Verilog: Frequently Asked Questions

Shivakumar Chonnad
Needamangalam Balachander

Verilog: Frequently Asked Questions

Language, Applications and Extensions

Springer

eBook ISBN: 0-387-22899-3
Print ISBN: 0-387-22834-9

To our wives, Manjula Chonnad
and jayanthi Balachander

To our children, Akshata Chonnad,
Puja Balachander, and Manya Balachander

Contents

Contributing Authors

Shivakumar Chonnad is a Staff Engineer at Synopsys Inc. He has been working in the industry for over 15 years, covering the various stages of ASIC Design & Verification, from specification to hardware validation. Shiv currently deals with IP based design and Verification. Shiv has a Bachelor's degree in Electronics and Communications Engineering from the Karnatak University, India. Shiv's areas of professional interest include Design and Verification of IPs.

Needamangalam Balachander is a CAE Manager at Synopsys Inc. He has been working in the industry for over 15 years, covering the areas of system/board-level design & diagnostics, ASIC Design and Verification, and currently deals with mixed-signal IP design and support issues. Bala has a Bachelor's degree in Electronics and Communications Engineering from the Indian Institute of Science in Bangalore, India. He also holds a B.S degree in Physics. Bala's areas of professional interest include Formal Verification methodologies, timing abstractions of mixed-signal IPs, and ATPG issues in mixed-signal IPs.

Foreword

The Verilog Hardware Description Language was first introduced in 1984. Over the 20 year history of Verilog, every Verilog engineer has developed his own personal "bag of tricks" for coding with Verilog. These tricks enable modeling or verifying designs more easily and more accurately. Developing this bag of tricks is often based on years of trial and error. Through experience, engineers learn that one specific coding style works best in some circumstances, while in another situation, a different coding style is best.

As with any high-level language, Verilog often provides engineers several ways to accomplish a specific task. Wouldn't it be wonderful if an engineer first learning Verilog could start with another engineer's bag of tricks, without having to go through years of trial and error to decide which style is best for which circumstance? That is where this book becomes an invaluable resource. The book presents dozens of Verilog tricks of the trade on how to best use the Verilog HDL for modeling designs at various level of abstraction, and for writing test benches to verify designs. The book not only shows the correct ways of using Verilog for different situations, it also presents alternate styles, and discusses the pros and cons of these styles.

When I first received a draft of this book to look over, I expected to read a book that would only be of interest to the beginning Verilog user. I quickly discovered that the tricks of the trade presented in this book are not just for the novice. Even engineers with many years of experience with Verilog will likely find insights on using Verilog, and additional tidbits that they can add to their own bag of tricks. Both novice and experienced Verilog engineers will also benefit from the many references in the book on using the newest generation of Verilog, SystemVerilog.

The authors of this book have done a great job of making it easier for all engineers to become masters of Verilog.

Stuart Sutherland
Verilog, System Verilog and PLI Consultant
Sutherland HDL, Inc.
www.sutherland-hdl.com

Preface

Verilog has been a popular Hardware Description Language (HDL) since the mid 80's. Its popularity has increased with the addition of many new enhancements into it. Some key reasons for the adoption of Verilog as the language of choice for designers are the simplicity of the language usage and the availability of high-performance simulators from multiple EDA vendors, which results in reduced execution time for large regression simulations.

Like any other programming language, experienced users of Verilog are fully aware of the language's capabilities, and have amassed a "bag of tricks", gathered in the course of execution of multiple projects. Beginners to the language are often consumed by questions relating to the implications of coding styles on synthesis, static timing, power etc. It is important to factor in these functional and environmental implications as part of the RTL coding stage of the ASIC design process. Not doing so could result in expensive iteration cycles.

This book is for digital designers who use Verilog as the HDL for their design and verification. This book will also be useful to those who have learned Verilog, and would like to use the various language-constructs, but have questions on the capabilities of these constructs. Although the same functionality can be implemented by coding in many different styles, some of the questions that arise during coding would be:

Is this the right construct to infer the required logic?
Is this the best way to implement the required functionality?
Does this approach help in meeting the design constraint?

By reading this book, the user is presented with:

- Multiple coding styles that are appropriate to specific design constraints such as area, timing, power, etc.
- Examples of logic inferred for different constructs or coding styles
- Illustrations of commonly encountered problems, so that the user can incorporate the style or approach that helps eliminate the problem aprior
- Implications of particular approaches or styles on design constraints.

We assume that the user has a very basic familiarity with the Verilog HDL. Readers who have a basic or intermediate level of expertise in the language can also refer to this book to know more implementation details of using the HDL in the different contexts of design, verification and implications to synthesis, static timing, etc.

In this book, the authors have delved into many different front end topics of RTL such as synthesis, area, power, testability, etc. Most issues typically encountered during these stages have been presented in the form of FAQs. Whenever there is more than one approach to meet a requirement, the pros and cons of each approach are presented.

We hope the book will also interest students who are learning Verilog for the first time. We believe that this book provides answers to many questions that normally pop up as students begin to use the language.

This book deals only with the front end issues, i.e., until completion of functional verification and synthesis with estimated wiring information. The book does not discuss any back-end issues like placement, floor-planning, or routing. The back-end processes are highly customized to the tools that implement them. Wherever appropriate, the implications of the coding style that would have an effect on the back-end steps are illustrated. This helps avoid expensive iterations in revisiting the golden code, in order to eliminate these back-end gotchas.

This book does not aim to teach the Verilog language for a novice user. Instead, we endeavour to address the various issues that typically arise in Verilog based chip design projects. Users who wish to learn Verilog from scratch may also refer to the Verilog Language Reference Manual (LRM), or some of the excellent books already available like "The Verilog Hardware Description Language" by Thomas & Moorby, and "SystemVerilog for Design" by Suart Sutherland, et al. The details of the syntax and the constructs, etc. are not explained within the book, and readers can refer to the LRM for this. In case of any contradiction of the contents in this book with the LRM, the content in the LRM is the final authority.

Throughout the book, we have tried to use simple examples that illustrate the point that is being made regarding the capability of the language. In certain examples where the illustrated RTL might not have been the most optimal way to code, we have deliberately illustrated it sub-optimally, to show what functionality or logic gets inferred out of that style of code. These simple working examples can be extrapolated and used in larger designs. A few times, only a snippet of the full RTL is presented, without the obligatory declarations (such as ***module, endmodule, input, output***) etc. These are assumed predefined by the users. Wherever appropriate, we have also included simplified schematics of the outcome of the synthesized results.

We have verified every RTL example with a simulator and a synthesis tool. In order to illustrate some of the capabilities or the limitations in the language, we have coded some RTL examples in particular styles, or using particular constructs. For the most part however, we have coded RTL examples in the most timing and area optimal approach.

Although this book does not provide the answers to all the possible questions that can arise, we hope it will address the most commonly encountered problems. We believe that this book will help readers make more informed choices between approaches in achieving functionality and constraints in their VLSI projects. Based on the feedbacks we receive, and more findings of interesting issues, we hope to keep this as an ongoing activity of incorporating more FAQs and their answers in the future editions.

This book is unique, because it addresses complex language issues, along with guidelines to address the coding, timing and synthesis issues, reliability of designs, and verification in the form of FAQs. It captures many scenarios and issues that have been encountered while dealing with complex pieces of IP during various stages of the project cycle. It also addresses the three versions of Verilog that current users must contend with:

Verilog ‘95
Verilog 2001
SystemVerilog 3.1a

Wherever applicable, we have also compared the coding semantics between the different Verilog versions from Verilog-95 to SystemVerilog.

The general organization of these topics have been categorized into different chapters as follows:

Chapter 1 : Basic Verilog discusses a few important constructs of Verilog and comparisons of what their implications mean in a Verilog based environment.

Chapter 2 : RTL Design discusses the various RTL design and synthesis related FAQs. This chapter will be of real interest to the RTL designers as it discusses the comparison of different coding constructs and styles. The chapter also discusses issues seen during design for area, timing, testability and power.

Chapter 3 : Verification emphasizes using Verilog constructs for Verification. The various issues and considerations for design of Bus Functional Model's and Bus Monitors are discussed in this chapter. This chapter will be of special interest to readers with verification responsibilities. It also discusses the various mechanisms of random stimulus generation and examples of the different mechanisms.

Chapter 4 : Miscellaneous has all the FAQs that do not explicitly fall in any of the above chapters of RTL and Verification. It discusses the subtle and interesting scenarios of using Verilog at a system level.

Chapter 5 : Common Mistakes illustrates most of the commonly made mistakes in the use of Verilog for design or verification. The chapter discusses how the functional issues go undetected, even though it goes through the compile stage without any errors. Any workaround's to prevent or detect these mistakes have also been illustrated appropriately.

Chapter 6 : Verilog during Simulation Regressions illustrates the different requirements seen during simulation regression, and how different constructs of Verilog can be incorporated within the testbench that will help during regressions.

Verilog is a registered trademark of Cadence Design Systems. Since the above chapters have been categorized to address the different topics like design and verification separately, some readers may find it suitable to directly begin with these chapters. The authors, however, recommend reading from Chapter 1 onwards until the end, to understand different issues presented through out the design cycle.

Also, the Table of Contents consists directly of the FAQs themselves. Therefore, by simply browsing through the Table of Contents, readers can determine if their particular questions or topics have been dealt with in the book.

Acknowledgments

This book would not have been possible without the help and support of the management of Synopsys Inc. Access to Synopsys' tools has been instrumental in verifying the concepts and examples in this book.

We have been extremely fortunate that this book was reviewed by Stuart Sutherland of Sutherland HDL Inc. His detailed review of the manuscript provided expert confirmation of our understanding of the Verilog language and the new SystemVerilog extensions to Verilog.

We gratefully acknowledge the following people who despite their work schedules, reviewed the drafts of this book throughout its development and providing valuable feedback and suggestions.

Warren Savage, Phil Dworsky, Arulmani Krishnan, Vijay Coimbatore, Kiran Kavoori, Haidar Ahmad, Sourabh Tandon, Manickam Kandaswamy, Bill Rogers, Veeresh Hullatti

Finally, we would like to express our gratitude to **Michael Hackett, Carl Harris** and the staff of Kluwer Academic Publishers, for their encouragement and support throughout the development of this book.

Chapter 1

BASIC VERILOG

INTRODUCTION

This chapter addresses frequently asked questions on the basics of the Verilog hardware description language. This chapter deals with FAQs on Verilog assignments, tasks, functions, parameters, and ports. These constructs form a large section of the Verilog code and interconnection in designs.

1.1 Assignments

The following section discusses the different kinds of assignments that are possible in Verilog, and what their features are.

1.1.1 What are the differences between continuous and procedural assignments?

The following table captures the differences between continuous and procedural assignments:

Table 1-1. Differences between continuous and procedural assignments

Continuous assignment	Procedural assignment
Assigns values primarily to **nets**	Assigns values primarily to **reg** variables
Variables and nets **continuously** drive values onto ports	Results of calculations involving variables and nets can be **stored** into variables

Continuous assignment	**Procedural assignment**
Used to infer combinatorial logic	Used to infer both storage elements like Flip-flops and latches and also combinatorial logic
Assignment occurs whenever the value on the RHS of the expression changes as a continuous process	The value of the previous assignment is held until another assignment is made to the variable
Occurs in assignments to ***wire, port, and net*** type	Occurs in constructs like ***always, initial, task, function***

For example,
`wire out1 = in1 & in2;`
or
`assign out1 = in1 & in2;`
	For example,
	`always @(posedge clk)`
	`   reg1 <= in1;`
	`always @(a or b or s)`
	`   y = (s == 1) ? a : b;`

1.1.2 What are the differences between assignments in *initial* and *always constructs?*

While both ***initial*** and ***always*** constructs are procedural assignments, they differ in the following ways:

Table 1-2. Differences between initial and always blocks

initial	**always**
Assignments in an initial block begin to execute from time 0 in simulation, and proceed in the specified sequence.	Assignments in an always block also begin from time 0, and repeat forever as a function of the changes on the blocks sensitivity list
Execution of statements in an initial begin-end block stops when the end of the block is reached, i.e., executed only once during simulation	Execution continuously repeats from the begin to the end of the process unless held by a ***wait*** construct throughout the simulation session
Non-synthesizable construct	Synthesizable construct

For example,
`reg [1:0] out1, out2;`
`initial begin`
`  out1 = 2'b10;`
`  #5 out2 = 2'b01;`
`end`
	For example,
	`reg [1:0] out1, out2;`
	`always @(posedge clk)`
	`begin`
	`  out1 <= in1;`
	`  out2 <= out1 & in2;`
	`end`

1.1.3 What are the differences between blocking and nonblocking assignments?

While both blocking and nonblocking assignments are procedural assignments, they differ in behaviour with respect to simulation and logic synthesis as follows:

Table 1-3. Differences between blocking and nonblocking assignments

Blocking assignments	**Nonblocking assignments**
In a blocking assignment, the evaluation of the expression on the RHS is updated to the LHS variable autonomously based on the delay value (either 0 if no delay specified, or scheduled as a future event if a non-0 value is specified)	Nonblocking assignment to LHS is *scheduled* to occur when the next evaluation cycle occurs in simulation and not immediately. Updates are not available immediately within the same time unit
When multiple blocking assignments are present in a process, the trailing assignments are blocked from occurring until the current assignment is completed	Multiple nonblocking assignments can be scheduled to occur concurrently on the next evaluation cycle in simulation
There is a possibility of race conditions on the variables of blocking assignments if assignments happen to it from two processes concurrently	The race conditions are avoided as the updated value is assigned after evaluation
Recommended to use within combinatorial **always** blocks	Recommended to use within the sequential **always** blocks
Can be used in procedural assignments like ***initial***, ***always*** and continuous assignments to nets like ***assign*** statements	Can be used only in the procedural blocks like ***initial*** and ***always***; Continuous assignment to nets like the ***assign*** statement is not permitted
Represented by "=" operator sign between LHS and RHS	Represented by "<=" operator sign between LHS and RHS

Blocking assignments	**Nonblocking assignments**
For example, `initial begin` `reg1 = #10 2'b10;` `reg2 = #5 2'b01;` `end` Starting from time 0, `reg1` will be assigned 2'b10 at time 10 units and `reg2` assigned 2'b01 at time 15 unit. Assignment to `reg2` happens after the assignment of `reg1`	For example, `initial begin` `reg1 <= #10 2'b10;` `reg2 <= #5 2'b01;` `end` Starting from time 0, `reg2` will be assigned 2'b01 at time 5 units and `reg1` will be assigned 2'b10 at time 10 unit. Assignment to `reg2` happens earlier than `reg1`

1.1.4 How can I model a bi-directional net with assignments influencing both source and destination?

The ***assign*** statement constitutes a continuous assignment. The changes on the RHS of the statement immediately reflect on the LHS net. However, any changes on the LHS don't get reflected on the RHS. For example, in the following statement, changes to the `rhs` net will update the `lhs` net, but not vice versa.

```
wire rhs, lhs;
assign lhs = rhs;
```

System Verilog has introduced a keyword ***alias,*** which can be used only on nets to have a two-way assignment. For example, in the following code, any changes to the `rhs` is reflected to the `lhs`, and vice versa.

```
module test_alias;

wire [3:0] lhs, rhs;

alias lhs = rhs; // two way assignment

initial begin
  force rhs = 4'h2;
  $display ("lhs = %0h, rhs = %0h", lhs, rhs);
  release rhs;

  force lhs = 4'hc;
  $display ("lhs = %0h, rhs = %0h", lhs, rhs);
  release lhs;
end

endmodule // test_alias
```

Had the above ***alias*** command been ***assign***, the outputs of the above display outputs would be as follows:

```
lhs = 2, rhs = 2
lhs = c, rhs = z
```

However, with the ***alias*** command as it is, the outputs are as follows:

```
lhs = 2, rhs = 2
lhs = c, rhs = c
```

In the above example, any change to either side of the net gets reflected on the other side.

1.2 Tasks and Functions

This section discusses the different FAQs on ***task*** and ***function*** in Verilog. The section also discusses a few advancements on these constructs in System Verilog.

1.2.1 What are the differences between a *task* and a *function*?

Both ***tasks*** and ***functions*** in Verilog help in executing common procedures from different places in a module. They help in writing cleaner and maintainable code, by avoiding replication at different places in a module. Essentially, ***functions*** and ***tasks*** provide a “subroutine” mechanism of reusing the same section of code at different places in a module. This allows for easier maintenance of the code.

However, the ***tasks*** and ***functions*** differ in the following aspects:

Table 1-4. Differences between tasks and functions

task	***function***
Can contain time control statements like @(***posedge*** .), delay operator (#)	Executes in zero simulation time

task	***function***
Can call any number of function's or tasks within itself	Can call any number of ***function***'s within itself
Cannot return any value when called; instead the ***task*** can have output arguments	Returns a single value when called. In SytemVerilog the return value can be optionally voided
For example, `gt_result` is an output of a ***task*** to calculate the result of the greater of two input arguments `arg1` and `arg2` `greater_val(arg1, arg2, gt_result);`	For example, `gt_result` is assigned the return of a ***function*** call to calculate the result of the greater of two input arguments `arg1` and `arg2` `gt_result = greater_val(arg1, arg2)`

1.2.2 Are *tasks* and *functions* re-entrant, and how are they different from static task and function calls? Illustrate with an example.

In Verilog-95, tasks and functions were not re-entrant. From Verilog version 2001 onwards, the ***tasks*** and ***functions*** are reentrant. The reentrant ***tasks*** have a keyword ***automatic*** between the keyword ***task*** and the name of the ***task.*** The presence of the keyword ***automatic*** replicates and allocates the variables within a ***task*** dynamically for each task entry during concurrent ***task*** calls, i.e., the values don't get overwritten for each ***task*** call. Without the keyword, the variables are allocated statically, which means these variables are shared across different ***task*** calls, and can hence get overwritten by each ***task*** call.

The following example illustrates the effect of the keyword ***automatic*** for re-entrant tasks. This is a <u>non-synthesizable</u> code for the purpose of illustration only.

```
module modify_taskval;

integer out_val;

task automatic modify_value;
  input [1:0] in_value;
  output [3:0] out_value;
  reg [1:0] my_value;
begin
// syntax error to use nonblocking assignment with
// automatic variables
  my_value = in_value; // blocking assignment
#5
  $display("my_value = \t%0d, t = %0d",
            my_value, $time);
  out_value = my_value + 2;
end
endtask

initial begin
  fork
    begin // First parallel call
      #1
      $display("in1= \t\t%0d, t = %0d",2, $time);
      modify_value(2, out_val);
    end
    begin // Second parallel call
      #2
      $display("in2 = \t\t%0d, t = %0d",3, $time);
      modify_value(3, out_val);
    end
  join
end

endmodule
```

In the above example, `my_value` is a local variable in the ***task*** `modify_value`. Whenever this ***task*** is called, the input `in_value` is assigned to the local variable after 5 simulation timeunits. Within the ***initial-begin***, there is a ***fork-join***, which launches two parallel processes. One starts after simulation timeunit #1, and other after #2. The first process assigns a value of 2 to the output of the task, and the second one assigns a value of 3 to the output of the task. Running the simulation with the above code, but without the ***automatic*** keyword, provides the following display:

```
in1 =          2, t = 1  // passed value is 2
in2 =          3, t = 2
my_value =     3, t = 6  // retained value is 3
my_value =     3, t = 7
```

The sequence of events without the keyword ***automatic*** is as follows:

1. The launch of the two processes from the ***fork-join*** happens from time 0.
2. The first process calls `modify_value` after #1, and the local variable `my_value` is assigned the value 2. This happens at t=1.
3. The second process calls `modify_value` after #2 and the local variable `my_value` is assigned the value 3. This happens at t=2. Note that the earlier value of 2 assigned to the local variable `my_value` is now overwritten with the value 3.
4. After 4 more time units i.e., at t = 1+5=6, the display of the first ***task*** call becomes active. Since the latest value is now "3", based on the previous step, the value of "3" is displayed for `my_value`, instead of what was passed as "2".
5. Similarly, for the second process i.e., 2+5=7, the display of the second ***task*** call becomes active. Since the latest value is still "3", the value of "3" is displayed for `my_value` here too.

The critical replacement happened in step 3 above, wherein the launch of the 2nd process actually overwrote the value of the first process *before* its turn to display. This occurred because without the ***automatic*** keyword, the variables within the task were ***static***, and shared by all calls to the ***task.***

Now, with the keyword ***automatic*** between the ***task*** and ***task*** name, the following is the output:

```
in1 =          2, t = 1 //passed value is 2
in2 =          3, t = 2
my_value =     2, t = 6 //passed value 2 preserved
my_value =     3, t = 7
```

Following the same steps as above, this time, due to the presence of the keyword ***automatic,*** the unique values of the variables are preserved in each call, and not overwritten by the subsequent ***task*** calls before the variable is being used.

The same explanation holds true for recursive ***function*** calls where a function calls itsef, with the placement of keyword ***automatic*** between ***function*** and the function name.

Note that the keyword ***automatic*** has influence only within the current hierarchy of the concurrent ***task*** calls. The same ***task*** called within separate module hierarchy doesn't overlap, and hence the need for ***automatic*** construct doesn't exist for that scenario.

The following table summarizes the differences between a reentrant ***task*** from a static ***task*** call:

Table 1-5. Differences between reentrant and static tasks

Reentrant task	**Static task**
Has the keyword ***automatic*** between the ***task*** keyword and identifier	*Doesn't* have the keyword ***automatic*** between the ***task*** keyword and the identifier
Variables declared within the task are allocated dynamically for each concurrent task call	Variable declarations within the task are allocated statically
All variables will be replicated in each concurrent call to store state specific to that invocation	Each concurrent call to the task will OVERWRITE the statically allocated local variables of the task from all other concurrent calls to the task
Variables declared are de-allocated at the end of task invocation	Variables retain their values between invocations
Task items cannot be accessed by hierarchical inferences	Task items can be accessed by hierarchical inferences
Task items shall be allocated new across all uses of the task executing concurrently	Task items can be shared across all uses of the task executing concurrently

1.2.3 How can I override variables in an automatic task?

By default, all variables in a ***module*** are static, i.e., these variables will be replicated for all instances of a module. However, in the case of ***task*** and ***function,*** either the ***task/function*** itself or the variables within them can be defined as ***static*** or ***automatic.*** The following explains the inferences through different combinations of the ***task/function*** and/or its variables, declared either as ***static*** or ***automatic***:

1. No ***automatic*** definition of ***task/function*** or its variables

This is the Verilog-1995 format, wherein the ***task/function*** and its variables were implicitly ***static.*** The variables are allocated only once. Without the mention of the ***automatic*** keyword, multiple calls to ***task/function*** will override their variables.

2. ***static task/function*** definition

System Verilog introduced the keyword ***static***. When a ***task/function*** is explicitly defined as ***static***, then its variables are allocated only once, and can be overridden. This scenario is exactly the same scenario as before.

3. ***automatic task/function*** definition

From Verilog-2001 onwards, and included within SystemVerilog, when the ***task/function*** is declared as ***automatic*,** its variables are also implicitly ***automatic*.** Hence, during multiple calls of the ***task/function***, the variables are allocated each time and replicated without any overwrites.

4. ***static task/function*** and ***automatic*** variables

SystemVerilog also allows the use of ***automatic*** variables in a ***static task/function*.** Those without any changes to ***automatic*** variables will remain implicitly ***static***. This will be useful in scenarios wherein the implicit static variables need to be initialised before the ***task*** call, and the ***automatic*** variables can be allocated each time.

5. ***automatic task/function*** and ***static*** variables

SystemVerilog also allows the use of ***static*** variables in an ***automatic task/function*.** Those without any changes to ***static*** variables will remain implicitly ***automatic*.** This will be useful in scenarios wherein the static variables need to be updated for each call, whereas the rest can be allocated each time.

1.2.4 What are the restrictions of using *automatic* tasks?

The following are the restrictions of using ***automatic*** tasks:

- Only blocking assignments can be used on ***automatic*** variables. Refer to the earlier FAQ 1.2.2 for an example on this.

- The variables in an ***automatic task*** shall not be referenced by procedural continuous assignments or procedural force statements. In the following code, the variable `my_value` in the ***task*** cannot be referenced by an ***assign*** statement.

```
task automatic modify_value;
  input [1:0] in_value;
  reg [1:0] my_value;
begin
  my_value = in_value;
end
endtask

initial begin
  force modify_value.my_value = 1; // not allowed
  $monitor (modify_value.my_value); //not allowed
end
```

- They shall not be traced by system calls like ***$monitor*** and ***$dumpvars*** as illustrated in the above example.

