6.1 Introduction

Verilog is a powerful and flexible language. In addition, it has many similarities with C. As a result, novice designers have a great tendency to use Verilog to capture behavioral intent and then try to make that intent look like synthesizable code. The results tend to be non-functional or inefficient designs that take a long time to debug. A far better approach is to spend effort to first determine the hardware structure and then capture that structure in code. The main purpose of this chapter is to teach the novice designer how to use and be successful with this approach.

With respect to sequential machines the specification style of this chapter is sometimes called \textit{explicit}, to distinguish it from the \textit{implicit} styles of chapter 8 and following. Although the explicit style is often recommended for beginning designers, both style families have their place, and the eventual style adopted by designer will be both a matter of preference as well as depending on the characteristics of a particular design project.
6.2 Basic Design Methodology

Modern designs tend to be very complex, consisting of thousands to millions of gates. Yet, in general, the first prototype of a silicon chip either works completely or has only a handful of bugs. First pass success is highly desirable as repairing and reimplementing a design is very time consuming, and time-to-market is a key factor in the eventual profitability of a new system. If the part being built is a CMOS chip, the fabrication facility will take eight weeks or more to return the chip after design submission. Even though the reimplementation delay is not as long if the design is implemented with an FPGA, it is a lot easier to debug a large design at the Verilog stage than at the hardware stage.

For this reason, good design teams enforce a disciplined approach to their design. This is usually referred to as a methodology – a method or process to be used during design. This chapter is going to emphasize a simple methodology to be used for module level design. This process is summarized in Figure 6.1 and described in detail below:

1) **Determine or obtain the Specification.** The specification details the behavior and interface of each module being designed. This stage is discussed more in Section 6.3.

2) **Structure the design to the register transfer level.** This is the ‘logic design phase’ when the designer determines a ‘block diagram’ for the design, including registers, functions of combinational logic, etc. A set of techniques for tackling this stage is the subject of Section 6.4

3) **Capture the design as Verilog.** If the structuring step is done well, then this step is very straightforward, as described in Section 6.5.

---

**FIGURE 6-1: Steps in the Design of Small Modules.**
4) **Verify the design.** A critical element in obtaining a working prototype is spending a lot of time verifying the design via Verilog simulation. In today’s chips, sometimes over half of the project resources are spent on verification. This topic is important enough to rate its own chapter, Chapter 7.

5) **Synthesize the Design.** The synthesis tools are used to transform the Verilog design into a gate level design. Synthesis is not just a black box, the designer has to be aware of what synthesis tools can and cannot do well. The partitioning of the design into Verilog modules also has a big impact on synthesis. These issues are discussed more in Chapters 8 through 11.

6) **Verify the results of synthesis.** Every tool has bugs and thus the output has to be checked. A combination of gate-level simulation, timing analysis, and other techniques, are used to verify that the design produced by the synthesis tool is correct and consistent with the Verilog RTL design.

7) **Place and Route.** This stage is often referred to as “physical design” because it is where the actual layout of the chip is determined. The gates in the chip are assigned to positions on the chip (placement) and then connected together with wires (routing).

8) **Final Verification.** Before sending the chip to the fabrication facility, or burning the FPGA, a number of final checks are done to make sure that the chip is wired up correctly, is manufacturable, etc. The nature of these checks are beyond the scope of this book.

### 6.3 The Specification

The specification details the behavior and interface of each module in the design. It is a very important step. Many chips fail in the marketplace because the feature set was not well thought through. In addition, a surprisingly large number of chips pass stand-alone functional test but do not work when inserted in the system board because the chip interfaces were incorrectly or incompletely specified. The specification covers multiple levels of the design, all the way from the chip level functionality down to module level timing. In order to speed up time-to-market, detailed design often starts while the specification is still being formulated. Thus the module designer has to cope with a changing specification and also often plays a role in formulating its details.

At the module level, the specification should include the following:

- A description of the top-level behavior of the module,
- A description of all its inputs and outputs, their timing and constraints.
• Performance requirements and constraints.

Ideally, the top-level behavior of the chip and each module should be captured in a high level language, such as C++, Java, Matlab, or Specification and Description Language (SDL - often used in communications system design). Such a high level model is useful for a number of reasons. First, by being able to simulate and verify the behavioral specification, fewer bugs will be introduced at this level. Second, the results of these simulations can later be checked against the results of the Verilog RTL simulations. This comparison will give the module designer greater assurance that the design is correct. Finally, this high level module could be used in place of the Verilog RTL module when simulating the entire design at the chip level. Since the high level language description simulates a lot faster than a Verilog description of the same module, final chip-level simulation can be speeded up by simulating only critical portions in Verilog, and the rest in a high level language. Verilog’s Programming Language Interface (PLI) is a useful mechanism for performing this type of co-simulation (see chapter 17).

