in every aspect of coding. For example, consider the following piece of code:

```
if (foo == 1'b1)
```

This is an example of thinking at the bit level. We are asking if the value of *foo* is equal to one, which we associate with a Boolean value true.

The following piece of code is functionally the same as before, but simpler and at a higher level of abstraction:

```
if (foo)
```

In this statement, we simply ask if foo is true. In fact, we know that this is equivalent to asking if foo is not equal to zero.

There are several (admittedly small) problems with the first approach.

1. It is more verbose than necessary, which can become a significant issue when reading large amounts of code.
2. It inserts an implementation issue (the fact that we are using a value of one represent a Boolean value true), when we are really interested in the functional or algorithmic aspects of the design.

Both ways of writing an *if* statement are perfectly legal, and both will produce exactly the same synthesis results, that is, the same gate level netlist. But the second version is more compact and more functional rather than structural.

All the techniques described in this chapter strive to achieve a single goal. There are many different ways of writing the same logic in RTL code. In the past, we had to choose the coding style that lead the synthesis tools to produce the optimum result. But today, with the explosion of complexity in design, we need to use a coding methodology that makes the code easy to understand, to review, to analyze and to debug.

http://www.springer.com/978-1-4419-8585-9

The Simple Art of SoC Design
Closing the Gap between RTL and ESL
Keating, Synopsys Fellow, M.
2011, XVII, 234 p., Hardcover
ISBN: 978-1-4419-8585-9