1.2.5 How can I call a function like a task, that is, not have a return value assigned to a variable?

Until Verilog 2001, any ***function*** call must return a value to the type ***reg***, ***integer***, ***real***, ***time or realtime*** and the code calling the ***function*** must receive the return value. For example, the following is a syntax error:

```
function my_funct;
. . .
endfunction

initial begin
  my_funct(..); // MUST have a destination
end
```

The line in the above example is a syntax error, since the call of `my_funct` does not have a destination. Only a ***task*** can be called without a destination value.

SystemVerilog has introduced a construct ***void*** to facilitate a voided ***function*** call, that is, there is no destination for the ***function*** call. This would make a ***function*** call similar to a ***task*** call. With System Verilog, functions can also have output and inout arguments. The following example illustrates a voided ***function*** call:

```
module func_1bit;

reg [31:0] int_result;    // Global variable
```

```
function void my_func;
  input [31:0] in1;
  input [31:0] in2;
  output [31:0] out1;
// no need to assign the function
// my_func = in1 + in2;
  int_result = in1 + in2;
endfunction

initial begin
  my_func(3,4,int_result);//no destination required
  $display("int_result = %0d",int_result);
end

endmodule
```

The above example displays the result of `int_result` = 7. Some key observations in the above example are:

- The assignment to the ***function*** `my_func` was not required, since its return value is ***void***.
- The 32 bit return range between the keyword ***function*** and `my_func` was also not required, since it is now a ***void*** return.
- The call of the ***function*** `my_func` within the ***initial-begin-end*** does not require a destination, since the return has been voided.
- Some other intermediate variable like `int_result` declared in the above example at the scope of that ***module*** can still be modified within the voided ***function***.
- SystemVerilog also allows functions with a return to be called as a task by casting the function call to void. For example:
  ```
  initial
    void (my_func(...));
  ```

1.2.6 What are the rules governing usage of a Verilog *function*?

The following rules govern the usage of a Verilog ***function*** construct:

- A ***function*** cannot advance simulation-time, using constructs like #, @. etc.
- A ***function*** shall not have nonblocking assignments.
- A ***function*** without a range defaults to a one bit ***reg*** for the return value.

- It is illegal to declare another object with the same name as the ***function*** in the scope where the function is declared.

1.3 Parameters

The following section discusses a few questions about the usage of parameters, pros and cons of the different approaches and what's new in System Verilog regarding parameters.

1.3.1 How can I override a module's *parameter* values during instantiation?

If a Verilog module uses parameters, there are two ways to override its values. Note that only parameters can be overridden. The localparam and specparam parameters cannot be overridden.

1.3.1.1 During instantiation

In this method, the new values are assigned inline during module instantiation. There are two ways to override during instantiation.

1.3.1.1.1 Assignment by ordered list

In this method, the order in which the parameters are assigned follow the order in which they are declared within the module. For example, the module `parameter_list` contains two parameters, that is, `width` and `depth`, that have been assigned default values within the module. It is instantiated in the following module, `example_parameter_list`, with examples of these parameters overridden with different values in different instantiations.

```
module parameter_list (addr, data); //1995 format
parameter width = 32;
parameter depth = 64;
parameter num_buses = 44;
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
...
endmodule
```

The same example above can be represented in the Verilog 2001 in the following format, in which the ***parameter*** declarations between the module and ***input/output*** declaration are now declared before the ***module*** port list.

```
module parameter_list
  # (parameter width = 32, // 2001 format
     parameter depth = 64,
     parameter num_buses = 4)
    (addr, data);
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
...
endmodule

module example_ordered_list;
reg [127 : 0] a;
reg [255 : 0] b;
reg [ 63 : 0] c;
reg [31 : 0] d;

// Instantiating parameter_list module and
// overriding width only
parameter_list #(128) U0 (a, c);

// Instantiating parameter_list module and
// overriding width and depth only
parameter_list #(128, 256) U1 (a, b);

// Instantiating parameter_list module and
// overriding num_buses only
parameter_list #(32, 256, 8) U2 (d, b);

endmodule
```

The restriction of using the above method is:

- The parameter override values have to be contiguous, that is, any ***parameter*** cannot be skipped during override. For example, in the above code with `U2` instantiation, the ***parameter*** `width` and `depth` cannot be skipped while trying to override `width` and `num_buses` only.

Two methods to overcome this restriction are:

- Precede the order of declaring the parameters within the module with the ones that will change, placing the subset that doesn't change later in the order. For example, in the above code with `U0` and `U1` instantiations, the `num_buses` was not required to be changed, and was last in the priority. The default value of 4, assigned to it within the module, will hold true in these two instantiations.

- Assign values to ALL the parameters, including the ones that don't need to be changed. In instantiation `U2`, although only the `num_buses` ***parameter*** needed to be changed, but the `width` and `depth` ***parameter***'s still required to be assigned with the same default value as in the module definition.

1.3.1.1.2 Assignment by name

This is a new feature, available from Verilog-2001 onwards. This is a better approach of overriding the module ***parameter*** by which the parameters are overridden by explicitly specifying the ***parameter*** name and its overriding value. This way, the ***parameter*** value is linked to its name, and not position of declaration.

Using the same module `parameter_list` as defined above, the following example shows the same ***parameter*** overriding, this time specifying by name.

```
module example_by_name;
reg [127 : 0] a;
reg [255 : 0] b;
reg [63 : 0]  c;
reg [31 : 0]  d;

// Instantiating parameter_list module and
// overriding width only
parameter_list #(.width(128)) U0 (a, c);

// Instantiating parameter_list module and
// overriding width and depth
parameter_list #(.width(128), .depth(256))
U1 (a, b);
```

```
// Instantiating parameter_list module and
// overriding depth only
parameter_list #(.depth(256)) U2 (d, b);

endmodule
```

Note that explicit ***parameter*** names were followed by their overriding values in the parenthesis. In the case of U2, just specifying the depth was sufficient, without having to specify anything for width parameter.

1.3.1.2 Using *defparam*

In this method, the ***parameter*** within a ***module*** is accessed by its hierarchical name from anywhere within the scope of the hierarchy. In the following example, the lower level module parameter_list gets instantiated in the example_defparam module. But the values of width and depth are overridden using the ***defparam*** construct.

```
module example_defparam;
reg [127 : 0] a;
reg [255 : 0] b;
reg [63  : 0] c;
reg [31  : 0] d;

// Instantiating parameter_list module and
// overriding width only
parameter_list U0 (a, c);
defparam U0.width = 128;

// Instantiating parameter_list module and
// overriding width and depth
parameter_list U1 (a, b);
defparam U1.width = 128;
defparam U1.depth = 256;

// Instantiating parameter_list module and
// overriding depth only
parameter_list U2 (d, b);
defparam U2.depth = 256;

endmodule
```

The following bullet items summarize the advantages of using the ***defparam*** approach:

- The ordered sequence need not be maintained in overriding the ***parameter*** values.
- A specific ***parameter*** can be overridden rather than re-specifying all the parameters prior to the one that's being overridden.
- Can help with code maintenance by grouping all the ***defparam***'s collectively in a single place, which can be compiled with the rest of the code.

Parameter redefinition at instantiation is <u>the recommended</u> style by most expert Verilog users. There are several reasons to avoid using ***defparam*** for parameter redefinition. Some of the reasons are:

1. The ***defparam*** statements if not collectively present in one place, can be buried in any module, anywhere in the design hierarchy, making code difficult to maintain or reuse (a form of spaghetti code, which should always be avoided).
2. Since the ***defparam*** statements can be buried anywhere in the hierarchy, they can prevent the Verilog language compilers from being able to do true independent compilation of the modules.
3. Since multiple ***defparam*** statements can be made to the same parameter instance, the final value of the parameter in this situation can (and probably will be) different with different tools.
4. The ***defparam*** statements are not supported in the official IEEE 1364.1-2002 synthesis subset for Verilog
5. The IEEE 1364 standards committee is considering a proposal to deprecate ***defparam*** in the next version of the Verilog standard, making the ***defparam*** an obsolete construct.

1.3.2 What are the rules governing *parameter* assignments?

The rules governing the ***parameter*** assignments are as follows:

- The ***parameter*** override at instantiation can be done either by specifying an ordered list or by name, but not a mix of both. For example, the following is an incorrect way of specifying both `width` and `depth`.

```
parameter_list (128, .depth(256)) U_wrong (a,b);
```

- While assigning the ***parameter*** during instantiation, once a ***parameter*** has been assigned a value, there cannot be another assignment to the same ***parameter***. For example, specifying the `width` ***parameter*** twice within the same instantiation is illegal.

```
parameter width = 64;
parameter width = 128;
// Specifying the same parameter more than once
   // is an Error
```

- If a ***parameter*** is assigned both by a ***defparam*** and in the module's instantiation, the ***defparam***'s assignment takes precedence. In the following example, the `width` ***parameter*** is instantiated with value 128, but a ***defparam*** to the same ***parameter*** with the value 64 also follows it, then the ***defparam*** gets precedence, and `width` will finally have the value 64.

```
   parameter_list #(128) U1 (a, b);
   defparam U1.width = 64; // This statement "wins"
```

1.3.3 How do I prevent selected *parameters* of a module from being overridden during instantiation?

If a particular ***parameter*** within a module should be prevented from being overridden, then it should be declared using the ***localparam*** construct, rather than the ***parameter*** construct. The ***localparam*** construct has been introduced from Verilog-2001. Note that a ***localparam*** variable is fully identical to being defined as a ***parameter,*** too. In the following example, the ***localparam*** construct is used to specify `num_bits`, and hence trying to override it directly gives an error message.

```
module localparam_list (addr, data);
parameter width = 32;
parameter depth = 64;
localparam num_bits = width * depth;
input  [width-1 : 0] addr;
input  [depth-1 : 0] data;
. . .
endmodule
```

Note, however, that, since the `width` and `depth` are specified using the ***parameter*** construct, they can be overridden during instantiation or using ***defparam***, and hence **will indirectly override** the `num_bits` values.

In general, ***localparam*** constructs are useful in defining new and localized identifiers whose values are *derived* from regular ***parameter***s.

1.3.4 What are the differences between using *`define*, and using either *parameter* or *defparam* for specifying variables?

Both ***`define*** and ***parameter*** constructs can be used to specify constants in the design. For example, the `width` ***parameter*** can be specified either as a ***`define*** or ***parameter***, as:

```
`define width 64
if (`width == 64) ...
or
parameter width 64;
if (width == 64) ...
```

However, the following are a few differences in using the two constructs:

Table 1-6. Differences between `define and parameter/defparam

`define	***parameter/defparam***
`define is basically a text substitution macro	***Parameter*** is used to specify constants in a design
Multiple ***`defines*** to the same variable name are not allowed, the final value of the macro is determined by source code order	Although multiple ***parameter*** definitions to the same variable are not allowed within a module, multiple ***defparam***'s to the same variable are allowed, however the final value of the parameter is indeterminate
Cannot be overridden in any mechanism	***Parameter*** can be overridden
Only one constant with the given name can exist in the full scope	Multiple modules can have the same ***parameter*** name, as it is limited to that scope only

1.3.5 What are the pros and cons of specifying the parameters using the *defparam* construct vs. specifying during instantiation?

The advantages of specifying parameters during instantiation method are:

- All the values to all the parameters don't need to be specified. Only those parameters that are assigned the new values need to be specified. The unspecified parameters will retain their default values specified within its module definition.

- The order of specifying the ***parameter*** is not relevant anymore, since the parameters are directly specified and linked by their name.

The disadvantage of specifying ***parameter*** during instantiation are:

- This has a lower precedence when compared to assigning using ***defparam***.

The advantages of specifying ***parameter*** assignments using ***defparam*** are:

- This method always has precedence over specifying parameters during instantiation.

- All the ***parameter*** value override assignments can be grouped inside one module and together in one place, typically in the top-level testbench itself.

- When multiple ***defparams*** for a single ***parameter*** are specified, the ***parameter*** takes the value of the last ***defparam*** statement encountered in the source if, and only if, the multiple ***defparam***'s are in the same file. If there are ***defparam***'s in different files that override the same parameter, the final value of the parameter is indeterminate.

The disadvantages of specifying ***parameter*** assignments using ***defparam*** are:

- The ***parameter*** is typically specified by the scope of the hierarchies underneath which it exists. If a particular module gets ungrouped in its hierarchy, [sometimes necessary during synthesis], then the scope to specify the ***parameter*** is lost, and is unspecified.

For example, if a module is instantiated in a simulation testbench, and its internal parameters are then overridden using hierarchical ***defparam*** constructs (For example, ***defparam*** `U1.U_fifo.width = 32;`). Later, when this module is synthesized, the internal hierarchy within `U1` may no longer exist in the gate-level netlist, depending upon the synthesis strategy chosen. Therefore post-synthesis simulation will fail on the hierarchical ***defparam*** override.

See the earlier FAQ 1.3.1.2 for additional disadvantages of ***defparam*** and why this construct should not be used.

1.3.6 What is the difference between the *specparam* and *parameter* constructs?

The ***specparam*** is a special kind of ***parameter*** that is intended to specify only timing and the delay values. The key differences in using the ***specparam*** and the ***parameter*** constructs are:

Table 1-7. Differences between specparam and parameter

specparam	***parameter***
Can be defined within **both** module and specify block	**Must** be defined **outside** the specify block and within module
A ***specparam*** can be assigned using another ***specparam*** or ***parameter*** or a combination of both	***Parameter*** cannot be assigned the value of a ***specparam***
Value is overridden using SDF annotation	Can be overridden during instantiation or using ***defparam***

1.3.7 What are derived parameters? When are derived parameters useful, and what are their limitations?

When one or more parameters are used to define another parameter, then the result is a derived parameter. The derived parameter can be either of the type ***parameter*** or ***localparam.*** In the following example, two parameters, `width` and `depth`, can be used to define a third parameter, `num_bits`. In this case, the `num_bits` takes a value of 32.

```
module derived_param;
parameter width = 4;
parameter depth = 8;
// num_bits is a derived parameter
```

```
localparam num_bits = width *depth;
endmodule
```

The advantages of using derived parameters are:

- Makes the RTL code reusable
- Enables use of the shorter name of `num_bits` instead of completely specifying (`width` * `depth`)

The consequence of using derived parameters is that derived parameters can be *indirectly* overridden by overriding their dependent parameters through ***defparam*** constructs. So, ***localparam*** constructs should be used with care when defining derived parameters.

1.4 Ports

The following section discusses a few questions about the usage of ports, pros and cons of the different approaches of port connections, and what's new in SystemVerilog regarding ports.

1.4.1 What are the different approaches of connecting ports in a hierarchical design? What are the pros and cons of each?

While instantiating the sub-modules in a given hierarchy, the port connections to those modules can be done in one of five ways:

1.4.1.1 Ordered port connection

In this method, the port expressions listed for module instance shall be in the same order as the ports listed in the ***module*** declaration, that is, the first element in the list is connected to the first port declared, the second element to the second port and so on. For example, in the code below, the `upper` module instantiates a `lower` module, and the ports are implicitly connected, that is, the connection is based on order **and** position.

```
module lower (addr, data);
input  [width-1 : 0] addr;
inout  [depth-1 : 0] data;
endmodule // lower

module upper (in1, out1);
input  [width-1 : 0] in1;
```

```
output  [depth-1 : 0] out1;

lower U0 (in1, out1); // implicit connection of
// in1 to addr and out1 to data ports

endmodule // upper
```

1.4.1.2 Named port connection

In this method, the connection between the ports can be done explicitly by linking the two names for each side of the connection, that is, the port declaration name from the module declaration can be linked to the name used in the instantiating module. The same example as above would be connected using the named port connection as follows. Note that the order of port connection is changed. However, it is recommended to keep the same order for reusability and readability.

```
lower U1 (
  .data(out1), // Order is changed,
               // connection is by name
  .addr (in1)  // only and not position
);
```

The two main advantages of this method are:

- It improves readability of the connections without having to refer to the port list of the instantiated module as the names from both sides are explicitly specified.

- The order of port connections is not relevant anymore since they are explicitly connected.

Note that the two types of module port connections *cannot* be mixed,, that is, all the connections to the ports of a particular module instance shall be either by order or by name. For example, the following is incorrect:

```
// gives a syntax error
lower U_wrong (in1, .addr(out1));
```

1.4.1.3 Implicit .* port connection

This is a feature available from SystemVerilog only. A new construct of specifying ".*" during module instantiation implicitly connects the ports of the instantiated module with the wires in the instantiating module. The precondition being the fact that the names and sizes need to be matched exactly. For example, in the following code, the `upper` module instantiates two `lower` modules. `U1` and `U2` The ".*" is equivalent to specifying three connections of `in1`, `in2`, and `in3` between the `lower` and `upper` modules.

```
module lower (in1, in2, in3, out1, out2);

input  [7:0] in1, in2, in3;
output [7:0] out1, out2;

assign out1 = in1 & in2;
assign out2 = in1 | in3;

endmodule // lower

module upper (in1, in2, in3, u_out11, u_out12,
              u_out21, u_out22);
input  [7:0] in1, in2, in3;
output [7:0] u_out11, u_out12;
output [7:0] u_out21, u_out22;

wire [7:0] u_out11, u_out12;
wire [7:0] u_out21, u_out22;

// Instantiating lower
lower U1 (
   .*, // .* does .in1(in1), .in2(in2), .in3(in3)
  .out1 (u_out11),
  .out2 (u_out12)
);

// Instantiating lower
lower U2 (
   .*, // .* does .in1(in1), .in2(in2), .in3(in3)
  .out1 (u_out21),
  .out2 (u_out22)
);

endmodule // upper
```

The synthesized logic of the above code instantiates the two lower modules, `U1` and `U2`. The correct port connections are also established for the ports `in1, in2,` and `in3`.

The advantage of the above method is that there is less chance of errors during instantiation, and it avoids repetition of names that implicitly match. Wherever exceptions and deviations exist, it needs to be explicitly specified. In the above, the connection to `u_out11, u_out12, u_out21,` and `u_out22` were made explicit.

The issue in the above method is that the user will not be able to physically "see" the connections.

1.4.1.4 Implicit .name port connection

This is a feature available from SystemVerilog only. A new construct of specifying the port name only once with the ".name" convention, where the "name" is the port name. This avoids specifying the port name twice when the port name and signal name are the same. The instance port name and size should match the connecting variable port name and size during module instantiation. In the following example, the ports `in1`, `in2` and `in3` of both the instances of lower module don't have any connecting variable port name.

```
module lower (in1, in2, in3, out1, out2);

input  [7:0] in1, in2, in3;
output [7:0] out1, out2;

assign out1 = in1 & in2;
assign out2 = in1 | in3;

endmodule

module upper (in1, in2, in3, u_out11, u_out12, u_out21,
u_out22);
input  [7:0] in1, in2, in3;
output [7:0] u_out11, u_out12;
output [7:0] u_out21, u_out22;

wire [7:0] u_out11, u_out12;
wire [7:0] u_out21, u_out22;

// Instantiating lower with out2 floating
lower U1 (
          .in1 , // no variable port name to these
          .in2 , // instance port names
          .in3 ,
          .out1 (u_out11) ,
          .out2 (u_out12)
         );

// Instantiating lower with out2 floating
lower U2 (
          .in1 , // no variable port names to
          .in2 , // these instance port names
          .in3 ,
          .out1 (u_out21) ,
          .out2 (u_out22)
         );

endmodule
```

The synthesized logic of the above code instantiates the two lower modules, `U1` and `U2`. The correct port connections are also established for the ports `in1, in2,` and `in3`.

The advantage of the above method is that there is less chance of errors during instantiation, and it avoids repetition of names that implicitly match. Wherever exceptions and deviations exist, it needs to be explicitly specified. In the above, the connection to `u_out11, u_out12, u_out21,` and `u_out22` were made explicit.

1.4.1.5 Interface port connection

SystemVerilog has introduced a construct ***interface,*** which basically encapsulates a bundle of nets and variables into one group. When there are numerous ports that need to be connected to each other, it is easier to make the connections through the ***interface*** construct. This helps create less verbose and more maintainable code by grouping all common connections in just one place. Any future changes to the interfaces can be modified in the ***interface*** definition, and this will propagate to all the instances where this is being used. The above example is illustrated using the ***interface*** construct as follows:

```
interface basic_con;
wire [7:0] in1, in2, in3; // bi-dir wires
endinterface : basic_con

module lower (basic_con all_ins, // all inputs
              output [7:0] out1, out2);

assign out1 = all_ins.in1 & all_ins.in2;
assign out2 = all_ins.in1 | all_ins.in3;

endmodule

module upper (in1, in2, in3, u_out11, u_out12,
              u_out21, u_out22);
input  [7:0] in1, in2, in3;
output [7:0] u_out11, u_out12;
output [7:0] u_out21, u_out22;

wire [7:0] u_out11, u_out12;
wire [7:0] u_out21, u_out22;

basic_con top_ins();

assign top_ins.in1 = in1;
assign top_ins.in2 = in2;
assign top_ins.in3 = in3;

// Instantiating lower
lower U1 (
// top_ins does .in1(in1), .in2(in2), .in3(in3)
  top_ins,
  u_out11,
  u_out12
);

// Instantiating lower
lower U2 (
// top_ins does .in1(in1), .in2(in2), .in3(in3)
  top_ins,
  u_out21,
  u_out22
);
```

```
endmodule
```

In the above example, the `top_ins` is the interface instantiation that sufficed the purpose of specifying the port connection of the in1 to in3 ports. Some of the salient points of the above example are:

- The above example could also be extended to connect inter-module connections, that is, connections to-from `U1` and `U2`.
- The ***interface*** specification and the explicit port connections could be mixed during one instantiation itself.

1.4.2 Can there be full or partial no-connects to a multi-bit port of a module during its instantiation?

No. There cannot be full or partial no-connects to a multi-bit port of a module during instantiation. For example, the following instantiation with an intermediate bit left to float is illegal, and gives a syntax error:

```
// Instantiating lower with some port bits
// unconnected
lower U1 (
  .in1(u_in1),

 // bit 6 for in2 is floating in 8 bit in2
  .in2({u_in2[7], ,u_in2[5:0]}), // Error

// bits [5:3] for out1 are unconnected in 8 bit
// out1
    .out1({u_out1[7:6], , u_out1[2:0]}), // Error
    .out2(u_out2)
);
```

In the case where there is a genuine situation to not connect a particular output, then it must be connected to an unused ***wire,*** and continue the concatenation with the appropriate bits to be connected. For example, in the above situation, the following two additional declarations, and the connections shown following it is a legal syntax:

```
wire unused1;
wire [2:0] unused2;
```

```
// Instantiating lower with out2 floating
lower U1 (
   .in1(u_in1),
   .in2({u_in2[7],unused1,u_in2[5:0]}),
   .out1({u_out1[7:5],unused2[2:0],u_out1[1:0]}),
   .out2(u_out2)
);
```

Note that a floating input or an unused wire on an output will cause a “z” propagation into the logic. The outputs will drive values onto the unused wires, but these wires do not fanout to other logic, and will be optimized away by synthesis tools.

1.4.3 What happens to the logic after synthesis, that is driving an unconnected output port that is left open (, that is, no-connect) during its module instantiation?

An unconnected output port in simulation will drive a value, but this value does not propagate to any other logic. In synthesis, the cone of any combinatorial logic that drives the unconnected output will get optimized away during boundary optimisation, that is, optimization by synthesis tools across hierarchical boundaries.