Note that the structure of the high level language description of the module does not have to relate to the internal hardware about to be designed. However, also note that one cannot turn C into Verilog simply by putting an `always @(posedge clock)` in front of blocks of C code!

6.4 Structuring the Design

This is a very important step that neophyte designers are tempted to skip or combine with the actual coding. When this step is given short shrift, the resulting design is often buggy and inefficient in performance and/or area. Often the time saved in avoiding effort in the structuring step is more than compensated for by the time spent debugging or improving the design. The sub-steps are as follows:

1) **Determine the control strategy.** An important rule to follow is:

   *Clearly separate Control from Datapath.*

The elements of the design that operate on the data (the **datapath**) must be clearly separated from the portion of the design that determines the sequence of events in the datapath (the **controller**). The datapath and controller are joined by clearly identified **control signals** and **status signals**. Together, these control and status signals are referred to as **control points** in the following chapters. Typically the controller consists of some combination of Finite State Machines and counters (or possibly a microcode ROM so as to avoid an overly large FSM). The datapath consists of a number of registers and combinational logic. The controller determines the sequence of events that take place on the datapath. It controls
this sequence of events via the control lines. The datapath processes the data and reports a number of status bits back to the controller.

There are a number of key elements that must be considered when determining the control strategy and sequence, including the following:

- **What is the Reset strategy?** Every chip has a global reset. Reset usually becomes active on power up and when the reset button is pushed. Typically active low, the Reset signal sets the contents of various registers to a predetermined value to put the chip into a known state. Which registers to reset will be illustrated in the examples which follow in subsequent chapters.

- **What is to be performed in each clock period?** The purpose of the controller is to determine what events are to be performed on the datapath during each clock period.

- **Pay special attention to transitions.** Transitions between different modes of operation, for example, the start and finish of sets of operations, requires special attention.

(2) **Determine the Register Transfer Level structure of the design.** At this stage, the following pieces of the design must be explicitly identified and labeled:

- **Module inputs and outputs.** Identify, name and determine the function of each input and output signal.

- **Registers and register outputs.** Identify each register and give a signal name to its output(s). Obviously, the purpose of each register must be clear in the designer’s head. Generally a register is necessary whenever a value has to be preserved across one or more clock edges. In most designs each register is clocked once every clock cycle. Usually it is dangerous to mix together registers that are clocked every clock cycle with registers that are clocked less or more frequently. There are several reasons why this practice is dangerous. First, it complicates synthesis, the designer will have to explicitly identify the multi-clock cycle registers. Second, it complicates the design. Let’s say one register that is clocked every cycle is feeding another register that is clocked every other cycle, through some combinational logic. Then there is a requirement that the first register must hold its outputs stable for two clock periods to permit the second register to have the correct inputs when it is ready to latch. Similarly, asynchronous (unclocked or self-timed) design is not supported by commercial Synthesis tools.

- **Combinational logic blocks and their functions.** Identify the blocks of logic, including arithmetic units, multiplexors, comparators, deselectors, coders, decoders, etc. Not a lot of detail is needed at this stage – you do not design these blocks down to the gate level.

During the RTL structuring phase, it is important to ensure that these control issues are properly addressed. However, above all, the following rule must always be obeyed:

> **Always design before coding.**
(3) Capture the design as Verilog. If the above steps are done correctly, then this step is straightforward, as different groupings of logic can be mapped onto groups of Verilog RTL descriptions with fairly unique mappings. The correspondence between different types of logic and their Verilog single or compound statement equivalents are illustrated using simple examples given in Figure 6.2:

- Edge-triggered flip-flops and their associated input logic are built using procedural blocks containing `always @(posedge clock)` statements as was discussed in section 5.6. Optionally, `@ (posedge clock)` can also be used. Remember, every variable assigned in a procedural block triggered by a statement of the type `always` `@ (posedge clock)` will become the output of a flip-flop when synthesized.
- Level sensitive latches (and associated input logic), if used, are built using procedural blocks headed with `always @(clock)` statements. (Latches are not illustrated in Figure 6-2 as latch usage is rare in modern design.)
- Simple combinational logic is built using continuous assignment statements.
- More complex combinational logic is built using procedural blocks that do not contain ‘clock’ in the timing sensitivity list.

In general, these are the only types of construct that are needed in order to capture a design at the Register Transfer Language level.

Note how the explicit parallelism of hardware is captured. Since each procedural block or continuous assignment statement in Verilog executes independently of each other, by mapping groups of gates onto different blocks or statements, we are modeling each group of gates independent of the other groups.

Now two designs are used to illustrate steps 1 to 4 above, with an emphasis on the structuring step.

### 6.5 Design Example 1 – A Simple Down Counter

Counters are often used to count events or clock ticks so as to cause control lines to toggle at certain pre-arranged times. Fundamentally, a counter is a simple version of a Finite State Machine (FSM), in which the next state is simply the current state plus +/- X, often +/- 1. In this section, a simple decrement (down) counter will be designed from specification through to verification.

**Specification**

For this module, the specification will be conveyed by a simple document identifying the module and specifying the behavior of the inputs and outputs. A simplified specification document is depicted in Figure 6.3. In this case, the specification only describes behavior, normally timing and other issues will also be addressed.

**Determine the Control Strategy**

In this case, there is no need to go through identifying the control strategy of this module, due to its simplicity.
Determine the RTL Structure of the Design

There are three steps in this stage, illustrated as a sequence in Figure 6.4. These steps are as follows:

1. Identify the inputs and outputs. These come directly from the specification.

2. Determine what registers are needed and name their outputs. In this case, this decision is straightforward. The current value of the count must be registered so as to preserve it across clock edges. In addition, the fact that the count value is fed back through the subtractor that produces it also dictates that a register is needed. Feedback of combinational logic is never permitted because that essentially creates a race condition. Consider what
would happen without the register (Figure 6.5). The count would cycle down to zero as fast as the logic would permit. This point is worth emphasizing and generalizing:

(3) Specify the function of the combinational logic. In this case, we need an subtractor, a multiplexor and some simple logic to produce the zero flag. Note the details are not specified, since at this stage, we are just producing a sketch.
Capture the Design As Verilog

Once the Register Transfer Level design is complete, capturing the design as Verilog is almost a straightforward translation of design to Verilog construct, as follows:

Registered outputs and their associated logic are described in procedural blocks starting with `always @(posedge clock)`. In this case, the flip-flops storing the count, the input multiplexor and the subtractor are captured in Verilog as in the first box:

```
always @(posedge clock)
begin
    if (latch)value <= in;
    else if (dec && !zero) value <= value - 1'b1;
end
```

Note, since the procedural block contains only one statement, the `begin` and `end` are not strictly necessary here. This implements the following hardware: A register with an output called ‘value’, a priority-selector (latch has priority over dec and zero since the if statement is always executed if latch=1, no matter the value of dec and zero) that connects to the data input of the flip-flops, and; a subtract-by-1 unit connected to one of the inputs to the selector. This logic could also have been implemented with a case statement as in the next box:

```
always@ (posedge clock)
casex ([latch, dec, zero])
  1xx : value <= in;
  010 : value <= value - 1'b1;
endcase
```
Simple combinational logic is best captured as a continuous assignment statement. In this case, the logic that produces the ‘zero’ flag is captured in Verilog as follows:

```
assign zero = ~|value;
```

More complex combinational logic is captured in procedural blocks with no clock in the sensitivity list. In this example, this is not necessary.

Putting it all together, including variable declarations and appropriate comments to describe the module, the final module is as follows:

```verilog
module counter (clock, in, latch, dec, zero);
input clock;/* clock */
input [3:0] in;/* input initial count */
input latch;/* latch input */
input dec;/* decrement */
output zero;/* zero flag */
reg [3:0] value;/* current count value */
wire zero;

// Count Flip-flops with input multiplexor and subtractor
always @(posedge clock)
if (latch) value <= in;
else if (dec && !zero) value <= value - 1'b1;
assign zero = ~|value;// combinational logic for zero flag
endmodule // counter
downcounter with zero flag
```

**Verify Correctness of the Design**

Generally about half of the effort in design goes into verifying correctness. A test fixture must be written to determine if the design is correct. Verification approaches will be discussed more in chapters 7 and 15. Here (in the box on the next page), is a simple test fixture to verify only the basic features in the above design. It is not a full verification of functionality.

As all of the syntax features included in this test fixture are illustrated elsewhere in this book, a detailed review of this code will not be conducted here. However, the waveforms produced by this code are shown in Figure 6.6, and the reader should note the following:

- An initial statement is used as the test fixture needs to start executing when the simulation starts.
Delay (\#) statements are used to determine when input vectors change.

Correct behavior of the module being tested is checked within the test fixture. There is no need to view the waveforms if no errors are reported.

The use of the \texttt{always} statement to invert the clock every 5 ns.

Note in Figure 6.6 that the outputs of the flip-flops do not respond to changes in their inputs until after the clock edge. This might seem obvious, but it is a common source of confusion to novice designers.

\section*{6.6 Example 2 – Unsigned Parallel-Serial Multiplier}

The goals of this example are as follows:

- To illustrate the steps involved in structuring on a more complex design
To illustrate some clock to clock timing issues
To show how the ‘control’ and ‘datapath’ are clearly separated and identified

The example design is an unsigned parallel serial multiplier. Before proceeding with the design, it is necessary to review the basic parallel serial multiplication algorithm. This review will be used in place of a detailed specification in this example. Consider the example of multiplying two 4-bit numbers:

Multiplicand: 1011 (11 in base 10)
Multiplier: 1011 (11 in base 10)

The product is formed by a series of four adds. At each stage, the multiplicand is added to the high order 4 bits of the product if the least significant bit of the multiplier is 1 or zero is added to the product if the least significant bit of the multiplier is 0. The multiplier and multiplicand are then both shifted to the right.