In the module `lower1` is instantiated into an `upper1` module, and the same pins are connected all the way to the top level. When this code is synthesized, it will produce the logic as shown in figure 1-1.

```
module lower1 (in1, in2, out1, out2);
input  in1, in2;
output out1, out2;
assign out1 = in1 & in2 ;
assign out2 = in1 | in2 ;
endmodule

module upper1 (u_in1, u_in2,
               u_out1, u_out2);
input  u_in1, u_in2;
output u_out1, u_out2;
lower1 U1 (
   .in1 (u_in1),
   .in2 (u_in2),
   .out1 (u_out1),
   .out2 (u_out2)
);
endmodule
```

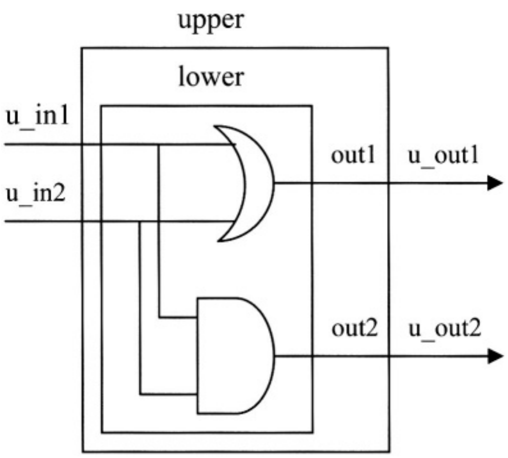

Figure 1-1. No unconnected ports

When `out1` is left unconnected during the instantiation of the lower module, (this port is not going all the way to the top level of `u_out 1`) as shown in this figure, then the logic gets optimized with only the AND gate remaining, and the OR gate getting optimized away.

Similarly, when `out2` is left unconnected during the instantiation of the lower module, the OR gate remains driving `out1` all the way to the top level, and the AND gate gets optimized away.

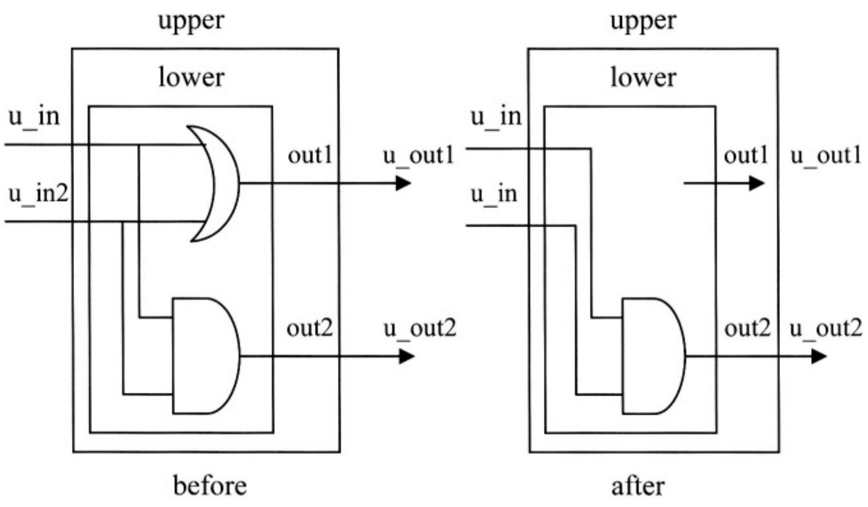

Figure 1-2. Gate eating behind an unconnected output port

1.4.4 What value is sampled by the logic from an input port that is left open (that is, no-connect) during its module instantiation?

By default, an unconnected input port is a floating port, and hence shows "z" during simulation. The logic following it will also propagate the "z", until gated off by an AND gate. The following figure shows the `in1` floating in lower instantiation.

Since `in1` was used as logic input to both the gates, and is no more driving both of them, the logic gets optimized and simplified into a simple wire connection between `in2` and `out2`. This connection still maintains the AND'ing logic required between these two ports, as per its design.

During synthesis, it is recommended to remove the unconnected ports using the synthesis tool commands, as it could potentially be undesirable during back-end processing.

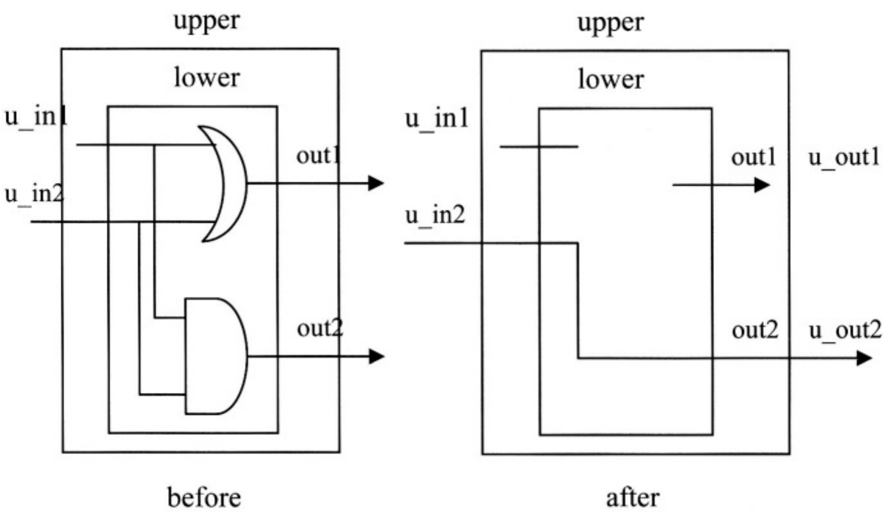

Figure 1-3. When one of the inputs is floating

The default value of z for unconnected input ports can be changed using the compiler directives:

```
`unconnected_drive pull0
```
and
```
`unconnected_drive pull1
```

The first directive causes all unconnected input ports to be pulled down to a logic 0. The second directive causes all unconnected input ports to be pulled up to logic 1. The effect of the `` `unconnected_drive `` directives can be turned off with the compiler directive `` `unconnected_drive ``. For example:

```
`unconnected_drive pull1
module ttl_74ls74 (clk, d, rst, pre, q, qb);
input clk, d, rst, pre; // will pull up if left
                        // unconnected
output q, qb;
...
endmodule
`nounconnected_drive // for the rest of the code
```

1.4.5 How is the connectivity established in Verilog when connecting wires of different widths?

When connecting wires or ports of different widths, the connections are right-justified, that is, the rightmost bit on the RHS gets connected to the rightmost bit of the LHS and so on, until the MSB of either of the net is reached. For example,

```
wire [7:0] net1;
wire [3:0] net2;

assign net1 = net2;
// implicitly net1[3:0] are connected to
// net2[3:0] and net1[7:4] is left floating

assign net2 = net1;
// The wires net1[3:0] are still connected to
// net2[3:0]
```

Note, however, that some simulation and synthesis tools will give a Warning when connecting nets or ports of dissimilar widths.

1.4.6 Can I use a Verilog *function* to define the width of a multi-bit port, *wire*, or *reg* type?

The width elements of ports, ***wire*** or ***reg*** declarations require a constant in both MSB and LSB. Before Verilog 2001, it is a syntax error to specify a ***function*** call to evaluate the value of these widths. For example, the following code is erroneous before Verilog 2001 version.

```
  reg [get_high(val1, val2) : get_low(val3, val4)] reg1;
```

In the above example, `get_high` and `get_low` are both ***function*** calls of evaluating a constant result for MSB and LSB respectively.

However, Verilog-2001 allows the use of a ***function*** call to evaluate the MSB or LSB of a width declaration.

SUMMARY

The chapter discussed a few basic questions on the usage of Verilog constructs during assignments and usage in ***task***, ***function***, port, and ***parameter.*** The chapter discusses the different approach of ***parameter*** and port specifications. A few SystemVerilog enhancements to the ***task*** and ***function*** have also been discussed. The next chapter discusses how the Verilog constructs are useful under the synthesis context.

Chapter 2

RTL DESIGN

INTRODUCTION

The chapter aims to address issues of the Verilog HDL that pertain to RTL design and logic synthesis. The focus is, in particular, on questions of logic inferences during synthesis, and static timing implications. The chapter concludes with explorations of power and DFT issues.

2.1 Assignments

This section discusses how the different assignments in Verilog are done, and what their implications are. The logic inferences of these different assignments are also discussed in this section.

2.1.1 What logic is inferred when there are multiple *assign* statements targeting the same *wire*?

It is illegal to specify multiple ***assign*** statements to the same ***wire*** in a synthesizable code that will become an output port of the module. The synthesis tools give a syntax error that a net is being driven by more than one source. For example, the following is illegal:

```
wire tmp;
assign tmp = in1 & in2; // only one type of
                         // output assignment is
assign tmp = in1 | in2; // legal for synthesis
```

However, it is legal to drive a three-state wire by multiple ***assign*** statements, as shown in the following example:

```
input enable1, enable2;
wire tmp;
assign tmp = (enable1 == 1'b1) ?
             (in1 & in2) : 1'bz;
assign tmp = (enable2 == 1'b1) ?
             (in3 | in4) : 1'bz;
```

2.1.2 What do conditional assignments get inferred into?

Conditionals in a continuous assignment are specified through the "*?:*" operator. Conditionals get inferred into a multiplexor. For example, the following is the code for a simple multiplexor:

```
wire wire1;
assign wire1 = (sel == 1'b1) ? a : b;
```

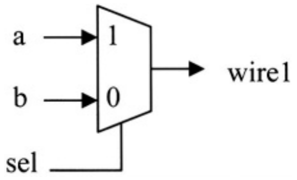

Figure 2-1. Conditionals infer into a multiplexor

2.1.3 What is the logic that gets synthesized when conditional operators in a single continuous assignment are nested?

Conditional operators in a single continuous assignment can be nested as shown in the following example. The logic gets elaborated into a tree of multiplexors.

```
input sel1, sel2, sel3, in1, in2, in3, in4;
output out1;
assign out1 = (sel1 == 1'b1) ? in1 :
              (sel2 == 1'b1) ? in2 :
              (sel3 == 1'b1) ? in3 : in4;
```

In the multiplexor units shown, it follows the logic that when `sel` is high, the output `Z` selects `A`, else selects `B`.

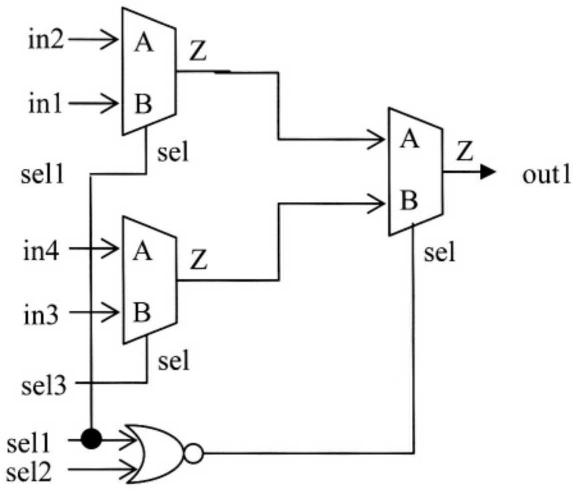

Figure 2-2. Tree of multiplexors inferred from nested conditionals

2.1.4 What value is inferred when multiple procedural assignments made to the same *reg* variable in an *always* block?

When there are multiple nonblocking assignments made to the same ***reg*** variable in a sequential ***always*** block, then the *last* assignment is picked up for logic synthesis. For example,

```
module lower (clk, in1, in2, out2);
  input  clk, in1, in2;
  output out2;
  reg tmp;
  always @(posedge clk) begin
    tmp <= (in1 ^ in2);
    tmp <= (in1 & in2);
    tmp <= (in1 | in2);
  end

assign out2 = tmp;
endmodule
```

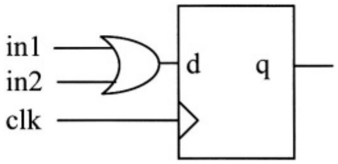

Figure 2-3. Multiple assignments to the same reg variable

In the example just shown, it is the OR logic that is the last assignment. Hence, the logic synthesized was indeed the OR gate. Had the last assignment been the "&" operator, it would have synthesized an AND gate.

Note that the optimised synthesis results match the simulation behaviour. The IEEE Verilog standard defines that nonblocking assignments in a ***begin...end*** will be assigned in the order listed. Hence, in simulation only, the value of the last assignment is seen.

Note, also, that the rules discussed and shown in this section apply when the variable on the LHS is not used on the RHS of subsequent assignments. The behaviour and synthesis implication for when a variable is used on both the LHS and RHS is discussed in the next FAQ.

The same would be the case for a combinatorial always block, too. For example,

```
always @(in1, in2) begin
  tmp = (in1 & in2);
     tmp = (in1 ^ in2);
  tmp = (in1 | in2);  // The final logic picked
                      // up is the OR gate
end
```

Since multiple assignments to the same variable is legal, the user has to keep track of the statements, as to what is the final assignment required. If only one among the multiple assignments was to be selected, it would typically be in an ***if-else*** tree or a ***case*** statement. For example, the above ***always*** block would be represented typically as follows, in which case only one unique assignment is executed at each clock cycle.

```
always @(posedge clk) begin
  if (sel1)
```

```
    tmp <= (in1 | in2);
  else if (sel2)
    tmp <= (in1 & in2);
  else
    tmp <= (in1 ^ in2);
end
```

In the above example, there is no ambiguity as to which statement gets selected, as the branching controls are clearly defined.

2.1.5 Why should a nonblocking assignment be used for sequential logic, and what would happen if a blocking assignment were used? Compare it with the same code in a combinatorial block.

As discussed in chapter 1, the main difference between the blocking and nonblocking assignment is that, in the blocking assignment, the RHS immediately gets assigned to the LHS, whereas for the nonblocking assignment, the assignment to the LHS is scheduled after the RHS is evaluated.

The following illustrate the different scenarios of using blocking and nonblocking in a sequential code.

2.1.5.1 Using blocking statements in a sequential logic

The following is an example of a Verilog module in which the blocking assignments have been used in the sequential block.

```
module reg_test (clk, in1, out1);
input clk,in1;
output out1;

reg reg1, reg2, reg3, out1;

always @(posedge clk) begin
  reg1 = in1;
  reg2 = reg1;
  reg3 = reg2;
  out1 = reg3;
end
endmodule
```

In the above example, the assignments to the `reg1`, `reg2`, `reg3`, `out1` have been made as blocking assignments. The synthesized result is a single FF, with the `d` input of `in1`, and `q` output of `reg3`, as shown in the following figure:

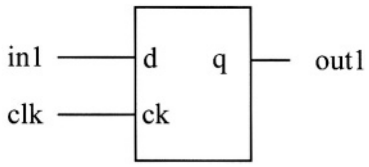

Figure 2-4. Logic inference with blocking assignments in sequential block

This is because the intermediate results between `in1` and `out1` were stored in `reg1`, `reg2`, and `reg3` in a blocking format. As a result, the evaluation of the final result to out1 didn't require waiting for all the events of the RHS to be completed. Rather, they were immediately assigned to the LHS in the order specified. Observe that the signals `reg1`, `reg2`, and `reg3` have been optimised away by synthesis.

2.1.5.2 Using nonblocking statements in a sequential logic

The following illustration of code uses the nonblocking assignments in a sequential block:

```
module reg_test (clk, in1, out1);
input clk,in1;
output out1;

reg reg1, reg2, reg3, out1;
always @(posedge clk) begin
  reg1 <= in1;
  reg2 <= reg1;
  reg3 <= reg2;
  out1 <= reg3;
end
endmodule
```

In the above example, the assignments to the `reg1`, `reg2`, `reg3`, `out1` have been made as nonblocking assignments. The synthesized result is the inference of as many FFs as specified in the always block [in this case, 4 FFs].

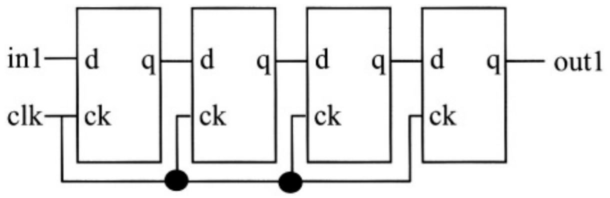

Figure 2-5. Using nonblocking assignments in sequential logic

This is because the intermediate results between `in1` and `out1` were stored in `reg1`, `reg2`, and `reg3` in a nonblocking format. As a result, the evaluation of the result to each individual ***reg*** required waiting for all the events of the RHS to be completed. In this case, it was the output of the previous register controlled by the `clk` event. As a result, the output is a shift register.

2.1.5.3 Using blocking statements in a combinatorial logic

The following example illustrates the use of blocking statements in combinatorial logic:

```
module reg_test (clk, in1, out1);
input clk,in1;
output out1;

reg reg1, reg2, reg3, out1;

always @(in1) begin
  reg1 = in1;
  reg2 = reg1;
  reg3 = reg2;
  out1 = reg3;
end
endmodule
```

In the above example, the blocking assignments are made in a combinatorial block. Note the absence of ***posedge*** and "<=", being replaced with “=”, in the assignments. The logic synthesized out of this is a simple wire between `in1` to `out1`.

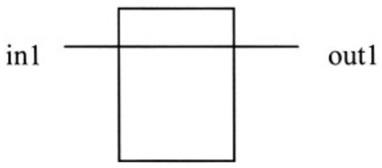

Figure 2-6. Blocking statements in combinatorial block

This is because all the assignments have been immediate, and there is no event to wait upon.

2.2 Tasks and Functions

Tasks and functions are primarily constructs that help in reusability of code that is being used in multiple places. Similar to the advantages seen in software programming, ***tasks*** and ***functions*** help in grouping statements with a particular intent in one code segment, and, hence, helps in better readability and maintenance.

2.2.1 What does the logic in a function get synthesized into? What are the area and timing implications of calling functions in RTL?

Since a ***function*** does not have any construct in it that advances time, a ***function*** basically infers combinatorial logic. If the logic falls into the critical path of a design, it is important to write the ***function*** in a timing optimal fashion.

For example, the following ***function*** does an arithmetic operation using two inputs and a control. Its result is used in another expression, that calls the ***function.***

```
module lower (in1, in2, out1, out2, out3, out4);
input  [1:0] in1, in2;
output [1:0] out1, out2, out3, out4;
wire   [1:0] out1, out2, out3, out4;

function [1:0] arith;  // declared in Verilog 1995
```

```
  input [1:0] in1;     // format with each input
  input [1:0] in2;     // declared separately after
  input [1:0] operation;  // the function
  begin
    case (operation)
      2'b00 : arith = in1 & in2 ; // AND gate
      2'b01 : arith = in1 | in2 ; // OR gate
      2'b10 : arith = in1 ^ in2 ; // XOR gate
      2'b11 : arith = in1 % in2 ; // mod operator
      default : arith = in1 & in2;
    endcase
  end
endfunction

assign out1 = arith(in1, in2, 0); // AND gate
assign out2 = arith(in1, in2, 1); // OR gate
assign out3 = arith(in1, in2, 2); // XOR gate
assign out4 = arith(in1, in2, 3); // mod operator

endmodule
```

Whether the repeated calls to a ***function*** replicate the logic within the ***function*** or it multiplexes the logic within ***function***, depends upon the path where the ***function*** is used. If the calls to a ***function*** are used in different paths, the logic gets replicated. In the above example, all the outputs had different use of the same ***function***, and, hence, independent logic for each function call implemented different logic. If `out1` and `out1` both required the OR gate functionality, then it would use the common logic of the two ***function*** calls for both the outputs, that is, in effect, the `out1` and `out2` would be connected to the same OR gate. However, any constant propagation techniques (see area optimisation techniques later this chapter for what constant propagation is) used within the ***function*** could influence the area.

Note that the above ***function*** call can be declared in the Verilog-2001 format, with the keyword ***input*** being part of ***function*** declaration, as follows:

```
function [1:0] arith     // no semicolon here
  (input [1:0] in1, in2, // the inputs are now part
   input [1:0] operation // of the function decl
  );// multiple inputs like in1 and in2 in one decl
```

Just like any other combinatorial logic, when the endpoint of the ***function*** is used as a D input to the flip-flop, then the ***function*** gets used to synthesize the sequential logic, too. For example, in the above code, the output out1 was a combinatorial output. If it is made a registered output, then the ***function*** output is used to derive the flip-flop, as illustrated in the following example:

```
reg [1:0] out1; // instead of wire

always @(posedge clk or negedge reset)
if (!reset) begin
  out1 <= 0;
end else begin
  out1 <= arith(in1, in2, 0); // function call
end
```

2.2.2 What are a few important considerations while writing a Verilog *function*?

The following are a few considerations while writing a Verilog ***function***:

- Local variables within a ***function*** and the function return value *should* be assigned values each time the ***function*** is called. Non initialization will cause a latch to be formed, as these variables are assigned every time upon entry of the ***function***. For example, the ***if*** condition within the following example does not have an ***else*** clause. Because the ***function*** is static in simulation, it will behave as latched logic. That is, if `sel` is false, the function will return the value of its previous call, as if the result were latched. Synthesis, however, still does not infer a latch. It simply infers a gated ***function***.

```
module lower (in1, in2, sel, out1, out2);
input in1, in2, sel;
output out1, out2;

wire out1, out2;

function bad_latch;
input in1, sel;

begin
```

```
  if (sel)
    bad_latch = in1;
  end

  endfunction

  assign out1 = bad_latch(in1, sel);
  assign out2 = bad_latch(in2, sel);

  endmodule // lower
```

In the above ***function*** calls, there are no variables to be initialized, and the logic inferred is the gating ***function***, as illustrated in this figure:

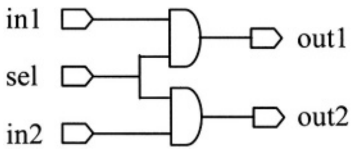

Figure 2-7. Function variables need to be assigned

- Ensure that the width of the return value from a ***function*** is specified fully, else it will end up with a default of one bit. For example:

```
module lower (in1, in2, all_outs);
input  [1:0] in1, in2;
output [7:0] all_outs;

wire [7:0] all_outs;

function arith;   // should have been
                  // function [7:0] arith
  input [1:0] in1;
  input [1:0] in2;
  reg [1:0] out1, out2, out3, out4;
  begin
    out1 = in1 & in2 ;
    out2 = in1 | in2 ;
    out3 = in1 ^ in2 ;
    out4 = in1 % in2 ;
    arith = {out1, out2, out3, out4};
  end
endfunction

assign all_outs = arith(in1, in2);

endmodule
```

In the above example, the desired output was actually 8 bits, but since the width of [7:0] was not specified between the keyword ***function*** and the function-name, the value returned by the ***function*** call is only the last bit, that is, bit [0] of the actual intended result.

- Functions are basically used to synthesize only combinatorial logic, however, the end result of this ***function*** can be used as a data input to the flip-flops, too.

- Functions should not include the delay(#) or event control (@, wait) statements.

- Functions may call other functions, but not other tasks.

- A ***function*** returns a value when it is called. For more than one return item, there are two ways to deal with it. Before SystemVerilog, this could be achieved by concatenating the multiple values into the single return. In the previous example, the output `arith` is a concatenation of multiple outputs that need to be driven by a single ***function*** call. The desired output fields from the result are then derived to drive the required signals. For example,

```
module lower (in1, in2, out1, out2, out3, out4);
input  [1:0] in1, in2;
output [1:0] out1, out2, out3, out4;

wire [1:0] out1, out2, out3, out4;
wire [7:0] all_outs;
function [7:0] arith;
  input [1:0] in1;
  input [1:0] in2;
  reg [1:0] out1, out2, out3, out4;
  begin
    out1 = in1 & in2 ;
    out2 = in1 | in2 ;
    out3 = in1 ^ in2 ;
    out4 = in1 % in2 ;
    arith = {out1, out2, out3, out4};
  end
endfunction

assign all_outs = arith(in1, in2);
assign out1 = all_outs[7:6];
assign out2 = all_outs[5:4];
assign out3 = all_outs[3:2];
assign out4 = all_outs[1:0];

endmodule
```

In the above example, the different outputs `out1` to `out4` were all concatenated and assigned to the function name. The different fields can be then extracted out of the wire to which the ***function*** drives.