The first step in the example is to clear the product register:
Since the least significant bit (lsb) of the multiplier is ‘1’, the multiplicand is added to the high order four bits of the product in the first cycle to give the result:

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Product</th>
<th>New Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>0000 0000</td>
<td>01011 0000</td>
</tr>
</tbody>
</table>

Note how the carry-out of the sum is included in the add. To finish the first cycle, the multiplier and product are both shifted to the right:

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>0101 1000</td>
</tr>
</tbody>
</table>

In the second cycle, the lsb of the multiplier is 1, so the multiplicand is added again to the product:

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Product</th>
<th>New Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>0101 1000</td>
<td>10000 1000</td>
</tr>
</tbody>
</table>

The carryout is 1 this time. The multiplier and product are again both shifted to the right:

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1000 0100</td>
</tr>
</tbody>
</table>

In the third cycle, the lsb of the multiplier is 0, so zero is added to the product:

<table>
<thead>
<tr>
<th>Zero</th>
<th>Product</th>
<th>New Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1000 0100</td>
<td>01000 0100</td>
</tr>
</tbody>
</table>

The multiplier and product are again both shifted to the right:

<table>
<thead>
<tr>
<th>Multiplier</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0100 0010</td>
</tr>
</tbody>
</table>

In the final cycle, the multiplicand is added again:

<table>
<thead>
<tr>
<th>Multiplicand</th>
<th>Product</th>
<th>New Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>1011</td>
<td>0100 0010</td>
<td>01111 0010</td>
</tr>
</tbody>
</table>

The multiplier and product are both shifted to the final time and the 8-bit result is left in the product:
Multiplier: 0111 1001  (121 in base 10, as expected)

The design procedure of a multiplier that can execute this algorithm is described next.

**Determine the Control Strategy**

Since the multiplier requires nine steps (one step to load the required registers and eight steps to do the actual multiplication), a counter-based controller makes the most sense.

**Determine the RTL Structure of the Design**

The three steps to this stage are as follows (see the structure of the design sketched in Figure 6.7):

1. **Identify the inputs and outputs.** These are simply as expected. We need to input the multiplicand and multiplier under the control of a ‘load’ signal, and output the product.

2. **Determine what registers are needed and name their outputs.** Remember to clearly separate the datapath from the controller. The following need to be registered in the datapath:
   - The multiplicand and multiplier, as their values are needed over several clock cycles.
   - The product, as it is determined through steps over many clock cycles.

   However, we note that there is potential here to save some flip-flops. Note that at each stage of the multiply, both the low order bits of the product register and the high order bits of the multiplier contain 0. They are not actually being used. We can safely use the low order half of the product register to store the multiplier when loading the inputs. Then, as the algorithm progresses, the product and multiplier will shift right in unison until at the end only the product is stored in this register.

   The following registers are needed in the controller:
   - A register to store the count

3. **Specify the function of the combinational logic.** Again the details of the datapath are determined before those of the controller.

   The first thing to note is that there is no reason not to perform the add and the shift in the same clock cycle. A novice designer would most likely have been tempted to do the add and the shift in two separate clock cycles, thinking naively that the shift HAS to be done in a shift register. Instead we do the shift in a logic block, thus halving the time necessary to perform the multiplication.
Thus, the logic consists of an adder, and a combinational logic block to use the result of the add, to perform the shift and the initial load. Note that Verilog recognizes that adding two 8-bit numbers actually produces a 9-bit result, and the code can capture the 9th bit (the carry out) explicitly as follows:

\[
\{\text{next}_E, \text{AdderOut}\} = \text{multiplicand} + \text{Product}[15:8];
\]
It is necessary to explicitly identify the control inputs to the datapath. They are as follows:

- load’ from an input
- The most significant bit, bit 3 of count to tell the logic that the multiply is finished and product should be left unchanged.
- The low order bit of the multiplier (stored in the product register) to determine whether to use the result of the add to the multiplier or not.

Then it is necessary to determine the details of the controller. A novice designer might think that a 3-bit counter would suffice to count the 8 clock cycles needed to do the multiplier. However, a ninth state is needed to ‘hold’ the product output until load goes high again. Thus, a 4-bit counter is needed, with the 4th bit being the ‘finish’ control bit.

**Capture the Design As Verilog**

The Verilog code that captures this design is presented in the box below (in a parameterized form):

```
This example illustrates each of the three types of constructs that are generally used to capture synthesizable designs. In this design, there are two procedural blocks, one for the registers (including smaller input selectors) and one for the large selector and shift logic. There is also a continuous assignment statement to build the adder including carry out. The design also has the following features:

- It is parameterized. By varying the parameters N and Cycles, different sized multipliers can be built easily and quickly. Thus, the design can be easily modified and reused in later chips.
- Because of the use of non-blocking assignment (‘<=’) in the procedural block building the flip-flops, all the statements in this block execute ‘in parallel’ and thus the registers being defined are all independent of each other.
- In the non-clock procedural block, all of the logical inputs are included in the timing sensitivity list, i.e., every variable on the right hand side of an assignment statement that is not produced internally to the block has to be in this list. Otherwise when it changes, the execution of this block will not be triggered. Since, in the real hardware, the logic does re-evaluate whenever an input to the hardware changes, the Verilog model would not accurately capture the hardware without an accurate timing sensitivity list. Unfortunately, if a variable is accidentally left out of this list the Verilog simulator will not complain, though the synthesis tool will when you read in the Verilog.
- The larger input multiplexor and shifter could have been captured in the `always @(posedge clock)` statement as in the following box:
```
Design Approaches for Single Modules

Note that with the case statement included in the clocked procedural block, there is no need for a default statement. We are building memory, so accidentally building memory by not having a default statement is not a concern! Also note that since the `casex` statement uses the output of the counter, non-blocking assignment is essential for correct operation. In this design this block was coded separately mainly to illustrate the use of a procedural block to capture combinational logic only rather than for any more fundamental reason.
Design Approaches for Single Modules

6.7 An Alternative Approach to Specifying Flip-Flops

In the above code, flip-flops are built by placing the names of their outputs in a procedural block starting with `always @(posedge clock)`. An alternative approach to specifying flip-flops is to place `@(posedge clock)` triggers within procedural blocks. For example, the following Verilog fragment is equivalent to the fragment that built the flip-flops in the counter example given earlier.

```
always @(posedge clock)
begin
    ...
    casez ((Product[0], Count[Cycles], load))
    3'b0: Product <= {N'b0, multiplierIn[N-1:0]};
    3'b100: Product = {next_E, AdderOut[N-1:0], Product[N-1:1]};
    3'b000: Product = {1'b0, Product[2*N-1:1]};
endcase
    ...
end
```

```
always
begin @(posedge clock)
    if (latch) value <= in;
    else if (dec && !zero) value <= value - 1'b1;
end
```

In this case the always statement specifies a continuous loop and the `@(posedge clock)` ‘trigger’ results in the body of the procedural block only executing when there is a positive edge of the clock; thus capturing the behavior of a flip-flop.

It is also possible, though sometimes a little dangerous, to embed multiple `@(posedge clock)` statements into a block and thus specify a series of events on a set of flip-flops. For example, consider the following code fragment.

```
always
begin @(posedge clock)
    if (dec1) value <= value – 1'b1;
    @(posedge clock)
    if (dec2) value <= value – 2'b10;
end
```

This code fragment actually specifies a downcounter that includes an implicit small 1-
flip-flop finite state machine, as well as the specified ‘value’ register. On every odd phase of the clock, it decrements value by 1 if dec1 is true. On every even phase of the clock, it decrements value by 2 if dec2 is true. Though this is a perfectly legal style for purposes of both synthesis and simulation, it does carry its dangers, especially for neophyte designers. In the example above, there is specified an implicit, unnamed one-bit finite state machine now buried in the datapath. This approach thus violates two of the guidelines above: the datapath and controller are not clearly separated and we have not explicitly identified the output signals of all flip-flops. However, many designers do use this technique successfully, and since it reduces the labor of specifying the controller, this style will also be developed in succeeding chapters.

6.8 Common Problems and Fixes

There are many common patterns to the errors that neophyte designers make. This section presents some of these common problems and discusses how to avoid them.

Accidental Instantiation of Latches

As introduced in section 5.5 and 5.6, latches are inferred whenever a non-clocked procedural block intended to model combinational logic does not give a fresh assignment to each and every variable assigned to it every time it is executed. This is such a common source of problems that the topic merits additional emphasis here. One common situation when this occurs is when all possible combinations of ‘control’ variables are not covered in a case or if--else statement. For example:

```vhdl
reg A,B,C,D;
always@((A or B or C)
casez ((B,C))
  2'b1? : D=A;
  2'b?1 : D=~-A;
endcase
```

Here, if \{B,C\} changes to \{0,0\}, this procedural block will execute but neither of the assignments to D will execute, i.e. the block is implicitly specifying that D should remain unchanged whenever \{B,C\}=2'b00. This is what was called a not-full case in chapter 5. For D to remain unchanged requires that its value be stored in a memory element. As there are no edge-triggers here, that memory element will be a latch ‘clocked’ through some combined value of B and C. The fix to this problem is to make sure that every case statement not in a clocked procedural block has a default. In this case, the designer does not care what value D takes when B or C is not equal to 1, and so D is specified as a don’t care. Remember, the appropriate use of don’t cares often leads to smaller, faster logic.
Another common way latches are accidently inferred occurs when a procedural block has multiple outputs, and not every output is assigned for all possible combinations in which the block could be executed. For example in the next box:

```verilog
reg A,B,C,D;
always @(A or B or C)
casez ([B,C])
  2'b1? : D=A;
  2'b?1 : D=¬A;
  default : D = 1'bx; // default to prevent latches
endcase
```