With SystemVerilog, it is possible to have a formal ***output*** and ***inout*** declaration. The same example in SystemVerilog is as follows:

```
module funct_output (in1, in2, out1, out2,
                     out3, out4);

input  [1:0] in1, in2;
output [1:0] out1, out2, out3, out4;

reg  [1:0] out1, out2, out3, out4;

// void (that is, doesn't return anything)
function void arith;
  input  [1:0] in1, in2;
  output [1:0] out1, out2, out3, out4;
  begin
    out1 = in1 & in2 ;
    out2 = in1 | in2 ;
    out3 = in1 ^ in2 ;
    out4 = in1 % in2 ;
  end
endfunction

always_comb // SystemVerilog construct
```

```
    begin
      arith (in1, in2, out1, out2, out3, out4);
    end

  endmodule
```

- Parameters and integers can be declared within a ***function***, but they become local only to that function, and cannot be used outside the scope of the ***function***. In the following example, the `width1` ***parameter*** defined within the ***function*** `double_width` is not visible outside its scope for the ***$display*** statement that follows later.

```
parameter width = 32;

function integer double_width;
  input integer in_width;
  parameter width1 = 64;
  double_width = in_width * 2;
endfunction

initial begin
  $display("width1 = %0d",width1); // syntax error
end
```

2.2.3 What does the logic in a task get synthesized into? Explain with an example.

Although it is legal to have time advancing or controlling constructs like @ within a ***task***, it works only for simulations. The synthesis tools **<u>ignore</u>** all timing constructs within a ***task***. Hence, a simulation and synthesis mismatch can occur if the functionality depends upon presence of timing control constructs within a ***task***. Thus, a ***task*** can be used to synthesize basic combinatorial logic. However, if the destination of the ***task*** call is a storage element used within a sequential block, then a sequential element gets synthesized. Whether the logic within the ***task*** will keep replicating whenever it is called or reused depends upon the path where the task is used. If the task call is for independent paths that can be used concurrently, then independent logic will be synthesized for each path, and the area grows linear to the number of tasks called. If the ***task*** is used among common paths, then the logic in its inputs or outputs could be reused, depending upon the path from which the ***task*** is called.

The following is an example of a combinatorial task, namely, `combtask`, which performs the task of unary OR'ing the input `in1`, and producing it in the output of the task. Note that an intermediate ***reg*** declaration of `int_out1` and `int_out2` was required, because the output of a ***task*** can be received **only by a *reg* and not a *wire*.**

```
module comb_task (in1, in2, out1, out2);
input [3:0] in1, in2;
output out1, out2;

reg int_out1, int_out2;

task modify_value;
  input [3:0] value;
  output int_val;
  reg int_val;
begin
  int_val = (|(value)); // Combinatorial
                        // operation in a task
end
endtask

always @(in1) begin
  modify_value(in1, int_out1);
end

always @(in2) begin
  modify_value(in2, int_out2);
end

assign out1 = int_out1;
assign out2 = int_out2;

endmodule
```

Many synthesis tools give a compilation error if sequential constructs are present *within* a task.

2.2.4 What are the differences between using a task, and defining a module for implementing reusable logic?

We have already seen in the previous question, that a ***task*** can be used to call same logic multiple times. Similarly, a ***module*** can also be defined, and the logic within it will get replicated as many times as it is instantiated. The following table summarizes the differences between the two approaches:

Table 2-1. Table summarizing the difference between task and module

task	**module**
A ***task*** cannot instantiate a ***module*** within it	Fundamentally ***task***s can be called only within a module
The logic of a task cannot be identified as a block to be moved around during floorplanning. It is a part of the sea-of-gates	A module instantiation has a hierarchy that can be fully identified and placed as a block during floorplanning
Prior to the advent of Verilog 2001, tasks in Verilog are not re-entrant. Therefore, if a task uses internal local variables, it could be multiply invoked in overlapping time-domains	By definition, modules can be instantiated multiple times. Each instance will carry its own context, including all of its internal registers and other variables. Therefore, processes within these instances are inherently concurrent among themselves and also across instances

2.2.5 Can tasks and functions be declared external to the scope of module-endmodule?

Yes. With SystemVerilog, it is possible to declare the ***task*** and ***function*** definitions external to the scope of ***module-endmodule.*** This is not possible with Verilog-1995 or Verilog-2001, and will give a compilation error. For example, in the following code, the ***task*** `modify_value` is declared outside the scope of the ***module-endmodule.***

```
task modify_value;        // this task is defined
  input [3:0] value;      // outside the scope of
  output int_val;         // module-endmodule
  reg int_val;
begin
  int_val = (|(value)); // Combinatorial operation
                          // in a task
end
```

```
endtask

module ext_task (in1, out1);
input [3:0] in1;
output out1;

reg int_out1;

always @(in1) begin
  modify_value(in1, int_out1);
end

assign out1 = int_out1;

endmodule // ext_task
```

Similarly, with SystemVerilog, a ***function-endfunction*** can also be declared outside the scope of ***module-endmodule*** within the same file. If these contents are defined in a separate file, it needs to be part of the same compilation command.

2.3 Storage Elements

There are primarily two kinds of storage elements inferred in logic synthesis, that is, Flip-flops and latches. This section describes the implementation and comparison between the two elements.

2.3.1 Summary of RTL templates for different flip-flops types

The storage element of flip-flop or latch inferred from RTL depends upon the style in which it is written. The following is a quick summary of a few templates of different register and latch inferences. In the flip-flop templates, they infer a positive edge triggered flip-flop. If the keyword `posedge clk` is replaced with `negedge clk`, then a negative edge triggered flip-flop is inferred.

1. Simple D Flip-flop

Positive edge triggered, no set or reset, value of Q is unknown at power on

```
module dff (clk, d, q);
```

```
input clk, d;
output q;

reg q;
always @(posedge clk) begin
  q <= d;
end
endmodule
```

In SystemVerilog, the same code would be implemented with ***always_ff*** in place of the ***always*** keyword, as follows:

```
always_ff @(posedge clk) begin
  q <= d;
end
```

The advantage of ***always_ff*** over ***always*** is that, ***always_ff*** indicates that the designers intent is to model clocked sequential logic. Software tools can then verify that the blocks sensitivity list and functionality correctly represent the type of logic intended.

2. Asynchronous set FF

Positive edge triggered, active high asynchronous set

```
module asff (clk, d, set, q);
input clk, d, set;
output q;

reg q;
always @(posedge clk or posedge set) begin
  if (set)
    q <= 1'b1;
  else
    q <= d;
end
endmodule
```

Replacing the ***always*** keyword with ***always_ff*** above would implement the asynchronous FF in SystemVerilog.

3. Asynchronous reset FF

Positive edge triggered, active high asynchronous reset

```
module arff (clk, d, reset, q);
input clk, d, reset;
output q;

reg q;
always @(posedge clk or posedge reset) begin
  if (reset)
    q <= 1'b0;
  else
    q <= d;
end
endmodule
```

Replacing the ***always*** keyword with ***always_ff*** above would implement the asynchronous FF in SystemVerilog.

4. Asynchronous set and reset FF

Positive edge triggered, active high asynchronous set and reset

```
module arsff(clk, d, set, reset);
input clk, d, set, reset;
output q;

reg q;
always @(posedge clk or posedge set or
         posedge reset)
begin
  if (set)
    q <= 1'b1;
  else if (reset)
    q <= 1'b0;
  else
    q <= d;
end
endmodule
```

Replacing the ***always*** keyword with ***always_ff*** above would implement the asynchronous FF in SystemVerilog.

5. Synchronous set FF

Positive edge triggered, active high synchronous set

```
module ssff(clk, d, set, q);
input clk, d, set;
output q;

reg q;
always @(posedge clk) begin
  if (set)
    q <= 1'b1;
  else
    q <= d;
end
endmodule
```

Replacing the ***always*** keyword with ***always_ff*** above would implement the asynchronous FF in SystemVerilog.

6. Synchronous reset FF

Positive edge triggered, active high synchronous reset

```
module srff (clk, d, reset, q);
input clk, d, reset;
output q;

reg q;
always @(posedge clk) begin
  if (reset)
    q <= 1'b0;
  else
    q <= d;
end
endmodule
```

Replacing the ***always*** keyword with ***always_ff*** above would implement the asynchronous FF in SystemVerilog.

7. Synchronous set and reset FF

Positive edge triggered, active high synchronous set and reset

```
module ssrff (lk, d, set, reset, q);
input clk, d, set, reset;
output q;

reg q;
always @(posedge clk) begin
  if (set)
    q <= 1'b1;
  else if (reset)
    q <= 1'b0;
  else
    q <= d;
end
endmodule
```

Replacing the ***always*** keyword with ***always_ff*** above would implement the asynchronous FF in SystemVerilog.

2.3.2 Summary of RTL templates for different Latch types

1. Simple D Latch

```
module dl (sel, d, q);
input sel, d;
output q;

reg q;
always @(sel, d) begin
  if (sel)
    q <= d;
end           // Note the else clause is missing
endmodule
```

In Verilog-2001, the same latch can be implemented as:

```
always @(*) // note implicit sensitivity list
  if (sel)
```

```
    q <= d;
end
```

In SystemVerilog, the same latch can be implemented using the keyword ***always_latch***, as:

```
always_latch // no explicit sensitivity list
  if (sel)
    q <= d;
end
```

Note that it was not required to specify anything in the sensitivity list of the ***always_latch*** block, as this procedure determines its sensitivity automatically. One advantage of the ***always_latch*** keyword is that, it explicitly shows the designer intends to model a latch. Software tools can then check that the functionality within the procedural block correctly represents latched logic. Another important advantage of ***always_latch*** is that, it is automatically evaluated once at simulation time 0, even if the `sel` input did not change at time 0. This ensures that at the start of simulation, the latch output is correctly reflecting the latch inputs.

2. Asynchronous set latch

```
module asl (sel, d, set, q);
input sel, d, set;
output q;

reg q;
always @(sel, d, set) begin
  if (set)
    q = 1'b1;
  else if (sel)
    q = d;
end       // Note the final else clause is missing
endmodule
```

In SystemVerilog, the same always procedure above can be implemented using the ***always_latch***, instead of the ***always*** keyword, without any sensitivity list.

3. Asynchronous reset latch

```
module arl (sel, d, reset, q);
input sel, d, set;
output q;

reg q;
   always @(sel, d, reset) begin
  if (reset)
    q = 1'b0;
  else if (sel)
    q = d;
end           // Note the else clause is missing
endmodule
```

In SystemVerilog, the same ***always*** procedure above can be implemented using the ***always_latch,*** instead of ***always*** keyword, without any sensitivity list.

4. Asynchronous set and reset latch

```
module asrl (sel, d, set, reset, q);
input sel, d, set, reset;
output q;

reg q;
always @(sel, d, reset) begin
  if (reset)
    q = 1'b0;
else if (set)
    q = 1;
  else if (sel)
    q = d;
end       // Note the final else clause is missing
endmodule
```

In SystemVerilog, the same ***always*** procedure above can be implemented using the ***always_latch***, instead of ***always*** keyword, without any sensitivity list.

A few salient points to be noted in the above inferences:

- In asynchronous set or reset storage elements, the asynchronous input has higher priority than the data input (hence, it is in the top of the ***if-else*** tree). Therefore, when an asynchronous input and the data inputs arrive at the same time, the effect of the asynchronous input prevails at the output.
- All the above examples show a single bit implementation of the defined storage element. By increasing the bit width of the ***reg*** declaration, the number of flip-flops or latches will be equal to the width of the ***reg*** declaration. For example,
  `reg [3:0] out1;`
  This will create 4 of the `out1` flip-flops or latches.
- There need not be one always block for each flip-flop. Many flip-flops can be inferred within an ***always*** block. But the restriction in this approach is that **ALL** of the FF definitions within that ***always*** block will infer the same type of flip-flop as defined in the sensitivity list of the ***always*** block.
- Although normally all the bits of the storage elements are either set or reset, it is not uncommon to assign values of 1'b1 and 1'b0 to the different bits of the same register during the set or reset condition. For example, in this 4 bit FF, the reset values of the flops are 4'b1010.

```
module lower (in1, clk, reset, out1);
input  [3:0] in1;
input  clk, reset;
output [3:0] out1;

reg [3:0] out1;

always @(posedge clk or negedge reset)
begin
  if (!reset)
    out1 <= 4'b1010;
  else
    out1 <= in1;
end

endmodule
```

- A common coding practice is to use only nonblocking assignments for inferring flip-flops and latches, and only blocking for inferring combinatorial logic.
- Using SystemVerilog frees the user from specifying the elements of the sensitivity list for the latch inferences. It also ensures that the latch output values are correct at the start of simulation. These features would reduce the possibility of simulation and synthesis mismatches.

2.3.3 What are the considerations to be taken choosing between flop-flops vs. latches in a design?

Both latches and FFs have their relative advantages and disadvantages in their implications, as summarized in the table below:

Table 2-2. Consideration of latch and Flip-flop features for design choice

Latch	Flip-flop
Area of a latch is typically less than that of a Flip-flop	Area of a Flip-flop for same features is more than that of a latch
Consumes lesser power, due to lesser switching activity and lesser area	Power consumption is typically higher, due to the area and free running clock. Additional controls required to save power
Facilitates time borrowing or cycle stealing; Helps increase pipeline depth with lesser area.; Even if the path is longer than a clock cycle for a latch based pipeline, it is okay as long as it meets the next latch setup margin	Since the clock boundaries are rigid, the facility of time borrowing or cycle stealing doesn't exist with FFs. A negative slack cannot be propagated to the timing of the next stage in pipeline and hence must execute within a clock period
In multiple clock schemes, the clock edges must not be overlapping; It makes the logic design, vector generation for verification and clock tree synthesis difficult	Clock tree synthesis is less tedious in FF based designs. Since the stimulus needs to be stable before the setup time of the clock, the vector generation is relatively easier
With time borrowing* and cycle stealing, the operating frequency is higher than the slowest logic path	Due to rigid timing boundaries, the slowest path pretty much decides the operating frequency
Makes time budgeting and characterizing the interfaces tedious	The time budgeting is clearer and characterizing the interface is easier

(*) Time borrowing is a mechanism in which a latch based design takes advantage of the transparency between two back to back latches that are enabled in order to meet the propagation delay between the two latches. This is best illustrated by a simple analysis as follows:

Consider two latches L1 and L2. While both of them have the same clock frequency, the enables for L1 and L2 are opposite in polarity. The L1 is enabled in the high phase of the clock, while L2 is enabled in the low phase of the clock. This connection is shown in the following figure:

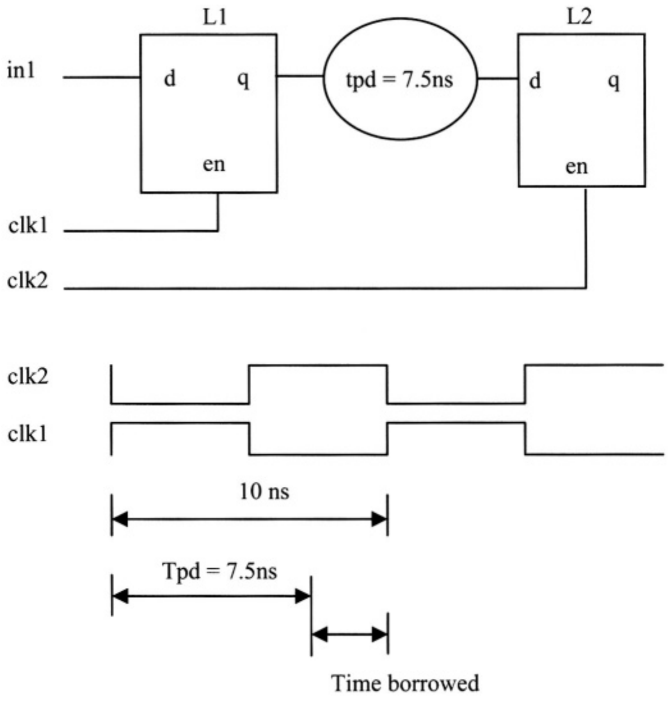

Figure 2-8. Illustration of time borrowing in latches

For the purpose of simplifying the analysis, the **d→q** delay or **en→q** delay of L1 and L2 is assumed to be 0ns. The propagation delay of the combinatorial logic is 7.5ns. If it were a flip-flop based design with the same rising edge clock in place of clk1 and clk2, this would be clearly a setup violation. However, in a latch based design above, since the delay through latch is 0ns, the input in1 is latched immediately at the output of L1, and begins to propagate. The propagation delay enters into the ON time of the second latch L2, and settles at some point during its ON time. The propagation delay has caused the logic to borrow time from the second latch, in order to settle its outputs, and hence is called time borrowing.

2.3.4 Which one is better, asynchronous or synchronous reset for the storage elements?

The following table summarizes the comparison between using synchronous and asynchronous reset logic for a design:

Table 2-3. Summary of differences between asynchronous and synchronous reset

Asynchronous reset	**Synchronous reset**
Reset signal is not a part of the data path, that is, not a part of logic for D input of the FF	Reset signal is part of the data path, that is, the D input of the FF
Effect of reset can happen anytime asynchronously	Effect of reset will happen only on the active edge of a clock
Doesn't depend upon the presence of an active clock signal	Depends upon the presence of the clock signal for the reset to happen
Asynchronous event is an overload, compared to synchronous reset in the cycle based simulators	Works well when using cycle based simulators
Not recommended for internally generated resets, due to glitches	For internally generated resets, synchronous approach is the best mechanism
Reset input from external sources can be prone to glitches, the final reset signal needs to be synchronized before applying it to all storage elements	Not prone to glitches from internal or external sources
Asynchronous reset input still needs the double FF synchronization to avoid race condition during de-assertion	The additional synchronization circuitry is not required as it is a part of the default synchronous logic requirement
Needs to meet only the minimum reset pulse width required for the FF	Reset pulse width has to be long enough to be sampled on an active clock edge

Asynchronous reset	**Synchronous reset**
Example code of an asynchronous low reset	Example code for a synchronous low reset
`always @(posedge clk or negedge reset)`	`always @(posedge clk)`
`begin`	`begin`
`if (!reset)`	`if (!reset)`
`out1 <= 0;`	`out2 <= 0;`
`else`	`else`
`out1 <= in1;`	`out2 <= in2;`
`end`	`end`

2.3.5 What logic gets synthesized when I use an *integer* instead of a *reg* variable as a storage element? Is use of *integer* recommended?

An ***integer*** can take the place of a ***reg*** as a storage element. An example to illustrate this is as follows:

```
module int_insteadof_reg (in1, clk, reset, out1);
input  [3:0] in1;
input  clk, reset;
output [3:0] out1;

integer int_tmp;
// reg [3:0] int_tmp;   // Normally we use this reg
// declaration

always @(posedge clk or negedge reset)
begin
  if (!reset)
    int_tmp <= 0;
  else
    int_tmp <= in1;
end

assign out1 = int_tmp;

endmodule
```

In this example, the variable `int_tmp` is defined as an ***integer***, instead of the ***reg*** that it would normally be (the ***reg*** declaration is commented in the example for illustration). Note that, although the default width of the ***integer*** declaration is 32 bits, the final result of the `int_tmp` registers synthesis yield is only 4 bits. This is because the optimiser in the synthesis tool removes the unnecessary higher order bits, in order to minimize the area.

Although the use of integer as shown above is a legal construct, it **is not recommended** for the synthesis of storage elements.

2.4 Flow-control Constructs

Verilog has primarily three kinds of flow control constructs, that is, ***case***, ***if-else*** and "***?:***" conditionals. The "***?:***" construct has already been discussed earlier FAQ 2.1.2 and 2.1.3. This section primarily illustrates the implementation details and questions about ***case*** and ***if-else*** constructs.

2.4.1 How do I choose between a *case* statement and a multi-way *if-else* statement?

Both ***case*** and ***if-else*** are flow control constructs. Functionally in simulation they yield similar results. While both these constructs get elaborated into combinatorial logic, the usage scenarios for these constructs are different.

A ***case*** statement is typically chosen for the following scenarios:

- When the conditionals are mutually exclusive and only *one variable* controls the flow in the case statement. The case variable itself could be a concatenation of different signals.
- To specify the various state transitions of a finite state machine
- Use of ***casex*** and ***casez*** allows use of x and z to represent don't-care bits in the control expression

A multi way ***if*** statement is typically chosen in the following scenarios:

- Synthesizing priority encoded logic
- When the conditionals are not mutually exclusive and more general in using **multiple expressions** for the condition expression.

The advantages of using the ***case*** over ***if-else*** is as follows:
- ***case*** statements are more readable than ***if-else***
- When used for state machines, there is a direct mapping between the state machine's "bubble diagram" and the ***case*** description.

In a ***case*** construct, if all the possible cases are not specified, and the ***default*** clause is missing, a latch is inferred. Likewise, for an ***if-else*** construct, if a final ***else*** clause is missing, a latch is inferred.

2.4.2 How do I avoid a priority encoder in an if-else tree?

An ***if-else*** tree may synthesize to a priority encoded logic. For example, the following code produces a priority encoder:

```
module priorityencoder (in0, in1, in2, in3, sel);

input  in0, in1, in2, in3;
output [1:0] sel;
reg [1:0] sel;

always @(in0, in1, in2, in3) begin
  sel = 2'b00;
  if (in0) sel = 2'b00;
  else if (in1) sel = 2'b01;
  else if (in2) sel = 2'b10;
  else if (in3) sel = 2'b11;
end

endmodule // priorityencoder
```

In simulation, the if-else-if series is evaluated in the order listed. If `in0` and `in1` were both true, the `in0` branch would be taken, because `in0` is evaluated first. Synthesis tools will create a priority encoded logic in this example, so that the logic generated will behave the same as the RTL simulation.

If a priority encoder is not the intention, the logic needs to be synthesized in parallel. The keyword ***unique*** that is introduced in the SystemVerilog can be used for this purpose. The ***unique*** keyword indicates that the order of decisions is not important. The ***if*** statement would be the same, with the ***unique*** keyword prepending the first ***if***, as follows:

```
  unique if (in0) sel = 2'b00;
  else if (in1) sel = 2'b01;
  else if (in2) sel = 2'b10;
  else if (in3) sel = 2'b11;
```

This would synthesize into a parallel logic, that is, a multiplexor.

The SystemVerilog standard requires that simulation (or other tools) report a Warning if they detect that more than one branch could be executed at the same time. In the preceeding example with ***unique if***, if both `in0` and `in1` were both true at the same time, a run-time Warning would be reported.

On a related note, SystemVerilog has also introduced the keyword ***priority,*** which functions opposite to ***unique***, by enforcing priority encoded logic. When the ***priority*** construct is used, it indicates that the order of decision making is important. If the ***unique*** statement in the above is replaced by ***priority,*** then the same priority select logic tree will be regenerated.

2.4.3 What are the differences between *if-else* and the ("*?:*") conditional operator?

The following table summarizes the differences between the two flow control constructs, that is, conditional "***?:***" and the ***if-else.***

Table 2-4. Summary of differences between the conditional "***?:***" and ***if-else*** operator

Conditional "*?:*" operator	***if-else***
Typically used in procedural or continuous assignments	Typically used within ***initial*** or ***always*** blocks

A TRUE and a FALSE expression is always required to be fully specified, that is, in the example:
 	
`assign a=(b == 0) ? c:d;`
 	
both expression `c` and `d` are required
	The ***else*** portion is optional in the ***if-else*** statement, in the following example:
	`if (en)`
	`  q = d;`
	`// else not necessarily`
	`// required`

Expressions to the left and right of the colon ":" can only be a single expression, that is, not a block of expressions within begin-end. For example, the following is a syntax error:
`a = (b==0)? c : begin d; e; end; // wrong`
	The expressions within the ***if-else*** can be block code enclosed within a begin-end. For example,
	`if (en) begin`
	`… // many expressions`
	`end else begin`
	`… // many expressions`
	`end`

While the "***?:***" operator is useful in specifying simple expressions, readability is an issue when it is deeply nested	***if-else*** is visually more readable code in all expressions, especially when it is nested

2.4.4 What is the importance of a *default* clause in a *case* construct?

The ***default*** clause in a ***case*** statement indicates that when all other cases are not met, then the flow can branch to the statements in the default clause.

This gives the synthesis tool an option to pick a branch when no other condition is satisfied. If the ***default*** clause is missing, the logic will have to remember what the output was earlier, and hence a latch will get synthesized. For example, the following ***case*** statement will generate a latch:

```
module default_latch (in1, in2, opcode, out1);
input  [1:0] in1, in2, opcode;
output [1:0] out1;

reg [1:0] out1;

always @(in1 or in2 or opcode) begin
  case (opcode)
    2'b00 : out1 = in1 & in2 ;
    2'b01 : out1 = in1 | in2 ;
    2'b10 : out1 = in1 ^ in2 ;
//  2'b11 : out1 = in1 % in2 ; // uncommenting
                               // either of these
                               // two lines will
// default : out1 = in1 & in2; // avoid a latch
  endcase
end

endmodule
```

In the above, with the two lines commented, a latch gets synthesized for out1 register. Un-commenting either the ***default*** clause or the last condition of 2'b11, or both, will result in the combinatorial logic of a multiplexor to be synthesized.