This problem is best fixed by assigning defaults to all the outputs at the start of execution of the procedural block. Again, use don’t cares if they are appropriate.

```verilog
reg A,B,C,D,E;
always @(A or B or C)
casez ([B,C])
  2'b1? : D=A;
  2'b?1 : E=¬A;
  default : E = A;
endcase
```

Remember that a procedural block like this describes the behavior of a block or logic, not a sequence of events. You can reassign the outputs as often as you want within the block.

When you read your Verilog file into Synopsys, it provides a listing of all flip-flops and latches described within the code. For example:
Inferred memory devices in process in routine counter line 27 in file
'/afs/eos.ncsu.edu/users/p/paulf/ece520/examples/countbad.v'.

<table>
<thead>
<tr>
<th>Register Name</th>
<th>Type</th>
<th>Width</th>
<th>Bus</th>
<th>MB</th>
<th>AR</th>
<th>AS</th>
<th>SR</th>
<th>SS</th>
<th>ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_reg</td>
<td>Latch</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Always take a moment to look at the Synopsys log output to make sure that you are not building unintentional latches.

Incomplete Timing Specifications (Sensitivity Lists)

Another common problem that is easy to fix occurs when the designer forgets to make sure that all inputs to a block are specified in the timing control block or sensitivity list. For example:

```verilog
reg A,B,C,D,E;
always @(A or B)
begin
  D = A^B;
  E = C | D;
end
```

In this case, if \( C \) changes, the execution of this block will not be triggered. Since synthesis will build the XOR and OR gate as described, the result will be an inconsistency between the pre-synthesis Verilog and the post-synthesis logic, that can lead to a bug in the design. Remember, in the actual hardware, whenever \( C \) changes the OR gate will ‘execute’ and change state if appropriate. The fix is to include \( C \) in the sensitivity list:

```verilog
reg A,B,C,D,E;
always @(A or B or C)
begin
  D = A^B;
  E = C | D;
end
```

In this example, that there is no need to include \( D \) in the sensitivity list, even though it is an input to the OR gate because \( D \) is produced internally within this block. Synopsys will generate a warning about this type of problem, when the Verilog file is read in, as follows:
Warning: Variable 'D' is being read in routine counter line 24 in file
'/afs/eos.ncsu.edu/users/p/paulf/ece520/examples/countbad.v',
but does not occur in the timing control of the block which begins there. (HDL-180)

Unintentional Specification of Wired-Or Logic

This is a more subtle problem that typically occurs when a designer does not design
before coding and follows the methodology presented in Section 6.2. This type of prob-
lem commonly occurs when the designer is thinking of terms of C software coding instead
of specifying hardware. It is best illustrated by an example:

```
reg A,B,C,D;
always @(posedge clock)
C <= A^B;
always @(posedge clock)
if (D==1) C <= A | B;
```

The logic being described here is illustrated in Figure 6.8. The designer has unintention-
ally specified that the outputs of two different flip-flops are actually the same signal, in a
sense specifying wired-or logic. Most CMOS logic families do not permit wired-or logic
and so this specification cannot be synthesized. To fix this problem, the designer has to
make sure that C is assigned only within one procedural block. Here, the details of the fix
depend on the logical intent but one possibility is as follows:

The existence of this general problem leads to the specification of a general rule:
Every variable must be assigned only within one procedural block or continuous assignment statement.

Again, Synopsys will warn you about this problem when you read the Verilog file in:

```
Warning: Variable 'Z' is driven in more than one process or block in file
/afs/eos.ncsu.edu/users/p/paulf/ece520/examples/countbad.v.
This may cause mismatch between simulation and synthesis. (HDL-220)
```

Improper Use of Loop Constructs

This is another problem that commonly occurs when the designer is thinking in terms of software constructs instead of hardware is the improper use of loop constructs. For example, consider the code fragment in the box below:

```
reg [7:0] A, B;
reg [7:0] C [0:7];
integer i;
always @(posedge clock)
for (i=0; i<=7; i=i+1)
if (A[i]==1) B <= B+C[i];
```

This fragment describes a chain of eight adders that eventually reach a register. It can be seen that this is a slow and inefficient implementation. There is a logic depth of at least eight adders between the input registers and the output register B, giving a slow clock cycle. In addition, the eight adders that are built will take up considerable area. One solution is to ‘break’ the for loop with a clock edge.

```
reg [7:0] A, B;
reg [7:0] C [0:7];
integer i;
always @(posedge clock)
for (i=0; i<=7; i=i+1)
begin
  if (A[i]==1) B <= B+C[i];
  @posedge(clock);
  \ wait for clock edge before proceeding
end
```
This code fragment is implicitly combining datapath and control in that an implicit 8-bit counter is now specified to determine which of \( C[i] \) is presented on each clock cycle. This code is potentially confusing. For example, a novice designer might not realize that this 8-cycle operation is looping continuously. However, this implementation will not have the speed and area penalties of the previous one. A better solution still is to explicitly separate control and datapath and redesign it without a \textbf{for} loop, either as a pipeline or as a single adder and accumulator. This is illustrated below, as controlled by a simple counter.

\begin{verbatim}
reg [7:0] A,B,C;
reg [4:0] countAccumB;
always@(posedge clock)
begin
if (clear==1) countAccumB <= 4'b0; // counter = controller
else if (countAccumB[3] != 1) countAccumB <= CountAccumB + 1'b1;
if (clear==1) B <= 0; // accumulator B
else if (countAccumB[3] != 1)
  B <= B+C[countAccumB[2:0]];
endmodule
\end{verbatim}

In general, loop structures (\textbf{for}, \textbf{while}, and \textbf{repeat}) are best used for iterating over the variables of an array, rather than for describing a sequence of operations on the same variable, as above. An example of good use of a \textbf{for} loop, a simple odd parity generator (generating an output of ‘1’ if \( A \) contains an odd number of 1’s), is coded below:

\begin{verbatim}
reg [7:0] A;
reg OddParity;
always@(A)
begin
  OddParity = 1'b0;
  for (i=0; i<=7; i=i+1)
    if (A[i]==1) OddParity = ~OddParity; // Inverse parity
end
\end{verbatim}

Note, however, that the loop must have a constant termination. Consider the next example:

\begin{verbatim}
reg [7:0] A,N;
reg OddParity;
always@(A)
begin
  OddParity = 1'b0;
  for (i=0; i<=N; i=i+1)
    if (A[i]==1) OddParity = ~OddParity; // Inverse parity
end
\end{verbatim}
Since \( N \) is a variable, the synthesis tool will not know how much hardware to build. Loops that do not have a constant termination cannot be synthesized.

### 6.9 Debugging Strategies

Getting bugs out of a design is often a difficult and tiresome task. There are rarely shortcuts to make the designer's job less difficult. Debugging strategies that generally work well include the following:

- **Localize the bug.** This is a very important step. It is necessary to keep narrowing down the search until the line that is toggling at the wrong time, or the state machine that is spending too long in the wrong state, is found. Obviously, turning on trace mode, viewing all relevant waveforms and judicious use of `$display` and `$monitor` are all helpful. Divide and conquer strategies must sometimes be used to localize the problem.

- **Trace the offending code in your head.** Trace the code by hand as a Verilog simulator would. Often the problem will appear when you notice a branch incorrectly specified or other inconsistency.

- **If all else fails, try something different.** Sometimes, the cause of the problem is still elusive after a lot of effort. In this case, simply trying a different approach to coding the offending piece of the design will often lead to a solution. For example, redesign a case statement so that it is different. Other examples are converting an if-then-else statement to a case statement and changing a filename if there seems to be an Operating System problem, etc.

- **Check the manual.** Often an incorrect approach to coding a particular example can be discovered by reviewing the Language Reference Manual or Synthesis manual. Alternatively, it is possible to glean some ways of doing ‘something different’ by looking at the manual.\(^1\)

### 6.10 Conclusions

The purpose of this chapter was to instruct the reader on how to convert a module-level hardware concept into Verilog RTL. Some important principles enumerated in this chapter include the following.

---

\(^1\) Other debugging tips may be found in sections 4.4, chapter 7, and 15.1
• **Clearly separate control and datapath.** They are separated by the ‘control points’; control lines and status lines. The controller is generally some combination of Finite State Machines and counters.

• **Design before coding.** Create a sketch of the design before writing any Verilog. Clearly identify all registers. Give names to all important signals, including all register outputs.

• **Convert areas of the design to blocks of Verilog code.** Each area of the design becomes one procedural block or one continuous assignment statement. The behavior of registers and their connected input logic is described in a procedural block starting with `always@(posedge clock)`. The behavior of large combinatorial logic blocks is coded in non-clocked procedural blocks, while the logic required for small pieces of combinatorial logic is coded using continuous assignments.

*Watch out for the following common problems:*

• Accidentally instantiating latches by not using defaults in combinational procedural blocks.

• Incomplete sensitivity lists in combinational procedural blocks.

• Assigning the same variable in two different procedural blocks of continuous assignment statements.

• Improperly using loop constructs, resulting in inefficient hardware.

• Assigning the same variable in two different procedural blocks or continuous assignment statements.