2.4.5 What is the difference between full_case and parallel_case synthesis directive?

The difference between full case and parallel case synthesis directives is summarized in the table below:

Table 2-5. Difference between full case and parallel case

full_case	**parallel_case**
Indicates that the case statement has been fully specified, and all unspecified case expressions can be optimized away	Indicates that all case items need to be evaluated in parallel and not infer any priority encoding logic
All control paths are specified explicitly or by using a default	There is no overlap among the case items
Helps avoid latches as all cases are fully specified	Results in multiplexor logic as a parallel logic
Although not recommended, the default clause can be avoided, and still not infer a latch	A priority encoder is NOT synthesized, as each path is unique

An example of a case statement that is full (and parallel) is shown below:
 	
`reg var1 [1:0];`
`always @(a or b or c) begin`
`  case (var1)`
`    2'b00 : out1 = a;`
`    2'b01 : out1 = b;`
`    2'b10 : out1 = c;`
`    2'b11 : out1 = a&b;`
`  endcase`
`end`
 	
Note that the ***default*** clause was not required here as it is fully specified (although having it is a good coding practice).
	An example of a case statement that is parallel (not full) is shown as follows:
	`reg var1 [2:0];`
	`always @(a or b or c) begin`
	`  case (var1)`
	`    3'b000 : out1 = a;`
	`    3'b001 : out1 = b;`
	`    3'b010 : out1 = c;`
	`// rest of the cases are //`
	`not defined`
	`  endcase`
	`end`
	
	Note that the above case **doesn't** have a ***default*** clause; but each branch is definitely **unique**, but all cases are not specified, that is, branches missing for 2,3,4,5,6,7. The `out1` register **will** get synthesized into a latch

2.4.6 What is the difference in implementation with sequential and combinatorial processes, when the final *else* clause in a multi-way *if-else* construct is missing?

The results are different, depending upon whether the ***if*** statement is a part of a sequential ***always*** block or a combinatorial ***always*** block.

In a combinatorial ***always*** block, when the final ***else*** clause in a multi-way ***if-else*** statement is missing, it will infer a latch. The latch is inferred because the register has to remember the value until it is reloaded again. For example,

```
reg latch1;
always @(sel, in1) begin
  if (sel)
    latch1 <= in1;
end
```

In a sequential ***always*** block, if the final ***else*** clause in a multi-way ***if-else*** statement is missing, it will still go ahead and infer the flip-flop, with the combinatorial inference of the logic in the D input of the flop. For example,

```
reg ff1;
always @(posedge clk or negedge reset) begin
  if (reset)
    ff1 <= 1'b0;
  else begin
    if (sel1)
      ff1 <= in1; // no else clause here
  end
end
```

The above code will infer logic, as shown below. The D input to the flop is now a simple gated function of the inputs.

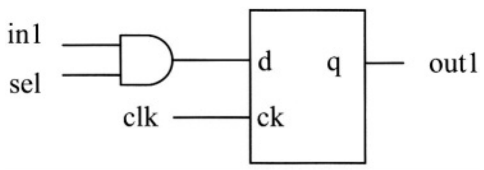

Figure 2-9. Logic inference of if statement without final else in a FF

2.4.7 What is the difference in using (== or !=) vs. (===or !==) in decision making of a flow control construct in a synthesizable code?

In Verilog, the (==) operator is called logical equality, and (!=) is called logical inequality operator. The (===) operator is called case equality, and (!==) is called case inequality. The following are the differences in using these constructs in synthesizable code.

Table 2-6. Differences between === and == operators

Use of == or != operators	**Use of === or !== operators**
These operators can be used in a synthesizable code	Cannot be used in a synthesizable code
If either of the operands have x or z value, the result is unknown	The operands will be compared, even if they have x and z values in the bits
If any of the operators is x or z, the logical result of comparison is **always** FALSE	The x and z bits will be used in comparison, and the logical result will be a TRUE or FALSE, based on actual comparison
Since the operands contain x and z, the result will be an x. Hence, the comparison can be non-deterministic	Since x and z are also used in comparison, the result of comparison will be Boolean 1 or 0. Hence the comparison can be deterministic

Example of using (== or !=) operators
`if (a == b)`
`  out1 = a & b;`
`else`
`  out1 = a | b;`
 	
If either `a` or `b` becomes `x` or `z`, the else clause will be executed and `out1` will be driven by OR gate
	Example of using (=== or !==) operators
	`if (a === b)`
	`  out1 = a & b;`
	`else`
	`  out1 = a | b;`
	
	If `a` and `b` are identical, even if they becomes `x` or `z`, the if clause will be executed and `out1` will be driven by AND gate

2.4.8 Explain the differences and advantages of *casex* and *casez* over the *case* statement?

The ***casex*** operator has to be used when both the high impedance value (z) and unknown (x) in any bit has to be treated as a don't-care during case comparisons. The ***casez*** operator treats the (z) operator as a don't-care during case comparisons.

In both cases, the bits which that are treated as don't-care will not be considered for comparison, that is, only bit values other than don't care bits are used in the comparison. The wildcard character "?" can be used in place of "z" for literal numbers.

The following is an example of a ***casex*** statement

```
input [2:0] in1;
reg [2:0] reg1;
casex (in1)
  3'b0x0 : out1 = a & b;  // same as conditions
                          // 3'b010, 3'b000
  3'bx10 : out1 = a | b;  // same as conditions
                          // 3'b110, 3'b010
  default : out1 = a ^ b; // for all other
                          // conditions
endcase
```

The same example, if written with an if-else tree, would look like:

```
// bit in1[1] is not considered at all
  if (!in1[2] & !in1[0]) out1 = (a & b);
// bit in1[2] is not considered
  else if (in1[1] & !in1[0]) out1 = (a | b);
// default clause
else out1 = (a ^ b);
```

Using ***casex*** or ***casez*** has the following coding advantages:

- it reduces the number of lines, especially if the number of bits had been more
- makes code look more clear and less cluttered
- Simplifies the optimization, as it is clear that the bits with x are to be ignored.

2.5 Finite State Machines

Finite State Machines or FSMs form an important part of the control logic in the designs. This section also discusses the differences among the various types of the FSM coding styles.

2.5.1 What are the differences between synchronous and asynchronous state machines?

Synchronous and asynchronous are two fundamental types of state machines. They differ in the following ways:

Table 2-7. Differences between asynchronous and synchronous state machines

Asynchronous state machines	**Synchronous state machines**
State transitions depend upon the order in which the input signals change	State transitions are controlled by a clock signal
State transitions happen after propagation delay of the state line	State transitions happen at intervals of the clock period
Delay lines act as memory elements	Edge triggered FFs or level sensitive latches act as storage elements
Output response time is not predictable	Output response time is predictable; will happen at clock period intervals

2.5.2 Illustrate the differences between Mealy and Moore state machines.

Both Mealy machine and Moore machine are two commonly used coding styles of state machines. The basic block diagram of these two state machines are shown as follows:

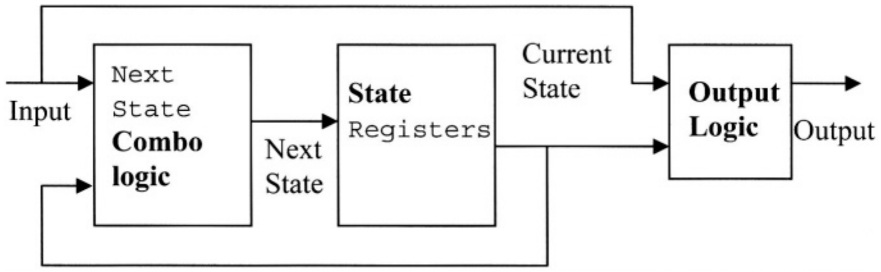

Figure 2-10. Block diagram of a Mealy machine

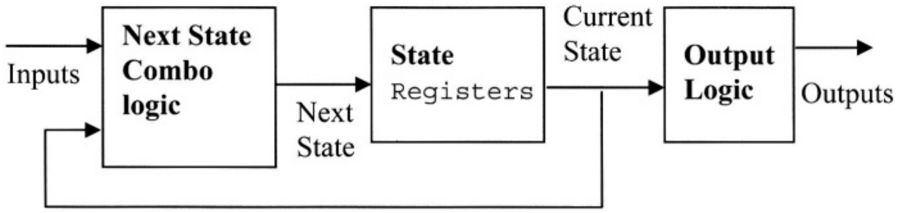

Figure 2-11. Block diagram of a Moore machine

The state machines differ in the following ways:

Table 2-8. Differences between Mealy and Moore state machines

Mealy machine	**Moore machine**
Outputs are a function of current state and input signals	Outputs are a function of current state only
Output can change between changes between state	Outputs change only when the current state changes
Output can changes any number of times during a clock cycle, which may result in glitches on the outputs	Output is delayed by one clock cycle, but is stable
More output combinations are possible as the outputs are a function of inputs too	Since the outputs are a function only of the current state, the numbers of output combinations are fewer with the Mealy machine

Mealy machine	Moore machine
If the inputs are not registered, the combinatorial paths could potentially be larger than Moore machine; Hence, a relatively lower frequency is expected compared to a Moore machine	Can expect higher frequency compared to Mealy machine, as the combinatorial paths are typically shorter, and no input paths are involved

2.5.3 Illustrate the differences between binary encoding and one-hot encoding mechanisms state machines.

The encoding in state machines are primarily either binary [sometimes called sequential] encoding or the one-hot encoding. Both mechanisms eventually lead to decoding of the states, but their logic implementation, timing and area implications differ. The differences are summarized in the table as follows:

Table 2-9. Difference between sequential and one-hot encoding

Binary encoding	One-hot encoding
Requires fewer number of FFs to represent current state	Number of FFs required is equal to the number of states in the FSM
As there is combinatorial path in the output logic, its timing is not as good as the one-hot encoding mechanism	Better output timing, as there is no output logic. Only clk→q delay, and hence faster
Preferred approach in ASICs unless the timing in output path is critical	Useful and necessary in register rich application like FPGAs
Since the number of FFs is limited, good optimization is required for encoding	Don't need to optimize the state encoding, as each state has unique flop anyway.
Adding or deleting states requires tracking the side effects to the other states in the FSM	Easy to maintain, that is, adding or deleting states is easy, and doesn't effect the rest of the states
Tedious to debug, since a wrong state transition needs a walk through of the next state combinatorial logic	Easy to debug, since a wrong state transition can be easily detected by looking at the current state values
Critical path analysis requires tracking the combinatorial logic	Easy to find critical paths during Static Timing Analysis (STA)

2.5.4 Explain a reversed case statement, and how it can be useful to infer a one-hot state machine?

The ***case*** expression need not necessarily be a variable. When a constant is used in a ***case*** expression, the value of the constant expression will be compared against each of the ***case*** item expressions. This is called a reversed case statement. This coding style fits the one-hot state machine scenario very well.

In the following code, a one-hot state-machine is illustrated, using reversed case statement. Since the case statement expression will cause entry into the case statements for any value, the first case item that matches will cause the exit from the case statement.

```
module one_hot(clk, rst_n, rd_n, ready, done,
               out0, out1, out2, out3);

input clk, rst_n, rd_n, ready, done;
output out0, out1, out2, out3;

parameter drive_out0 = 4'b0001;
parameter drive_out1 = 4'b0010;
parameter drive_out2 = 4'b0100;
parameter drive_out3 = 4'b1000;

reg [3:0] current_state, next_state;

// the sequential process
always_ff @(posedge clk or negedge rst_n)
  if (rst_n == 1'b0)
    current_state <= drive_out0;
  else
    current_state <= next_state;

// The combinatorial process
always_comb
  begin
  next_state = current_state;
  case (1'b1)
    current_state[0]: // drive_out0
      if (~rd_n)
        next_state = drive_out1;
      else
        next_state = drive_out2;
    current_state[1] : // drive_out1
      if (!ready)
        next_state = drive_out3;
      else if (done)
        next_state = drive_out0;
    current_state[2] : // drive_out2
      if (!ready)
        next_state = drive_out3;
      else if (done)
        next_state = drive_out0;
    current_state[3] : // drive_out3
      if (ready & ~rd_n)
        next_state = drive_out1;
      else if (ready & rd_n)
        next_state = drive_out2;
    default: next_state = drive_out0;
  endcase
  end

assign out0 = current_state[0]; // no operation
assign out1 = current_state[1]; // read operation
assign out2 = current_state[2]; // write opeartion
assign out3 = current_state[3]; // waiting for ready

endmodule
```

2.6 Memories

Memories form an important part of the chip design. The memories can be small enough to form a simple register array or as a cache. The presence of memories is increasing in the chips as the size of the area grows. This section discusses the implications of inferring multi-dimensional arrays as memories in the designs and a few considerations in choosing the memories from technology vendors.

2.6.1 Illustrate how a multi-dimensional array is implemented.

Static memories can be synthesized by the synthesis tools implementing the storage element inferred within the array construct. The following is an example code for synthesizing small synchronous static memories that can be used like a simple register file within the larger design.

```
module my_memory (datai, datao, clk, wr_n, addr);

parameter width = 4;
parameter log2_depth = 16;

input [width - 1 :0] datai, addr;
input clk, wr_n, rd_n;
output [width - 1 :0] datao;

reg [width - 1 : 0] memory [log2_depth -1 : 0];
reg [width - 1 : 0] datao;

always @(posedge clk) begin
  if (wr_n == 1'b0)
    memory[addr] <= datai;
  else if (rd_n == 0)
    datao <= memory[addr]; // Synchronous read
end

// Combinatorial read
// assign datao = memory[addr];

endmodule // my_memory
```

The above code effectively synthesizes 64 FFs whose inputs and outputs will be tapped based on the address values.

Verilog-2001 has introduced multi-dimensional memories. The same example above can be extended for three dimensions of the memory, that is, x, y and z, as follows:

```
module my_memory (datai, datao, clk, wr_n,
                  addr_x, addr_y, addr_z);

parameter width = 4;
parameter log2_d = 4;

input [width -1 :0] datai, addr_x, addr_y, addr_z;
input clk, wr_n, rd_n;
```

```
output [width -1 :0] datao;

reg [width -1 : 0] memory [log2_d -1 : 0]  // addr_x
                          [log2_d -1 : 0]  // addr_y
                          [log2_d -1 : 0]; // addr_z
reg [width -1 : 0] datao;

always @(posedge clk) begin
  if (wr_n == 1'b0)
    memory[addr_x][addr_y][addr_z] <= datai;
  else if (rd_n == 1'b0)
  // synchronous datao
    datao <= memory[addr_x][addr_y][addr_z];
end

// combinatorial datao
// assign datao = memory[addr_x][addr_y][addr_z];

endmodule // my_memory
```

The multi-dimensional arrays above would eventually get synthesized into (x*y*z*width) = (4*4*4*4) = 256 individual FFs. Placing the appropriate multiplexes from the Q output of these FFs and gating logics for the D inputs decide the data in and data out.

Using a hardmacro of memory from a semiconductor vendor has better timing, area, and power, as its logic is optimally placed, rather than synthesizing it using discrete logic.

Note that instantiating a technology specific memory will make the design non-reusable with a different technology. One of the recommendations in this inevitable situation is to bring the pins of the memory all the way to the top level of the module, and instantiate the design and the memory in a wrapper, and not within the core of the design. Since most vendors have similar pin-outs of memory design, the user can also have a choice to instantiate the memory from any vendor in the wrapper. The wrapper can then be instantiated in the top-level netlist.

Check with your semiconductor vendor for the availability of the type of memory that you are interfacing into your system design.

2.6.2 What are the considerations in instantiating technology-specific memories?

Instantiating technology specific memories are required in many applications. Depending upon the application, the choice of memory is based on the following performance variables:

- **Area**: If the area is the prime concern on the die, then a high-density memory is required. This is typically targeted for high volume applications or chips with large on chip memory blocks. The overall area will also depend upon the process technology of the memory block.
- **Frequency**: If the speed is the prime concern, then high-speed memories are required which operate at high frequencies. Note that these memories could potentially be larger in area.
- **Power**: This is one of the critical concerns for low voltage and low power applications of chips in cellular phones, hand held devices, etc. Also, if the power dissipation becomes high, then the operating conditions begin to be de-rated, to the extent that the performance of the overall system becomes lower. It also increases the cost of final packaging of the chip for dissipation purposes. Note that power dissipation is tightly coupled with the frequency at which the memory will be used.

The other design variables in considering the memories are:

- **Capacity**: The capacity of the memory is typically specified in the resolution of bits. For example, a memory is specified as 512Kbits.
- **Voltage**: Since some memories are designed for specific voltage ranges, it is important to pick the memory meeting the desired voltage ranges.
- **Synchronous or asynchronous**: This variable specifies whether the memory will have a synchronous read/write or an asynchronous read/write. Which one is to be used primarily depends upon the presence of a clock element, and the matching of timing requirements of the memory and the design.
- **Single port or multi port**: This variable determines whether the storage within the memory is accessed by a single read/write port or multiple ports. One of the critical issues during the use of a multi-port memory is the resolution on what happens when

multiple ports are trying to do a write to the same memory location.

- **Flip-flop or latch based:** This variable determines if the storage element within the memory is based on a flip-flop or the latch. The important considerations for this memory are the testability and power. Note that a FF based design is more testable than a latch based design.
- **Scannable or not:** With the size of the memory increasing nowadays, the scannability of the memory is an important criteria. Many manufacturers and vendors are providing the BIST logic for making the memory scannable.

Just like any other electronic component, the following manufacturing variables also need to be considered in choice of memories for a mass production application:

- **Unit cost**: This variable will eventually drive the overall cost of the chip, board, and the system itself. It matters a lot in a mass production scenario.
- **Availability**: Availability of the memories will impact the time to market for the end-product success.
- **Failure rate**: The yield of the memory must be high, and the failure rate must be low. BIST circuits will be required to be added within the chip, along with the memories to test them.

The choice of memory will depend upon what the end application is, and hence requires a good balance in all the above considerations.

2.6.3 What are the factors that dictate the choice between synchronous and asynchronous memories?

Synchronous memories, as the name suggests, have a clock as one of the primary inputs. All the writes happen, based on the rising or falling edge of this clock, when the data meets the setup time requirements. All reads happen from the Q output of the flops, after the data ready time.

Asynchronous memories don't have a clock interface. The data writes and reads typically happen with an enable pin.

The main differences between the two memories are as follows:

Table 2-10. Difference between synchronous and asynchronous memories

Synchronous memories	Asynchronous memories
Data writes and reads based on a clock port	Data writes and reads typically based on an enable pin
Has better static timing, because the data output from a synchronous memory is registered	The data output from an asynchronous memory is a combinatorial lookup of its address inputs; Therefore, this combinatorial logic could potentially become a critical element in the timing path
Has larger area compared to asynchronous memory	Area is less compared to synchronous memory
Read operations are generally two clock cycles, minimum: the first cycle is usually used by the memory to sample the address, and the second cycle will be used by the external system to sample the read-data	Both read and write cycles are asynchronous, based on "enable" pins

2.7 General Design Considerations

This section briefly discusses the general design considerations like reusability and other factors that need to be considered early in the design cycle. Reusability of a design is not something that should be deferred until the end of an implementation. This needs to be considered early and all the way during the implementation of the design.

2.7.1 What are some reusable coding practices for RTL Design?

A reusable design mainly helps in reducing the design time of larger implementations using IPs. The topic of reusability has been very well discussed in the Reuse Methodology Manual (see References at the end of this book for details of the book). The following key points summarize the main considerations during the implementation phase:

- Register all the outputs of crucial design blocks. This will make the timing interface easy during system level integration
- If an IP is being developed in both Verilog and VHDL, try to use the constructs that will be translatable later into VHDL.
- Avoid snake paths, as it will make both debugging tedious and synthesis inefficient.

- Partition the design considering the clock domains and the functional goals.
- Follow lexical and naming conventions that are self-descriptive and facilitate future product maintenance.
- Avoid instantiation of technology specific gates
- Use parameters instead of hard-coded values in the design
- Avoid clocks and resets that are generated internal to the design
- Avoid glue logic during top level inter-module instantiations

2.7.2 What are "snake" paths, and why should they be avoided?

A snake path, as the name suggests is a path that traverses through a number of hierarchies, and may eventually return back to the same hierarchy from which it originated.

Snake paths must be avoided in a design for the following reasons:

- It will constitute a long timing path, and hence, be the surprise critical path when static timing analysis is done at the top level. It may not show up during the timing analysis of the unit level blocks if it is poorly constrained.
- The synthesis tools need to put more effort in characterizing the constraints of the path across the hierarchies, and the compile time can get higher.

Some tips that can be followed to avoid the snake paths are:

- Register the outputs of modules with different functional objectives.
- Partition the design functionally, to avoid long paths across different hierarchies.

Keep checking for the presence of the snake paths by periodically running synthesis on the fully integrated RTL, even if it is not fully verified functionally. This will give early feedback through the timing reports for the presence of a path traversing across multiple hierarchies.

2.7.3 What are a few considerations while partitioning large designs?

A large design needs to be approached in a hierarchical fashion. The following considerations need to be taken while partitioning these designs:

- **Functionality:** The functional grouping of the logic within a hierarchy is the prime criteria during partitioning the design. Typical partitioning of hierarchies are:
    - **Address and data paths:** This module typically contains the address and data path registers, which drive the address and data buses of the primary outputs.
    - **Control logic:** This module typically contains Finite State Machines (FSMs), and the module gets the inputs for the FSMs, whose outputs drive the controls for the rest of the logic.
- **Clock domains:** In a multiple clock design, it is recommended to group the logic connected in the same clock domain in a single module. When signals need to interact with another module with a different clock, it is recommended to go through a synchronizer module, which takes in the input from the source clock domain and synchronizes it to the clock domain of the destination module.
- **Area:** Having too little logic in a module will create too many hierarchies, and too much logic within a single module will create issue of not being able to do fine tune control during floorplanning later during the backend process.

Verilog doesn't constitute any limit on the number of hierarchies, but it is a good practice to not have too many (lots of leaf level hierarchies of FFs) or too few (just one huge module!) hierarchies.

2.8 Multiple clock Design Considerations

While each module works well at its unit levels, it is important to consider the perspective of reliability when the signals from the design unit communicate to/from the signals of the other design units. The approach to a synchronous design is quite helpful, but the presence of multiple clock domains in a circuit is getting common. The reliability becomes especially challenging when the signals are communicated across clock domains. This section discusses a few issues to be considered when signals cross the clock domains, and how the reliability can be improved.

2.8.1 How can I reliably convey control information across clock domains?

When control signals are traversing across clock domains, the signal appears as an asynchronous input at the destination clock domain. Hence, this signal needs to be synchronized to meet the setup and hold requirements of the destination clock domain, so that the downstream logic can have valid logic levels. Otherwise, the FF will enter into meta-stable state, in which case it will not be able to arrive at a valid state in a given amount of time. The output of a meta-stable FF can be at an intermediate voltage level, or may oscillate invalidating logic down the signal path.

One of the common methods is to have a two-stage synchronizer FFs between the source and destination clocks. If the first FF enters into a meta-stable state, due to any race condition between the clk and D inputs, then the Q value captured in the first flip-flop is an unknown, that is, either a 1 or a 0 ("x" in simulation), depending upon the resolution of the changes in the inputs.

By having two flip-flops in series, the second flip-flop is always sure to capture the resolved state of the first flip-flop as a stable data, even if the first one is meta-stable for a time after the rising edge of the clock.

The following is a 2-stage synchronizer. Note that the data is coming from source `clk1` while the two FFs are driven by `clk2`.

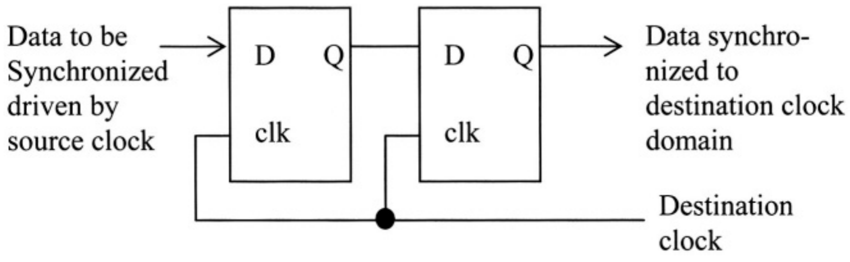
Data to be Synchronized driven by source clock
D
Q
clk
D
Q
clk
Data synchro-nized to destination clock domain
Destination clock

Figure 2-12. 2 FF synchronizer

Some chip and IP vendors even have a special optimised cell just for the synchronization purpose. Although these cells have lesser setup and hold time requirements, these cells may be larger in area than normal FFs and also consume more power. Note that instantiating such technology specific cells could make the design non-reusable with a different library vendor. In such a case, it is recommended to have a module defined with the two synchronizing flip-flops and instantiate them in the design.

The above synchronizer only takes care of the level signals long enough to be sampled by the next rising edge of the destination clock. In the case of a pulsed signal transmission, with widths that could be less than the destination clock frequency, the above synchronizer logic is not helpful. The

readers are encouraged to read about good design implementation in the following reference, titled, "Crossing the abyss: asynchronous signals in synchronous world", that can be found in the following URL
http://www.reed-
electronics.com/ednmag/article/CA310388?pubdate=7%2F24%2F2003

2.8.2 What is a safe strategy to transfer data of different bus-widths and across different clock domains?

When data is to be transferred across different bus width and different clock domains, a FIFO (First In First Out) is an ideal component. If the bus width between the write (the side which **pushes** the data into the FIFO) and read (the side which **pops** the data from the FIFO) sides are different, then it becomes an asymmetrical FIFO. Many IP and chip vendors have asymmetrical and dual clock FIFOs in their libraries. An entity diagram of a typical asymmetrical and dual clock FIFO is shown in the following figure:

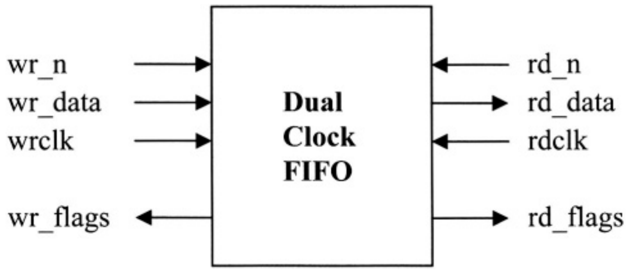

Figure 2-13. Assymmetrical width FIFO

The flags in the FIFO above are typically the full, empty, almost-full and almost-empty. The thresholds for these FIFOs can either be set as an input signal or as an instantiating parameter. The widths of the `wr_data` and `rd_data` busses are different, but are usually related by an integral multiple (that is, one width is an integral multiple of the other).

2.8.3 What are a few considerations while using FIFOs for posted writes or prefetched reads that influence the speed of the design?

FIFOs are typically used in numerous data transfer applications for performance and sustenance reasons. One of the main applications of FIFOs is to post write transactions and to prefetch the data reads.

The advantages of using the FIFOs for posted writes or prefetched reads are:

- FIFOs in general help as a temporary storage buffer, which stores the data written from the write path until it is popped out by the consumer. Thus, in an application like a bridge across two protocol buses with different frequencies, FIFOs help in completing the bus cycles of a faster host sooner. This allows other masters in the host bus to use the bus more efficiently.
- The performance of write data transfer from a bridge that is faster is a lot better when it stores the data in the FIFO, as it doesn't have to be held up by a slower slave through wait states during the individual beats of the data transfer.

The disadvantage of using the FIFOs for posted writes or prefetched reads are:

- Suppose the originating master posts the data into the FIFO, and assumes the data transfer to have happened to the destination slave, and the slave now issues an ERROR. It has to be communicated back to the master, since it assumes the data transfer to have taken place. Typically, in SoC environments, it is taken care of by issuing a high priority interrupt to the host or the originating master.
- If the originating master aborts a read transaction late in the cycle when the read prefetch has already taken place, there is a possibility of a stale data remaining in the read FIFO. When such a condition occurs, the read FIFO may need to be flushed before a new read transaction.
- In order to ensure data coherency between a read followed by write situation, all reads to the same slave address space must be blocked until the previous write transaction is completed. This is typically monitored by watching the empty signal of the FIFO.

In general, FIFOs are very useful to reduce bus latencies and functionally necessary when the bus widths are asymmetrical.

2.9 Common "Gotchas" in Synthesizable RTL

This section explains how and why certain unintentional "gotchas" occur after coding.

2.9.1 What will be synthesized of a module with only inputs and no outputs?

A module with only inputs and no outputs will synthesize into a module with no logic, since there is nothing to be synthesized as an output.

2.9.2 Why do I see latches in my synthesized logic?

There is more than one reason why latches could be seen in synthesized logic. This information is typically present in the elaboration log file of the synthesis tool.

- The ***if-else*** clause in the ***always*** block to which the latch is associated doesn't have a final else clause.
- The ***reg*** declaration of the variable doesn't have any value assigned upon entry to the combinatorial ***always*** block if the variable is used in an ***if*** statement without the else clause.
- There could be no default clause of a ***case*** construct that is not complete or the variables assigned within the ***case*** were not assigned a ***default*** value before entering the case statement.

2.9.3 What are "combinatorial timing loops"? Why should they be avoided?

Combinatorial timing loops are hardware loops in which the output of either a gate or a long combinatorial path is fed back as an input to the same gate or to another gate earlier in the combinatorial path. These paths are generally created unintentionally when a variable from one combinatorial block is used to drive a signal that is used in the same combinatorial block from which the variable was derived. This typically happens in large size combinatorial blocks, wherein it is difficult to visually track that a loop is getting created.

These combinatorial feedback loops are undesirable for the following reasons:

- Since there is no clock edge in between to break the path, the combinatorial loops will infinitely keep oscillating and triggering a square waveform, whose duty cycle is dependent upon the sum of ON delays and OFF delays across the combinatorial path. For example, the following code is a combinatorial loop:

```
assign out1 = out1 & in1;
```

This will cause the `out1` to feed in combinatorially back as one of the inputs.

- These loops cause a problem in testability, since they can inhibit the propagation of the logic forward.

Combinatorial loops can be caught quite early by one of the following means:

- Periodic use of linting tools throughout the development process. This is by far the best and easiest way to catch and fix loops early in the design cycle.
- During functional simulation, the desired output behavior doesn't appear in the output, or the simulation doesn't proceed ahead at all, because the simulator is hung.
- If the loop is undetected during simulation, many synthesis tools have suitable reporting commands, which detect the presence of a loop. Note that synthesis tools proceed with the static timing analysis by breaking the timing arc of the loop for critical path analysis.

2.9.4 How does the sensitivity list of a combinatorial always block affect pre- and post- synthesis simulation? Is this still an issue lately?

With Verilog-1995, between synthesis and simulation, it is important to have all elements that are in the RHS of the statements, or used within conditional statements, to be part of the sensitivity list of a combinatorial always block.

While the synthesis tools go ahead and make use of the nets that are not in the sensitivity list, simulation will ignore change on those nets during logic evaluation. As a result, the behavior seen during functional simulation and post synthesis is different.

Typically, text editors like emacs with a Verilog language mode have been able to automatically infer the right nets, and automatically add it into the sensitivity list. Linting tools will provide error messages during parsing of the RTL code.

From Verilog-2001 onwards, this is not an issue anymore. The language now has an implicit event_expression list, which adds all nets and variables read by procedural and timing control statements into the sensitivity list. The event_expression is indicated by @(*). For example, in the following combinatorial block, all elements of the RHS are in the sensitivity list, as required by Verilog-1995.

```
always @(in1 or in2 or in3 or in4)
begin
  out1 = (in1 ^ in2) & (in3 | in4);
end
```

The same in Verilog-2001 can be written in two ways, as:

```
// note the use of "," in the place of "or"
always @(in1, in2, in3, in4)
begin
  out1 = (in1 ^ in2) & (in3 | in4);
end
```

or

```
always @(*) // note the use of "*"
begin
  out1 = (in1 ^ in2) & (in3 | in4);
end
```

The same code in SystemVerilog can be represented using the always_comb procedure, as follows:

```
always_comb
begin
  out1 = (in1 ^ in2) & (in3 | in4);
end
```

Note that the code is now simpler, relinquishing the user from keeping track of the sensitivity list. It is much more maintainable and readable, too. The key advantage of using the ***always_comb*** procedure over the implicit sensitivity list of @(*) is that the former is executed right from time 0 like an ***assign*** statement, whereas the latter waits for an event to trigger its activation. The simulation and synthesis tools figure out the elements of sensitivity list automatically. Check with your simulation and synthesis tool vendor for the support of SystemVerilog and this construct.

2.10 Coding techniques for Area Minimization

This section describes some of the techniques using RTL coding and parameterized approach for providing optimal area requirement for a soft IP. Rather than have the entire logic, which may not be useful for all the different users, the area optimization of unwanted logic will be useful in large SoCs. Removing unwanted area not only reduces silicon area, but also reduces switching activity, and, hence, the power, too.

2.10.1 How do the *`ifdef*, *`ifndef*, *`elsif*, *`endif* constructs aid in minimizing area?

The proper use of compiler directives like *`ifdef*, *`ifndef*, etc. can help in minimizing the area during post elaboration and during logic optimization. Since the use of compiler directives is a compile time operation, it is a static decision for the session of simulation and during synthesis.

The following is an example of how the compiler directives can be used for minimizing the area of a logic design.

```
`define MIN

module area_min_byifdef (in1, in2, in3, in4, out1);

input  in1, in2, in3, in4;
output out1;

`ifdef MIN
   assign out1 = in1 & in2; // minimal area
 // more related logic to MIN
`else // larger area
   assign out1 = (in1 & in2) | (in3 ^ in4);
   // more related logic to not-MIN
`endif

endmodule
```

Note that the use of compiler directives is legal to pick instantiations of modules itself, and, hence, can be helpful to pick a module with appropriate area size. For example, in the following code, the compiler directive is used to pick the correct type of counter, that is, ripple counter or carry lookahead, counter depending upon the directive.

```
// `define CLA

module area_min_byifdef (in1, in2, cin, sum, cout);

input  in1, in2, cin;
output sum, cout;
`ifndef CLA
  ripple_adder U_ripple (
     .in1 (a), .in2(b), .in3(c),
     .sum(sum),.cout(cout)
   ); // smaller area, longer timing
`else
   cla_adder U_cla (
     .in1 (a), .in2(b), .in3(c),
     .sum(sum), .cout(cout)
   ); // larger area, faster timing
`endif

endmodule
```

In the above example, the *`ifndef* was used to illustrate the absence of a *`define* for CLA. Note that the `define for CLA has been commented out. If it gets uncommented, then the carry lookahead adder instantiation gets selected.

Hence, in this approach, the selection of the appropriate "section" of code during parsing and elaboration decides the final area of implementation.

2.10.2 What is "constant propagation"? How can I use constant propagation to minimize area?

Constant propagation is a very effective technique for area minimization, since it forces the synthesis tools to optimize the logic in both forward and backward directions. Since the area minimization is achieved using constants, this technique is called constant propagation. An example of constant propagation is shown below:

```
module const_prop (in1, in2, out1, out2);

input  in1, in2;
output out1, out2;
```

```
parameter create_logic = 0;

assign out1 = (create_logic == 1) ?
              in1 & in2 : 1'b0;
assign out2 = (create_logic == 1) ?
              in1 | in2 : 1'b0;

endmodule
```

Note that create_logic is a ***parameter*** within the module, that controls the logic backwards from both the outputs `out1` and `out2`. It could also control the logic forward from the inputs `in1` and `in2` by adding internal wires to either select the direct input `in1` or the 1'b0. An example of how the forward constant propagation works is as follows:

```
wire int_in1, int_in2;

assign int_in1 = (create_logic == 1) ? in1 : 1'b0;
assign int_in2 = (create_logic == 1) ? in2 : 1'b0;

assign out1 = int_in1 & int_in2;
assign out2 = int_in1 | int_in2;
```

When this parameter is 0, it forces the logic zero in the ***assign*** statements, it results in logic zero propagation in either direction. As a result, no logic gets enabled and the logic is optimized in synthesis. When this parameter is 1, the logic is synthesized.

Note that different techniques to override the parameter will also work, that is, the constant propagation will be effective, even with parameter override.

Hence, the default value of the parameter can be set to 1, and be overridden to 0, by different parameter overriding techniques, when required to minimize the area.

SystemVerilog has also introduced a new construct ***const*** which declares a variable as a constant. The const construct can be used to enforce constant propagation, just like other constants like ***parameter.*** For example, the same example above can be applicable by replacing the ***parameter*** with ***const***, as follows:

```
module test_const (in1, in2, out1, out2);

input  in1, in2;
output out1, out2;

const bit create_logic = 1;

assign out1 = (create_logic == 1) ?
               in1 & in2 : 1'b0;
assign out2 = (create_logic == 1) ?
               in1 | in2 : 1'b0;

endmodule
```

The output with the ***const*** construct above is exactly the same as when a ***parameter*** is used.

2.10.3 What happens to the bits of a *reg* which are declared, but not assigned or used?

When any of the bits of a ***reg*** declaration is unused, the logic corresponding to those bits gets optimized away. For example, in the following code, the bits 2:1 are unused, although the `int_tmp` is declared to be [3:0]. This code will synthesize the logic for bits [3] and [0], and no logic for bits [2:1].

```
module lower (in1, clk, reset, out1);
input  in1, in2;
input  clk, reset;
output [1:0] out1;

reg [3:0] int_tmp;

always @(posedge clk or negedge reset)
begin
  if (!reset)
    int_tmp <= 0;
  else
// Only bits 0 and 3 are used.
// Bits [2:1] are not assigned
    int_tmp[0] <= in1;
    int_tmp[3] <= in2;
end

assign out1 = int_tmp;

endmodule
```

2.10.4 How does the *generate* construct help in optimal area?

Verilog-2001 ***generate*** can be useful in area optimisation techniques. This construct must be coded within a module scope. Unlike the ***`define*** based approach, this construct allows the use of a variable, declared using the ***genvar*** construct, to control the logic generated.

The ***generate*** construct can be used in two ways: either with a ***for*** loop within the ***generate-endgenerate*** scope, or using the conditional ***if-else*** construct, or the conditional ***case*** construct within the ***endgenerate.*** The "generate for" usage helps in precisely instantiating the right amount of logic in a reusable design. The "generate if "usage determines whether the logic should get generated at all. The amount of logic is precisely controlled using the construct and its variable.

This is best illustrated using examples, as follows. The first is the use of an ***if-else*** clause within the ***generate-endgenerate*** constructs. The analogy is very similar to the use of ***`ifdef***, as discussed earlier in FAQ 2.10.1. Note that the ***genvar*** construct is not required in this case.

```
module if_generate (in1, in2, out1); // 1.12.5
parameter xor_logic = 1;

input  in1, in2;
output out1;

wire out1;
generate
  if (xor_logic == 1)
    out1 = in1 ^ in2;
  else
    out1 = in1 & in2;
endgenerate

endmodule
```

In the above example, depending upon the resolution of the value of the parameter `xor_logic`, either the XOR gate or the AND gate gets generated. Although this is a very simple use of this construct, the same analogy can be extended for multiple statements through the use of begin-end statements. The parameter can be overridden, using ***defparam*** construct, too. Note that even instantiations can be controlled in the if-or else clause.

The second way to use the ***generate*** construct is with ***for*** loops. The variable used in the ***for*** loop has to be a ***genvar*** declaration. The variable is then used in a ***for*** loop which is instantiating the exact number of modules required.

```
module andit (in1, in2, out1);
input in1, in2;
output out1;
wire out1;

assign out1 = in1 & in2;

endmodule

module forgen_test (in1, in2, out1);
parameter width = 4;
input [width-1 : 0] in1, in2;
output [width-1 : 0] out1;

wire [width-1 : 0] out1;

genvar i; // variable for the for loop

generate for (i=0; i < width; i = i+1)
  begin : AND_BLOCK
    andit U1 (in1[i], in2[i], out1[i]);
end endgenerate

endmodule
```

The "AND_BLOCK" block identifier is required for any heirarchical names generated by the concatenation of the block identifier and the variable value as {generate_block_identifier, genvar_value}. In this case, the hierarchical names generated were:

```
AND_BLOCK[0].U1  // Lowest index of the for loop
AND_BLOCK[1].U1
...
AND_BLOCK[3].U1  // Highest index of the for loop
```

These generated names can be used in hierarchical path names, just as in a hierarchical design. The above example saves a lot of code for explicit instantiations, especially if the variable size is large. Also, it allows good control on how many of these need to be instantiated by the parameter width. Thus, precise area control can be achieved. Note that this simple anding module can be extended for more complex hierarchies, too.

The third way to use the ***generate*** construct is through the ***case*** statement within the ***generate-endgenerate*** scope. This allows selective branching to take place through the ***case*** statement, and, hence, controlling which of the sections of the code to be finally ‘generated’. An example of the conditional selection of a module instantiation through a ***case*** statement is as follows:

```
module anding (in1, in2, out1);
input in1, in2;
output out1;
wire out1;
assign out1 = in1 & in2;
endmodule

module oring (in1, in2, out1);
input in1, in2;
output out1;
wire out1;
assign out1 = in1 | in2;
endmodule

module xoring (in1, in2, out1);
input in1, in2;
output out1;
wire out1;
assign out1 = in1 ^ in2;
endmodule
```

```
module casegen_test (in1, in2, out1);

input  in1, in2;
output out1;
wire out1;
parameter operation = 0;

generate
  case (operation)
    0 : anding U1 (in1, in2, out1);
    1 : oring  U1 (in1, in2, out1);
    2 : xoring U1 (in1, in2, out1);
    default : anding U1 (in1, in2, out1);
  endcase
endgenerate

endmodule
```

A few important points of the above example are:

- The case condition "operation" has to be a constant, or a ***genvar*** variable in order to make a definitive decision during the conditional instantiation. Otherwise, it is a syntax error, since the tools will encounter this as an unknown value during elaboration. Hence, the approach is useful in parameterised designs.
- Depending upon the value of operation above, the output either gets anded, ored, or xored. But, eventually only one of them will happen. This is useful in scenarios where a selective implementation needs to be instantiated, and the instantiation of that module can be selectively controlled.

2.10.5 What is the difference between using *`ifdef* and *generate* for the purpose of area minimization?

As discussed in the earlier questions, both ***`ifdef*** and ***generate*** constructs can be used for the purpose of area minimization. The difference between the two in using these constructs for area minimization is summarized in the following table:

Table 2-11. Difference between using `ifdef and generate to minimize area

`ifdef	***generate***
Construct can be used inside and outside the scope of ***module*** definition	Construct has to be used **only** within the scope of ***module*** definition
Construct works on the boolean presence or absence of a ***`define*** of the ***`ifdef*** variable	Construct can use the value of a variable using the ***genvar*** construct when used in a ***for*** or ***case*** constructs
Useful only in equivalence to the ***if-else*** construct, and cannot perform any looping or branching operations	The ***genvar*** variable can be used in a ***for*** loop or ***case*** branch, to allow multiple or selective instantiation of variables and modules

2.10.6 Can the *generate* construct be nested?

No. The ***generate*** construct cannot be nested. It is a syntax error to try to nest the ***generate-endgenerate*** construct.

However, the ***if, case,*** and ***for*** constructs within the ***generate-endgenerate*** can be nested. The constructs can also be used within one another, too, that is, ***if*** within ***case,for*** within ***if*** *etc.*

You can also use multiple non-nested ***generate-endgenerate*** constructs within the module.

2.11 Coding for Better Static Timing Optimization

When a design gets compiled into a netlist, the various elements of the delays in the path, like the cell delay, routing delay, etc., contribute in deciding the overall performance of the chip. The timing impact of the design should be factored very early in the design process, during functional partitioning and coding of the design. It will be too late to consider timing impacts later in the functional verification cycle.

This section discusses a few topics on the different factors impacting the static timing of the design.

2.11.1 What is a critical path in a design? What is the importance of understanding the critical path?

A critical path is the path through a circuit that has the least slack. It is not necessarily the longest path in the design. There can be more than one critical path in a design. In fact, all paths whose difference between the arrival time and required time at the endpoint is negative (, that is, negative slack) is a violating path.

Understanding and identifying the critical path in a design is important for the following reasons:

- It helps fix the static timing problems, especially when the endpoint is a D input to a flip-flop, and the critical path delay is violating the setup time requirement for the flop.
- Shortening the critical path delay obviously improves frequency and, hence, the performance of the logic.

If the critical path is identified early in the design flow, then appropriate functional changes can be done early on in the project to terminate the path to the D input of a flop at an appropriate point in the path. This point has to be carefully chosen, considering the side effects in latency and static timing that would arise due to the staging of the path through a flop.

If the source of the critical path is from a primary input, it is recommended to register the input. Although this could add to the latency, this strategy will eventually help in improving the frequency of operation.

2.11.2 How does proper partitioning of design help in achieving static timing?

Partitioning a design correctly helps in multiple stages of the design, all the way until the backend flow. The best approach for partitioning is to plan the partitioning of the design before writing HDL code. It is important to keep these considerations early on, to avoid hierarchical, port, or logic changes late in the design. The following are some of the criteria for design partitioning:

1. Logical partitioning: The partitioning of the modules with close logical associativity is a very common approach. This way, it is both easy to debug and modular in approach. Typically, the partitioning logic boundaries are datapaths (register's and glue), control (FSMs), memories and I/O. Logistically, it also helps with having multiple team members do thorough unit level verification, and it helps with better design management and version control. All combinatorial logic associated with the same clock domain should also be closely within the same module. Inter-module partitions can restrict logic optimization by synthesis tools. Hierarchical boundaries prevent any combining of related logic. Typically, a module size should be around 5K gates.

   For synthesis tools to consider resource sharing and freedom to optimize, all relevant resources need to be within one level of hierarchy. If the resources are not within one level of hierarchy, synthesis tools cannot make tradeoffs to determine whether or not the resources should be shared.

   It is during the logical partitioning that the designer has the freedom to decide upon the registering of the outputs between critical inter-module hierarchies. This will immensely reduce the possibilities of long combinatorial paths and combinatorial snake paths in the design, and, hence, better static timing implication.

   Special function logic, like the pads, I/O drivers, clock generation and boundary scan should be at separate logical hierarchies.

   Any on-chip memories, like SRAMs or DRAMs should be placed at the top level. This will make the physical design interaction and floorplanning tasks more effective by better timing analysis.
2. Goal based partitioning: Partitioning based on different design goals of speed and area will eventually help the tools do a good job. Modules with different goals can be specified with their respective constraints during the synthesis for the tools to do a good job.
3. Clock domain partitioning: Partitioning the logic according to same clock domain plays an important role in synthesis, static timing analysis, and scan insertion. The inter-clock false paths can be defined within a single synchronizer module, and the entire module is now with a single clock domain.
4. Reset based partitioning: If a particular SoC has multiple resets, then it is a good idea to consider reset based partitioning, too. This helps all the storage elements within the module to wake up gracefully at the same time.

2.11.3 What does it mean to “retime” logic between registers? How does it effect functionality?

Retiming is the process of relocating registers across logic gates, without affecting the underlying combinatorial logic structure. This process is achieved by borrowing logic from one time frame and lending it to the other, while maintaining the design behavior. When you have a pipelined design, for example, in a datapath of a design, then retiming is a technique for reducing the critical path within the pipeline.

Retiming has benefits as follows helps in balancing the paths between the pipeline stages

Retiming also has potential restrictions as follows:

- Note that, although retiming can be used to reduce the critical path between the pipeline registers, it cannot be used to reduce the latency of the design.
- A retimed design may not be formally equivalent to the original design.

2.11.4 Why is one-hot encoding preferred for FSMs designed for high-speed designs?

Since there is one explicit FF per stage of a one-hot encoded state machine, there is no need of output state decoding. Hence, the only anticipated delay is the clock to q delay of the FF. This makes the one-hot encoding mechanism preferable for high-speed operation.

2.12 Design for Testability (DFT) considerations

Design for Testability or DFT techniques are design efforts that need to be considered upfront during the design phase, to ensure that the design under test is eventually testable. This process could increase the area in the expense of increasing the fault coverage. By proper DFT considerations upfront, the test generation/development time and the time with the tester can be reduced. While there could be a few pins that get increased for better fault coverage, it provides better observability and controllability, which are the key considerations for good testability.

The following FAQs discuss a few factors that can effect the testability and fault coverage of a design, and what the DFT techniques are.