• Improper loop constructs, resulting in incorrect or inefficient hardware.
6.11 Exercises

(1) What signals, if any, are latched in the following code fragment? Recode the design to remove any latches.

\[
\text{always}@(\text{foo} \text{ or } \text{fred}) \\
\text{if } (\text{foo}=2\text{\'}h2) \text{ bar } = \text{ fred;}
\]

(2) What signals, if any, are latched in the following code fragment? Recode the design to remove any latches.

\[
\text{always}@(\text{foo} \text{ or hal} \text{ or tron}) \\
\text{begin} \\
\text{case } (\text{foo}) \\
2\text{\'}b00 : \text{ fred } = \text{ hal; } \\
2\text{\'}b01 : \text{ fred } = \text{ tron; } \\
2\text{\'}b10 : \text{ mike } = \text{ hal; } \\
2\text{\'}b11 : \text{ mike } = \text{ tron; } \\
\text{endcase} \\
\text{end}
\]

(3) Redesign the counter described above as a count-up instead of a count-down counter. Design it so that it halts when the count reaches 4'hF and replace the 'zero' flag with a 'ones' flag.

(4) A sequential piece of logic is designed to check to see if any register in a 32-entry register file is equal to zero. It continuously scans port A of the register file, setting the zero flag if any entry contains 32'h0. The zero flag is changed only after register 32 is reached.

A piece of C code describing the function of this unit is:

\[
\text{temp_zero } = 0; \\
\text{for } (i=0; i <=31; i++) \\
\text{if } (\text{RegisterFile}[i] ==0) \text{ temp_zero } =1; \\
\text{zero } = \text{ temp_zero;}
\]

and a Verilog description of the top-level interface to this module is provided as follows:

Assume that the Register File is asynchronous, i.e. whenever the address input changes, the data output (for the port) changes half a clock period later.
Then answer the following questions:

(a) Sketch the logic for this unit, explicitly identifying each register you plan to implement.

(b) Write the Verilog capturing your logic, completing the module `CheckForZero`. Note the following hints:
   - No Reset is needed.
   - If your answer contains a ‘for’ loop, you are probably wrong.

(5) Redesign the counter module as an 8 bit downcounter. In addition, add the input `divide-by-two`. Whenever `divide-by-two` is high and `latch` is low and `dec` is low, divide the current contents of the counter by 2 (by doing a right shift).

(6) You are to design logic to do the following:
   There are two 8-bit counters. Counter 1 is loaded from `in1[7:0]` when the signal `load1` is high, and is decremented by 1 when `dec1` is high. Counter2 is loaded from `in2[7:0]` and is decremented by the amount specified in `in3[7:0]` when `dec2` is high. Whenever the contents of the 2 8-bit counters become the same as each other, then the output flag 'ended' goes high AND the counter contents must remain the same until one of them are reloaded from `in1` or `in2`. Also, if either counter overflows (the counters contain unsigned numbers), then 'ended' should go high AND the counter contents must remain the same until one of them are reloaded from `in1` or `in2`. (Overflow is indicated simply by a carry out = 1 in this case).
   The value on `in3` will be held constant for you. You do not need to register it.

Design and implement your solution. Make sure you include a block diagram sketch of your design, as well as a timing diagram. Also, make your solution consistent with my input/output declarations.

(7) You are to design, and implement in Verilog, a communications interface. Often data sent over a data link are organized as ‘packets’ of data, each packet containing some identification bits, data, and some check bits used to determine if a transmission error has occurred. Your hardware must meet the following specification:
   i.e. This piece of hardware checks 'InData' on each clock cycle. If `InData[12:9]=1`, then it
Design Approaches for Single Modules

**Inputs:**

- **Clock:**
- **Reset:**  // reset is active low
- **Clear:**  // clears output registers - active high
- **InData[11:0]:** // Input data, organized as follows:
  - **InData[11:8]:** contains the 'header'
  - **InData[7:4]:** contains the 'data payload'
  - **InData[3:1]:** are not used
  - **InData[0]:** is a parity bit.
  - it is '1' if InData[18:4] is meant to be Even Parity
  - A new 'InData' arrives every clock

**Outputs:**

- All outputs are registered and are cleared when 'reset' is low or 'clear' is high.
- **Payload[3:0]:** is changed to contain the 'data payload' when 'InData' is of type 1
- **Count[7:0]:** // Total count of type 1 datas
- **Error[7:0]:** // Number of type 1 datas of WRONG parity

Sample Timing:

- Clock
- Clear
- InData 1F1 0E0 170

(ie. 1F1 = 'type 1' payload=F, parity should be even
0E0 = 'not type 0', payload=E, odd parity
170 = 'type 1' payload=7, payload should have even parity)

- Payload 0 F F 7 7
- Count 0 1 1 2 2
- Error 0 0 0 1 1

transfer the middle 4 bits of InData to payload and increments 'count'. At the same time it checks the parity of the middle 4 bits and see if it is as expected. If it is not, then there is a transmission error, and 'error' is incremented.