2.12.1 What are the main factors that affect testability of a design?

The following are some of the main factors affecting the testability and fault coverage of a design:

- Presence of tri-state buses in the design
- Reset of a FF driven by the output of another FF
- Presence of derived clocks in the design
- Presence of gated clocks in the design
- Presence of latches in the design

Each of the above issues are discussed in the following FAQs.

2.12.2 My chip has on-chip tri-state buses. What are the testability implications, and how do I take care of it?

Normally, tri-state buses shouldn't be present within the chip, as they consume more power. However, if the tri-state buses are present inside a chip, care should be taken to avoid bus contention, that is, driving different values at the same time. It affects the power, since the bus conflict will drain huge currents, and cause damage to the chip. To avoid bus contention during the scan testing phase, the enable to the tri-state buffer should be controllable, that is, by AND'ing it with the scan enable signal. In normal mode, the `scan_en_n` signal is de-asserted (logic 1), to allow the control to flow through, but in test mode, the drivers are disabled to avoid contention. The control inputs to these enables are assumed to be originated from the outputs of FFs. This is shown in the following figure:

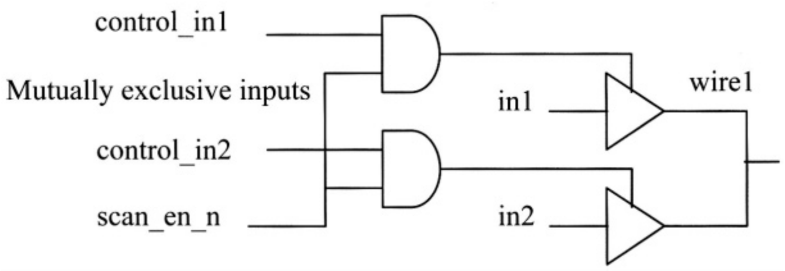

Figure 2-14. Tri-state and DFT

Verilog sample code for these buffers is illustrated in the following:

```
assign wire1 = (control_in1 & ~scan_en_n) ?
```

```
                    in1 : 1'bz;
assign wire1 = (control_in2 & ~scan_en_n) ?
                    in2 : 1'bz;
```

2.12.3 Some Flip-Flops in my chip have their resets driven by other Flip-Flops within the chip. How will this affect the testability, and what's the workaround?

Normally, the asynchronous set or reset of a FF is controlled by the primary reset input pin. Sometimes it becomes inevitable to have the output of one FF to drive the asynchronous set/reset of another FF. In that case, during the scan testing, if the driving FF gets a pattern such that it resets the driven FF, it will destroy its data. To prevent this, the reset should be OR'ed with a `test_mode` test mode signal. The following figure illustrates this mechanism:

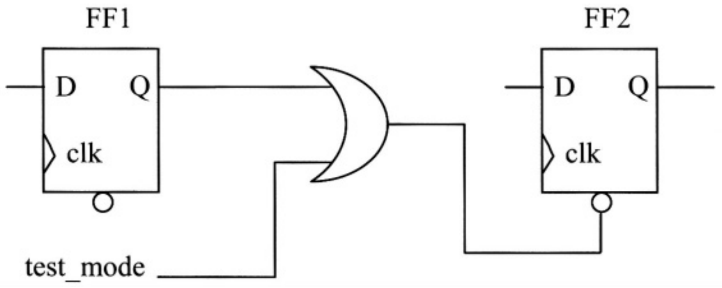

Figure 2-15. Reset and DFT

The `test_mode` primary input is disabled (in this case 1'b0) during normal operations. However, during testing, the `test_mode` signal is asserted to 1'b1, thus making the asynchronous reset deactivated. This will avoid corruption of FF2 output when a predictable pattern is being sent from FF1 to FF2.

2.12.4 I have derived clocks in my chip. What are the testability implications, and what is the workaround for it?

Derived clocks are generated by clock dividers through Flip-Flops or PLLs in a chip. Since these are derived from within the chip, there should be a control input from the primary pins, to avoid the Flip-Flops capturing data when they are not supposed to.

In this case, a multiplexor needs to be added in the clock path, with the control being the `test_mode`, and the inputs to the multiplexor being the regular clock and derived clock. That way, the final clock to the Flip-Flops is controllable between the derived clock in the normal mode and the regular clock in the test mode. Note that the `test_mode` signal doesn't change dynamically, and, hence, it is okay to have a multiplexor in the clock path. Note, too, that anytime a switch in the clock is done, all the Flip-Flops need to be reset, to have a known starting value, and avoid spurious capture of data in their data lines.

The following diagram illustrates the implementation:

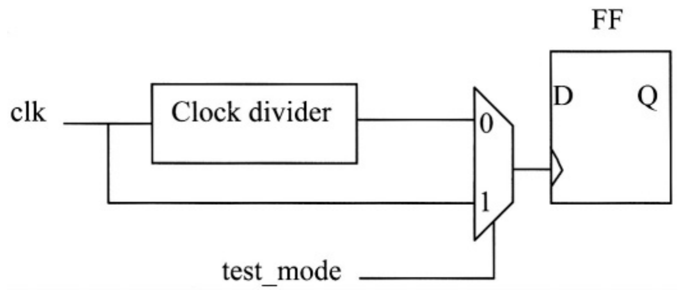

Figure 2-16. Multiplexor in clock path using derived clocks

2.12.5 My chip is power sensitive, and, hence, there are gated clocks in it. What are its testability implications and workaround?

Gated clocks are inevitable in some designs to save power. Since the clock now passes through combinatorial logic, the gated clock is no longer controlled from a primary input, making it impossible to scan in the data.

The workaround is to logically OR a `test_enable` pin to the enabling pin of the AND gate that gates the clock. Look into a FAQ 2.13.5 in this chapter for more details of implementing this workaround.

2.12.6 What is the implication of a combinatorial feedback loops in design testability?

The presence of feedback loops should be avoided at any stage of the design, by periodically checking for it, using the lint or synthesis tools. The presence of the feedback loop causes races and hazards in the design, and leads to unpredictable logic behavior. Since the loops are delay-dependent, they cannot be tested with any ATPG algorithm. Hence, combinatorial loops should be avoided in the logic.

2.12.7 How does the presence of latches affect the testability, and what's the workaround?

Since the enable to a latch isn't the regular clock going to the rest of the Flip-Flops in the design, its output is not controllable directly from a primary input. In order to bring controllability to the latch, the enable to the latch needs to be OR'ed with a primary input pin like `test_mode`, as shown in the following figure.

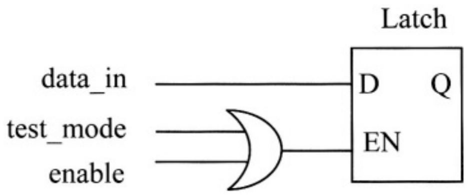

Figure 2-17. Latch with OR'ed test enable

This way, the latch can be forced to become transparent when the test data needs to be forced into it.

2.13 Power Reduction considerations

Power reduction is a critical requirement in design of chips that are used in battery-operated devices. The more power a chip uses, the hotter it operates, slower it runs. The reliability of the chip decreases at higher temperatures. This section discusses how RTL can be used to influence the power dissipation within a chip, and what issues need to be considered when coding for the power saving.

2.13.1 What are the various methods to contain power during RTL coding?

Any switching activity in a CMOS circuit creates a momentary current flow from VDD to GND during logic transition, when both N and P type transistors are ON, and, hence, increases power consumption.

The most common storage element in the designs being the synchronous FF, its output can change whenever its data input toggles, and the clock triggers. Hence, if these two elements can be asserted in a controlled fashion, so that the data is presented to the D input of the FF only when required, and the clock is also triggered only when required, then it will reduce the switching activity, and, automatically the power. The following bullets summarize a few mechanisms to reduce the power consumption:

- Reduce switching of the data input to the Flip-Flops.
- Reduce the clock switching of the Flip-Flops.
- Have area reduction techniques within the chip, since the number of gates/Flip-Flops that toggle can be reduced.

The following FAQs discuss in depth how each of the above can be implemented in RTL.

2.13.2 Illustrate how the switching of data input to the Flip-Flops helps in power reduction.

In a circuit where the Flip-Flops need to be updated very rarely compared to the frequency of the clock, then it is appropriate to update the FF only at that time, and avoid the switching of its output all other times. This can be achieved through an enable FF, as shown in the following figure:

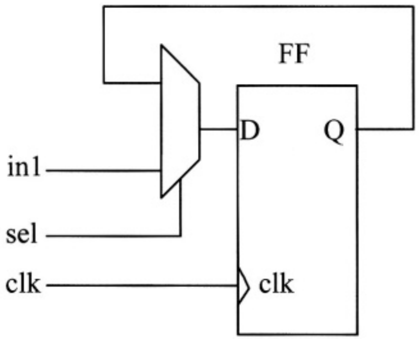

Figure 2-18. Using enable FF for power saving

If the control input comes from a state machine which can track exactly when this FF has to be enabled to capture the new input data, then the enable to the multiplexor can switch the multiplexor towards the input data.

Otherwise, it will be feeding the previous stable output from Q into the data input of the FF.

An illustration of Verilog RTL that implements the enable FF is illustrated as follows:

```
module enable_ff (clk, sel, reset_n, in1, out1);
input reset_n, sel, clk,in1;
output out1;

reg out1;

always @(posedge clk or negedge reset_n) begin
  if (! reset_n)
    out1 <= 1'b0;
  else if (sel)
    out1 <= in1;
  else
    out1 <= out1;
end

endmodule
```

The above style can be incorporated in the designs by following a coding convention for the flip-flops. But, this technique alone is not sufficient as a power reduction technique, as it has a drawback which is discussed in the next FAQ 2.13.3.

2.13.3 What is the drawback of using the enable flip-flop to reduce the power consumption?

Although the switching of the data is reduced using enable Flip-Flops, the clock input to the Flip-Flops is still running to a large number of other Flip-Flops.

One side effect of the enable FF method is that it will introduce logic into the setup time of the D input, and possibly add to the delay, if the D input was the endpoint of a critical path.

The other side effect is that the area increases if these Flip-Flops happen to be the storage elements of a large bank of registers.

2.13.4 Illustrate an example of clock gating to help in reduction of power.

Clock gating is a common mechanism to save power. This technique reduces the switching activity of the output of the FF by:

- eliminating the need for reloading the same value in the register during multiple clock cycle.
- Reducing the clock network power dissipation.

The most common method of clock gating is through the use of a latch and a gate. The following figure illustrates the implementation of this mechanism:

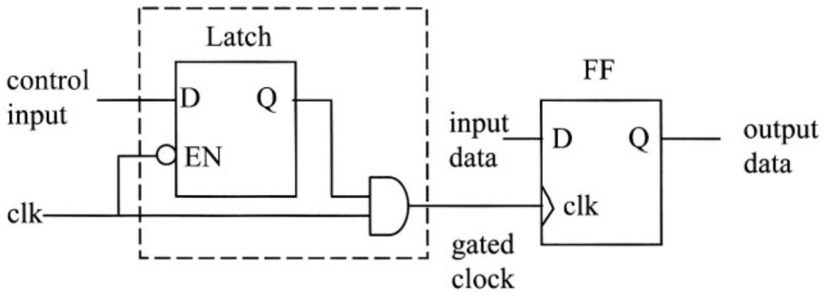

Figure 2-19. Using latch for clock gating

When the `clk` is in its low phase, the latch is enabled. The control input, which actually decides whether to gate the clock or not, is now propagated through the clock to its Q output. Here, if the control input is high, the Q of the latch is high during the low phase, and remains so until the next low phase of the `clk`. This keeps the AND gate enabled. In the mean time, when the `clk` arrives, it gets propagated to the gated clock net. This happens cleanly, without any glitches, because the latch output is stable for sufficient time to meet the Flip-Flops setup requirements. When the control input goes low, it negates the AND gate and, hence, prevents the `clk` from being propagated to the gated clock net. This makes the gated clock net to be at 0 without any switching activity.

A simple Verilog code that illustrates the above logic is illustrated as follows. Note that the implementation of this strategy in large designs is best done through the synthesis tools without having to manually implement this strategy in the designs containing a large number of FFs.

```
module gated_ff (in1, cntrl_in, clk, reset_n, out1);

input  cntrl_in, in1, clk, reset_n;
output out1;

wire gated_clk;
reg  d_latch, out1;

always @(cntrl_in, clk) begin
  if (clk)
    d_latch <= cntrl_in;
end

assign gated_clk = d_latch & clk;

always @(posedge gated_clk or negedge reset_n) begin
  if (! reset_n)
    out1 <= 1'b0;
  else
    out1 <= in1;
end

endmodule
```

The main reason for using a latch is to prevent the glitches on the `gated_clk` net since its changes happen during the low phase of the clock.

Although the above illustration is shown for only one FF, the gated clock can actually be driven to all the remaining Flip-Flops in its clock domain. Also, the gating element has been an AND gate, depending upon the polarity of the enable to the latch and the low phase of the clock for a rising edge. This gate can change, depending upon any changes to these two polarities, that is, the logic level to enable the latch, and the edge of the clock, whether it is rising or falling.

The logic shown within the dashed box will require being instantiated multiple times, depending upon how many branches the main clock tree has. Depending upon the buffering, clock skew, and loading, many such instances could be placed on each branch of the clock tree or at the root level.

2.13.5 What are the side effects of latched clock gating logic, and how is it fixed?

Although the use of clock gating through latches is a good way to save power, it introduces the problem of testability, as illustrated. Due to the latch, the controllability of the gated clock signal is reduced, that is, the gated clock signal is now in the mercy of the control input only. During testability, if this signal is low, then it disables the propagation of the clock itself.

To resolve both the above issues, additional logic needs to be added to enhance the testability. One way to increase the controllability of the gated clock is to introduce a control point in the input of the latch, so that the latch is "ON" during scan testing. This is illustrated in the following figure:

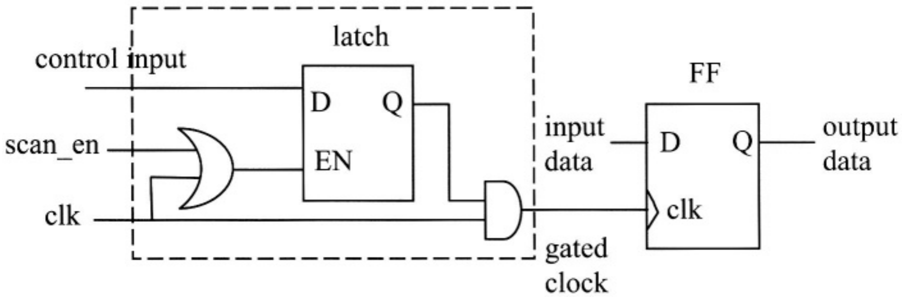

Figure 2-20. Using latch for clock gating

Based on the OR gating above, the scan enable signal will override the control input, such that the output of the latch enables the AND gate to propagate the `clk` net into the input of the Flip-Flops.

A simple Verilog code that illustrates the above implementation is as follows. Note that synthesis tools can implement this logic illustration automatically, for all the FFs in a large design, rather than having to do manually.

```
module gated_ff (in1, scan_en, clk,
                 reset_n, cntrl_in, out1);

input  scan_en, in1, clk, reset_n, cntrl_in;
output out1;
```

```
wire gated_clk, latch_en;
reg  d_latch, out1;

assign latch_en = scan_en | clk;

always @(cntrl_in, latch_en) begin
  if (latch_en)
    d_latch <= cntrl_in;
end

assign gated_clk = d_latch & clk;

always @(posedge gated_clk or negedge reset_n)
  if (! reset_n)
    out1 <= 1'b0;
  else
    out1 <= in1;

endmodule
```

Sometimes there have been situations that the test tools used in the foundries don't support the control before the latch and require it to be present after the latch. Since such a requirement comes from the foundry, the above circuit can be easily changed to position the OR gate after the latch, as shown below:

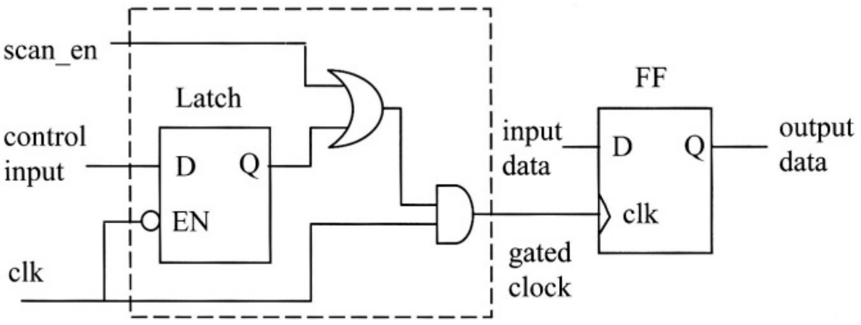

Figure 2-21. Latch controllability after the output

A simple Verilog code illustrating the above is as follows:

```
module gated_ff_out_or (in1, scan_en, clk,
                         reset_n, cntrl_in, out1);
```

```
input  scan_en, in1, clk, reset_n, cntrl_in;
output out1;

wire gated_clk, latch_en;
reg  d_latch, out1;

// This is the latch
always @(cntrl_in, clk) begin
  if (~clk)
    d_latch <= cntrl_in;
end

assign clk_gate = scan_en | d_latch;
assign gated_clk = clk_gate & clk;

// This is the gated ff
always @(posedge gated_clk or negedge reset_n)
  if (! reset_n)
    out1 <= 1'b0;
  else
    out1 <= in1;

endmodule
```

2.13.6 What are a few other techniques of power saving that can be achieved during the RTL design stage?

The following design considerations during RTL coding help in the reduction of power within the logic:

- Run high frequency signals through as few intermediate logic levels as possible. This way, only those cells which need to be run at high frequency switch, and the rest of the logic can run at a relatively lower frequency. This would require multi clock design within the chip, preferably where the clocks are integral multiples of each other. The safe approach would be to route one master clock into the chip, and generate its sub clocks within the chip.

- Only use as many Flip-Flops as required to store the data values, that is, if only 4 bits of a 32 bit register are going to be used, it is not required to register the remaining 28 bits. Normally the additional unused FFs will be optimized away by the synthesis tools.

- Gate the inputs using a select line. For example, the address lines from a CPU are continuously changing, and may not all the time refer to your device. In that case, it is better to gate the rest of the logic following the address lines with a signal like chip-select, which will hence reduce unnecessary switching activity. The chip select can be generated from one central address decoder. Although this decoder is switching all the time, it helps in the unnecessary switching in lots of other logic distributed elsewhere.

- Choose Gray coding for state machines instead of binary encoding: Since only one bit changes at any Gray transition, the number of Flip-Flops switching, and the switching in the logic that it drives, is reduced. Note that this would potentially require more Flip-Flops than the binary encoded approach. Hence, for the most frequent transition arcs, use Gray coded transitions. Focus the gray coding efforts on common return to zero state transitions.

- Choose a multiplexor instead of on chip tri-state buses: The biggest issue of on chip tri-state buses is the bus contention. Since there is a high possibility of one buffer beginning to drive the interconnect before the other has finished, there is a small window in which potentially opposite polarities are driven. This causes a transient short circuit on the internal bus. The choice of a multiplexor avoids the bus contention, but it could potentially add to the number of gates and logic path. Consider registering the inputs that come from these long paths. Tri-state buses also require internal pull up resistors and higher current signal drivers.

2.13.7 What are a few system level techniques, apart from RTL, that can influence in the reduction of power for the chip?

Having discussed a few techniques of saving power through RTL in the above FAQs, the following are a few system level variables that can influence power reduction:

- Reducing operating voltage: Since power consumed is directly proportional to the square of the voltage, operating at a lower voltage is one way of saving power. Many of the semiconductor vendors have libraries that are designed specifically for low power. However, note that there could be side effects in the static timing when the low power libraries are used.

- Reducing operating frequency: Since the power consumed is directly proportional to the frequency, design technique of operating at a lower frequency, but increased bus widths to maintain the data rate performance requirements should be considered. For example, the rate of data transfer of a 32 bit bus at 100MHz is the same as a 64 bit bus at 50MHz. Note that there will be additional design corners that get introduced as the widths increase, especially in non aligned byte transfer scenarios.

- Running the I/O voltage different from the core voltage: In this technique, the I/O ring of cells are working at a different voltage from the rest of the core cells. This achieves interfacing to the signals external to the chip with a different voltage requirements than the core. It also isolates the core from output-transition noise.

- Lower the capacitance of the routing network, especially for high frequency signals.

2.13.8 What are a few power reduction techniques that can be achieved through static timing?

Power reduction can be achieved in all stages of the chip process, that is, RTL techniques of gate clock, synthesis tool optimizing away unused logic, reducing capacitance of the routing network during backend, and also through good static timing. The following are a few considerations on how the power can be reduced through static timing:

- Control clock skew between logic gate inputs.
- Ensure that flip-flop inputs meet setup and hold time requirements, to avoid extended output settling transitions caused by metastability.

2.13.9 What are a few power reduction techniques that can be implemented during the backend analysis?

The following are a few parameters within the chip that can significantly influence the overall power consumption, which can be taken care of during the backend phase:

- Have shorter routes for power and timing sensitive logic: Since the capacitance of a routing net is a function of the length, width and impedance of the route, a long route typically has higher capacitance than the shorter alternative. Since dynamic power consumed is directly proportional to the capacitance, that is, $P = CV^2f$, lower capacitance means lesser power. This would mean the logic blocks need to be closer to each other.

- Reduce excessive loading: Heavily loaded nets cause higher capacitance and higher power consumption.

2.13.10 What are a few power reduction techniques that can be implemented during board design?

The following are a few techniques that can reduce power consumption at a board level

- Reduce the chip interconnection dynamic power by limiting the number of I/O pins, the loading on each, pin and the average frequency at which each pin toggles.

- Minimize the trace lengths between the chips output and other device inputs.

SUMMARY

This chapter discussed how the various Verilog constructs get inferred during synthesis, and the static timing implications. The chapter also discussed a few techniques on area reduction, and issues on testability and power.

Chapter 4

MISCELLANEOUS

INTRODUCTION

This chapter lists various questions that may come up during the course of using the Verilog HDL. These FAQs are not in any particular order or category.

4.1.1 What is the difference between a vectored and a scalared net?

Both ***scalared*** and ***vectored*** are Verilog constructs used on multi-bit nets to specify whether or not specifying bit and part select of the nets is permitted. For example,

```
module test_scalared_vectored;

wire scalared [3:0] wire1;
wire vectored [3:0] wire2;

wire bit1, bit2;

// syntax error to use bit select of a vectored net
assign wire2[1] = 1'b0;
// okay to use bit select of a scalared net
assign wire1[2] = 1'b1;

// syntax error to use bit selects of a vectored net
```

```
assign bit1 = wire2[1];
// okay to use bit selects of a scalared net
assign bit2 = wire1[2]; // scalared net

endmodule // test_vectored_scalared
```

4.1.2 What is the difference between *assign-deassign* and *force-release*?

The ***assign-deassign*** and ***force-release*** constructs in Verilog have similar effects, but differ in the fact that ***force-release*** can be applicable to *nets and variables,* whereas ***assign-deassign*** is applicable *only to variables.*

The procedural ***assign-deassign*** construct is intended to be used for modelling hardware behaviour, but the construct is not synthesizable by most logic synthesis tools. The ***force-release*** construct is intended for design verification, and is not synthesizable.

4.1.3 What is the order of precedence when both *assign-deassign* and *force-release* are used on the same variable?

The ***force*** statement overrides the value of ***assign*** statement until it is released. The following example illustrates the same:

```
module forcerelease;

reg [1:0] w1;

initial begin
$display("1 w1 = %0d, t = %0d",w1, $time);

assign w1 = 1;
#5 $display("2 w1 = %0d, t = %0d",w1, $time);

force w1 = 2;
#5 $display("3 w1 = %0d, t = %0d",w1, $time);

release w1;
#5 $display("4 w1 = %0d, t = %0d",w1, $time);

deassign w1;
#5 $display("5 w1 = %0d, t = %0d",w1, $time);
```

```
end

endmodule
```

The above code produces the following output:

```
1 w1 = x, t = 0
2 w1 = 1, t = 5
3 w1 = 2, t = 10
4 w1 = 1, t = 15
5 w1 = 1, t = 20
```

As evident from the above, the ***force*** command has overridden the assigned value earlier and relinquished it back to its assigned value after the release command.

4.1.4 How can I abort execution of a task or a block of code?

The Verilog ***disable*** statement will be able to abort the execution of a task or block of code. Disabling a block of code would be useful in scenarios like:

- Executing a “break” command within a loop, to skip the rest of them loop iterations, and exit the loop
- Terminating a task before its completion

Note that the ***disable*** statement is used with a block name. For example,

```
initial begin : block1
  begin : block2
    statement1;
    //etc.
    disable block2;
    statement5;
    statement6;
  end // of block2
  statement7;
end // of block1
```

In the above example, “block1” and “block2” are the block names. As the statements get executed, when the ***disable*** statement is hit, the remaining statements in block2, that is, statements 5 and 6 don't get executed. They are skipped, and the execution resumes from statement 7.

The only restriction on using the ***disable*** statement is that it cannot be used in a ***function,*** as it would invalidate the ***function*** and its return value. It can, however, be used in a ***task.***

SystemVerilog also introduces the ***break*** command to exit the loop. This is more graceful than the ***disable*** statement as illustrated in the following example:

```
module test_break;
integer i;
initial begin
  i = 10;
  while (i) begin
    i--;
    if (i == 5) begin
      break;
    end else
      $display("i = %0d",i);
  end // while
end // initial

endmodule // test_break
```

If the current iteration needs to be skipped on certain conditions, SystemVerilog has added a command ***continue*** which will directly jump to the end of the loop. For example, in the following code, the loop would be skipped for all odd values of the variable i.

```
module test_continue;

integer i;

initial begin
  i = 10;
  while (i) begin
    i--;
    if (i % 2) begin
      // $display("iodd = \t%0d",i);
      continue;
    end else
      $display("ieven = %0d",i);
  end // while
end // initial

endmodule // test continue
```

The output of this test program displays:

```
ieven = 8
ieven = 6
ieven = 4
ieven = 2
ieven = 0
```

4.1.5 What are the differences between the looping constructs *forever, repeat, while, for, and do-while*?

The statements ***forever, repeat, while,*** and ***for*** are the looping statements supported in Verilog-2001 and the ***do-while*** construct is introduced in SystemVerilog. These statements fundamentally differ in how many times the statements within the ***begin-end*** scope of the loop is executed. The following bullets summarize these differences:

- ***forever***: Executes the statements within its begin-end block **forever**, without any variable to control it until the simulation session terminates. For example,:

```
initial begin
  clk = 1;
  forever begin : clk_block
    #(clk_period/2) clk = ~clk;
  end
end
```

Note that a ***forever*** loop *cannot* be terminated via a ***disable*** statement.

- ***repeat:*** Executes statements within its ***begin-end*** block a **fixed** number of times that is evaluated *once* at the beginning of the loop. For example:

```
integer var1, i;
```

```
initial begin
  var1 = 8;
  i = 0;
  repeat (var1) begin : this_loop
    i = i + 1;
    $display("i = %0d",i);
  end
  $finish;
end
```

Note above that the `var1` has to be within brackets, as `(var1)`. Without the brackets, it is a syntax error. Since the variable size is a constant that needs to be fixed a priori before entering the ***repeat*** loop, the possibility of an infinite loop through the ***repeat*** construct doesn't occur. The ***disable*** statement can be used to exit the loop prematurely.

- ***while:*** Executes the statements within its ***begin-end*** block indefinitely, *until* its expression becomes false. The loop expression will also evaluate to false if it has a X or Z value in it. For example,

```
integer i;
initial begin
  i = 8;
  while (i) begin : this_loop
    i = i - 1;
    $display("i = %0d",i);
  end
  $finish;
end
```

The above code can potentially end up being an infinite loop, if there is no statement to falsify the expression of the ***while*** loop. The ***disable*** statement can be used to exit the loop prematurely.

- ***for***: Executes the statements within its ***begin-end*** block, based on the number of times its variable is modified, in steps, until the variable evaluates to X or Z or false.

```
integer i;
initial begin
  for (i = 0; i < 8; i = i + 1) begin : loop1
```

```
      $display("i = %0d",i);
      // i = i + 1;
    end
    $finish;
  end
```

The above code displays the values of i as 0,1,2,3,4,5,6,7. Note that ***for*** loop also has a potential of entering into an infinite loop, if the expressions don't get falsified over a period of time. The ***disable*** statement can be used to exit the loop prematurely.

Unlike the ***repeat*** loop, the loop variable can be manipulated within the ***for*** loop. For example, in the above code, if the statement "i = i + 1" within the ***begin-end*** block is uncommented, then the display will be of values 0,2,4,6. This capability could, if used incorrectly, also be a cause for entering an infinite loop. Thus, modifying the loop variable in a ***for*** loop is not a best practice, and should be strongly discouraged.

Another unique feature of ***for*** loop is that it is the only looping construct supported by the many synthesis tools. The statements within the ***for*** loop are replicated, once for each value of the looping index. For this reason, the bounds of ***for*** loop need to be fully deterministic when the code is read by a logic synthesis tool.

- ***do-while***: Executes the statements within its ***begin-end*** block,until the variable within the **while** statement evaluates to X or Z or false. The expression is evaluated at the *end* of the loop.

```
module test_dowhile;
integer i;

initial
begin
  i = 4'd2;
  do
    begin
      i++;
    end
  while (i <= 4'd15);
  $display ("i = %0d",i); // displays i = 16
  $finish;
```

```
    end

    endmodule // test_dowhile
```

The key advantage of the ***do-while*** statement is that it guarantees the execution of the loop statements at least once before reaching the end of the loop. Hence, this avoids the duplication of the loop body outside the start of the loop before checking the entry of a normal ***while*** loop.

4.1.6 What is the difference between based and unbased numbers?

Based numbers are those which have a base identifier preceding the actual number. For example, `4'habcd` represents a number with a hexadecimal base. An unbased number has no base specified before it, and represents a simple integer. For example, the integer 23 is an unbased number, since it has no base specification preceding it.

4.1.7 What does it mean to "short-circuit" the evaluation of an expression?

Verilog supports numerous operators that have rules of associativity and precedence. In some of the expressions, the result of the expression can be evaluated early on, due to the precedence and influence to override the rest of the expression. In that case, the entire expression need not be evaluated. This is called short-circuiting and expression evaluation.

For example:

```
input in1, in2, in3, in4;
wire wire1, wire2;

assign wire2 = (in1 > in2) & (in3 | in4);
```

In the expression above, the result of the test `(in1 > in2)` is ANDed with the result of `(in3 | in4)`. If the result of `(in1 > in2)` is false (`1'b0`), then tools can already determine that the result of the AND operation will be 0. Thus, there is no need to evaluate `(in3 | in4)` and rest of the equation is short-circuited.

4.1.8 What is the difference between the logical (==) and the case (===) equality operators?

The "==" operator specifies logical equality and the "===" equality represents the case equality. The "==" logical equality operator is used to model hardware, where comparisons to high-impedance or unknown values would yield ambiguous results (an unknown or X in simulation). In other words, if any of the operands of the "==" operator contain X or Z, then the result is ambiguous and is an "X". For example,

```
a = 2'b1x;
b = 2'b1x;

if (a == b)
  $display ("reached if");
else
  $display ("reached else");
```

Since at least one of the operands, `a`, contains X in one of its bits, the result is X and in this case, the message "reached else" is displayed. Note that in this example, even though it appears that both `a` and `b` appear to be equal, the presence of a X in either will result in a mismatch during the comparison operation.

The "===" case equality operator is intended for verification, where it is important to test if a value is high-impedance or unknown. If any of the operands of "===" contain X or Z bits, their comparison is still considered during evaluation and a Boolean result is reached, that is, the result is a `1'b1` or a `1'b0`. For example,

```
a = 2'b1x;
b = 2'b1x;

if (a === b)
  $display ("reached if");
else
  $display ("reached else");
```

In the above, "reached if" is displayed. The example works the same if X is replaced with Z.

4.1.9 What are the differences and similarities between the logical (<<, >>) and the arithmetic (<<<, >>>) shift operators?

The logical shift operators are (<< and >>). The logical shift operator has been present from Verilog-1995. The arithmetic shift operators are (<<< and >>>), which were introduced with Verilog-2001.

- Three of them, that is, logical left shift (<<), arithmetic left shift (<<<) and logical shift right(>>) operators, shift the bits left/right by the number of bit positions specified by the right operand, and the vacated bits are filled with zeros.
- The arithmetic right shift operator (>>>) will fill the vacated bits with 0 if the left operand is unsigned, and the most significant bit if the left operand is signed.

The following example illustrates all the above facts:

```
module test;

reg [7 : 0] tmp1, tmp2; // default unsigned
reg signed [7 : 0] tmp3, tmp4; // signed

initial begin

tmp1 = 8'b00001100;

tmp2 = tmp1 << 4; // logical unsigned shift left
$display("tmp2 = %b",tmp2); // tmp2 = 11000000

//arithmetic unsigned shift left
tmp2 = tmp1 <<< 4;
$display("tmp2 = %b",tmp2); // tmp2 = 11000000;

tmp1 = 8'b10001100;

// logical unsigned shift right
tmp2 = tmp1 >> 2;
$display("tmp2 = %b",tmp2); // tmp2 = 00100011

// arithmetic unsigned shift right
tmp2 = tmp1 >>> 2;
$display("tmp2 = %b",tmp2); // tmp2 = 00100011
```

```
tmp3 = 8'b10001100;

// arithmetic signed shift right
tmp4 = tmp3 >>> 2;
$display("tmp4 = %b",tmp4); // tmp4 = 11100011
// Note that the msb "1" got filled in all
// vacated bit's

end

endmodule  // test
```

4.1.10 What is the difference between a constant part-select and an indexed part-select of a vectored net?

The constant part-select and indexed part-select are two types of addressing the contiguous bits of a vectored net/reg, or any multi-bit variable declaration.

The constant part select, as the name suggests, has a constant definition for its upper and lower bounds. For example,

```
reg [msb : lsb]
```

where both `msb` and `lsb` must be constant expressions, that is, they have fixed values during compile time itself.

In the case of indexed part select, as the name suggests, the width of the part select (the right operand) must be constant, but the starting or ending point of the part select (the left operand)can vary. For example,:

```
module test;

reg [7 : 0] tmp1;  // descending order
reg [0 : 15] tmp2; // ascending order
integer i;
initial begin
i = 0;
tmp1[i +:8] = 8'hab; // assigns to tmp1[7:0]
$display("tmp1[7:0] = %0h",tmp1[0 +: 8]);
i = 6;
```

```
tmp1[i -:4] = 4'ha; // assigns to tmp1[6:3]
$display("tmp1[6:3] = %0h",tmp1[6 -: 4]);
i = 0;
tmp2[i +: 8] = 8'hef; // assigns to tmp2[0 : 7]
$display("tmp2[0:7] = %0h",tmp2[0 +: 8]);
i = 15;
tmp2[i -:8] = 8'hcd; // assigns to tmp2[8 : 15]
$display("tmp2[8:15] = %0h",tmp2[15 -: 8]);

end

endmodule
```

Note the "**+:**" and "**-:**" syntax in the above usage. The + : indicates that the part select bit numbers will *increment*from the value of the left operand up to the width specified by the right operand (which must be constant). The - : indicates that the part select of the bit numbers will *decrement*from the value of the left operand up to the width specified by the right operand (which must be constant). For the purpose of understanding this better, the general expressions used for analysis are:

variable[base +: offset] and
variable[base -: offset]

to arrive at the variable [physical_msb: physical_lsb].

The following table summarizes the context and usage scenarios:

Table 4-16. Interpretation of physical MSB and LSB for indexed part select

	Physical MSB	Physical LSB
In the case of descending index like reg [7:0] tmp1;		
Base +: offset	offset - 1	base
Base -: offset	base	offset – 1
In the case of ascending index like reg[0:15] tmp2;		
Base +: offset	base	offset – 1
Base -: offset	offset	base

4.1.11 Illustrate how memory indirection is achieved in Verilog.

Indirection is a mechanism where a pointer is passed as a value of an argument to memory. This is very commonly used in software. In Verilog, the address of a memory location can be specified as an expression, too. One way to specify this is in the form of the value of another location in the same memory. For example, in the following:

```
new_value = my_memory[my_memory[10]]
```

suppose the value of `my_memory[10]` = 16'h1a00, then the result of `new_value` is as good as specifying as `my_memory`[16'h1a00]. Hence, the memory indirection can be achieved.

4.1.12 What is the logic synthesized when a non-constant is used as an index in a bit-select?

A multiplexor is synthesized when a non-constant is used as an index in a bit-select. The following is an example:

```
module indexed_mux (data_in, select, data_out);
input [7:0] data_in;
input [2:0] select;
output data_out;

assign data_out = data_in[select];

endmodule // indexed_mux
```

The above RTL code will get synthesized into a multiplexor with 8 bits of `data_in` and 3 bits of `select`, as follows. Typically, the synthesis tools will try to pick up an 8:1 multiplexor from the target library. If it is not available, the multiplexor gets synthesized, using logic gates.

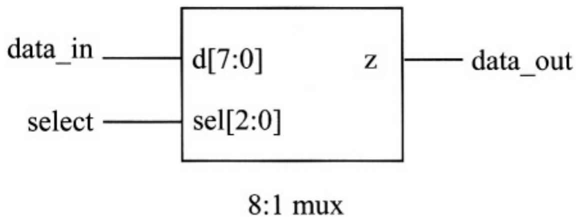

Figure 4-1. A multiplexor generated out of non-constant bit select

4.1.13 How are string operands stored as constant numbers in a *reg* variable?

Strings are stored as ASCII characters in 8 bit fields. For example, the ASCII characters for lower case a-z are 8'h61 to 8'h7a. Hence, the ***reg*** declaration that uses these fields needs to accommodate the correct number of bits with 8 bits for each character.

Since the ascii characters are stored as 8 bit fields, modifying these bits would change the value being displayed. The following example illustrates this.

```
module test;

reg [5*8 -1 : 0] tmp1; //5 chars, 8 bits each

initial begin

//assigns ASCII of each character
tmp1 = "hello";

//displays 'h68_65_6c_6c_6f
$display("tmp1 = %0h",tmp1);

// displays hello
$display("tmp1 = %s",tmp1);

// represents ASCII "y"
tmp1[4*8-1 : 3*8] = 8'h79;

// displays hyllo
$display("tmp1 = %s",tmp1);

end

endmodule
```

SystemVerilog allows the definition of variables as ***string.*** This is a very flexible mechanism of not only initialising these variables with a dynamically allocated array of characters, but also includes a set of associated functions which return the length of the string, character manipulation, case conversion etc.

4.1.14 How can I typecast an expression to control its sign?

The sign of an expression can be controlled by typecasting with two system functions namely ***$signed*** and ***$unsigned.*** These functions evaluate the expression to return the type of sign as requested. For example,

```
module test;

reg [3 : 0] regU; //default unsigned reg variable
reg signed [3 : 0] regS; // signed reg variable

initial begin

   regU = -2; // stored as unsigned value
   regS = -5; // stored as signed value

   // Unsigned math
   $display ("regU+regS = %0d", (regU+regS));

   // Signed math
   $display ("\$signed(regU)+regS = %0d",
             ($signed(regU)+regS));
end

endmodule
```

As illustrated above, casting is very beneficial in the middle of compound operations.

4.1.15 What are the pros and cons of using hierarchical names to refer to Verilog objects?

Verilog allows the access of variables by using hierarchical paths. For example, the status net at the top level can be assigned directly with the value, as seen in the hierarchy underneath:

```
assign status = top.DUT.U_core.U_CSM.status;
```

The advantage of using hierarchical names is:

- It is easy to debug the internal signals of a design, especially if they are not a part of the top level pinout.

The disadvantages of using hierarchical names are:

- Sometimes, during synthesis, these hierarchical names get ungrouped or dissolved or renamed, depending upon the synthesis strategy and switches used, and hence, will cease to exist. In that case, special switches need to be added to the synthesis compiler commands, in order to maintain the hierarchical naming.
- If the Verilog code needs to be translated into VHDL, the hierarchical names are not translatable.

4.1.16 Does Verilog support an (a^b) operator?

Yes. Verilog supports the a^b operation by using two astrices, back to back. This operator was added with the Verilog-2001 release. For example,

```
module powerof (in1, out1);
parameter power = 2;
input   [1 : 0] in1;
output   [3 : 0] out1;

assign out1 = in1 ** power;

endmodule  // powerof
```

A value of 2 would mean `out1 = in1 * in1`, that is, the value getting multiplied to itself. Simulation, however, works for powers other than 2, as well.

4.1.17 What is the main limitation of fork-join in Verilog, and how is this overcome in SystemVerilog?

The main limitation of ***fork-join*** construct in Verilog is that it is static, that is, the execution of the code beyond the join is suspended until all the processes within the ***fork-join*** are completed. For example, in the following code, the last ***$display*** statement gets executed only after 10 time units, although the process 1 is completed in 5 time units:

```
module fork_join_tests;

integer out_val;
```

```
initial begin
  fork
    begin // First process
     #5 $display("exit first process at t = %0d",
                  $time);
    end
    begin // Second process
     #10 $display("exit second process at t = %0d",
                   $time);
    end
  join
  $display("exit fork join at t = %0d", $time);
end

endmodule // fork_join_tests
```

The above code produces the following display outputs:

```
exit first process at t = 5
exit second process at t = 10
exit fork join at t = 10
```

SystemVerilog adds two new keywords for joining parallel processes: ***join_any*** and ***join_none.***

When the ***join*** in the code above is replaced by ***join_any,*** then the following display outputs are produced:

```
exit first process at t = 5
exit second process at t = 10
exit fork join at t = 5
```

Notice that the ***fork-join_any*** exits after the first process gets completed, that is, at 5 time units.

When the ***join*** in is replaced by ***join_none,*** then the following display outputs are produced:

```
exit first process at t = 5
exit second process at t = 10
exit fork join at t = 0
```

Notice that the ***fork-join_none*** exits after spawning both the processes and not waiting for any of them to be completed, that is, exits at at 0 time units.

The ***join_any*** and ***join_none*** constructs of SystemVerilog do not hold the ***fork*** process until all of its process are necessarily completed.

4.1.18 Can I return from a *function* without having it disabled?

It is illegal to disable a function in Verilog-1995 and 2001. However, SystemVerilog has introduced a keyword ***return,*** that skips the rest of the lines of the ***function,*** and returns back to the block that called the ***function.*** For example, in the following code, the function would return back, if `in1` is greater than `in2`.

```
function [31:0] my_func;
  input [31:0] in1;
  input [31:0] in2;
  reg tmp_reg;
  if (in1 > in2) begin
    $display("my_func returning back");
    return;
  end else
    my_func = in1 + in2;
endfunction

module func_multibit;

reg  [31:0] result;

initial begin
  result = my_func(3, 4);
  $display("result = %0d",result);
  result = my_func(4, 3);
end

endmodule // func_multibit
```

The above code would produce the output displays of:

```
result = 7
my_func returning back
```

4.1.19 What is strobing? How do I selectively strobe a net?

Strobing is a facility defined in Verilog by which simulation data on selected nets or variables can be captured at the end of the current simulation time instant, after all the events scheduled for this time have occurred, and just before the simulation time is advanced. In Verilog, strobing is facilitated by the ***$strobe*** system call. Syntactically, this system call is very similar to the ***$display*** system call. An example of the ***$strobe*** system call follows:

```
always @(negedge system_clock)
  $strobe ("Time = %0d, rx_active = %b rx_data = %h",
$time,rx_active, rx_data[7:0]);
```

Functionally, the ***$strobe*** system call creates internal monitoring events of its arguments, which are re-enabled at every user-specified time-step.

Variants of ***$strobe*** include ***$strobeh*** (hexadecimal formatted), ***$strobeo*** (octal formatted), ***$strobeb*** (binary formatted). All of these system calls print their results on the standard output device. For printing the strobed outputs to a specific file, instead of the standard output, there exist the file-specific variants of these system calls: ***$fstrobe, $fstrobeb, $fstrobeh,*** and ***$fstrobeo.*** The syntax of these calls takes on an additional argument for the file-handle. For example:

```
integer file_handle;
initial
   begin
     file_handle = $fopen("vectors.stb");
   end

 always @(negedge system_clock)
   begin
     $fstrobe(file_handle,
        "Time = %0d, rx_active = %b rx_data=%h",
         $time, rx_active, rx_data[7:0]);
   end
```

In addition to ***$strobe,*** Verilog has an additional system call, called ***$monitor,*** which is used for taking snapshots of signal changes during simulation. Like ***$strobe,*** the ***$monitor*** system call also creates internal monitoring-events of its arguments. However, unlike ***$strobe,*** a ***$monitor*** call cannot create other simulation events. For example, we can create a time-event (that is, simulation event) for a ***$strobe:***

```
always @(posedge clock)
  $strobe ( …arguments…);
```

In this example, the values of the argument signals of ***$strobe*** are printed out to the standard output at every posedge of the clock signal.

However, ***$monitor*** effectively, "stands by itself". Example:

```
initial
  begin
    $monitor(
     "Time = %0d: rx_active = %b, rx_data = %h",
      $time,rx_active, rx_data[7:0]);
  end
```

In this example, **every change** on the `rx_active` and `rx_data` signals causes ***$monitor*** to print out the changes to the standard output device.

From this example, it is clear that a ***$monitor*** call for a given set of arguments should be issued **only once** in a simulation.If ***$monitor*** is called more than once, then the most recent invocation overrides all previous calls. In the following example, on the signals `tx_valid` and `tx_data` will be monitored:

```
initial
   begin
     // monitor for receive activity
     $monitor(
      "Time = %0d: rx_active = %b, rx_data = $h",
       $time,rx_active, rx_data[7:0]);
     // monitor for transmit activity:
     // REPLACES the previous $monitor.
     $monitor(
      "Time = %0d: tx_valid = %b, tx_data = $h",
       $time,tx_valid, tx_data[7:0]);
   end
```

Finally, as with ***$strobe,*** there are variants of the ***$monitor*** system call as well: ***$monitorh, $monitoro, $monitorb, $fmonitor, $fmonitorh, $fmonitoro,*** and ***$fmonitorb,*** with the exact same meanings as corresponding with ***$strobe***.

4.1.20 Summarize the main differences between *$strobe* and *$monitor.*

The differences between ***$strobe*** and ***$monitor*** are summarized in the following points:

- ***$strobe*** can be used to create new simulation events, simply by encapsulating the ***$strobe*** system call within a simulation construct that moves simulation time, such as *@(posedge clock), @(negedge clock),@(any_signal)* etc.There can exist multiple ***$strobe*** system calls at the same time, with identical or different arguments.

- ***$monitor*** stands alone. A given set of arguments of ***$monitor*** form their own unique sensitivity list. Only one ***$monitor*** call can be active at any time. Each call to ***$monitor*** replaces any previous call(s) to ***$monitor.***

4.1.21 How can I selectively enable or disable monitoring?

$monitor can be selectively enabled or disabled by the ***$monitoron*** and the ***$monitoroff*** system calls, respectively. The ***$monitoron*** and ***$monitoroff*** system calls affect only the *most recent* call to ***$monitor.***

4.1.22 How can I specify arguments on the Verilog simulator's command line?

User defined command line arguments to the Verilog simulator are usually preceded by a "+", and are generally referred to as "plusargs". For example, a Verilog command line may look like this:

```
<my_Verilog_simulator> +MYGPA=4.0 +MYSCHOOL=geek_factory
```

Here, the plusargs are MYGPA and MYSCHOOL. The values assigned to these plusargs are "`4.0`" for MYGPA and "`geek_factory`" for MYSCHOOL.

Verilog defines system-tasks for determining

- which plusargs are defined : ***$test$plusarg***
- what is the value assigned to each plusarg: ***$value$plusarg***

Continuing with the above example:

```
$test$plusarg("MYSCHOOL")
// would return a non-zero integer,

$test$plusarg("MYGPA")
// would also return a non-zero integer, whereas

$test$plusarg("MYSPECIALIZATION")
// would return an integer zero.
```

Therefore, we can use ***$test$plusargs*** in a Verilog testbench to query if particular plusargs were defined on the command line.

After knowing which plusargs were defined on the command line, the value assigned to each plusarg can be queried as well, using the ***$value$plusarg*** system task.

Again, continuing with the previous example:

```
real gpa_value;
$value$plusarg("MYGPA=%f", gpa_value);
```

would result in

```
gpa_value = 4.0
```

Similarly, the following snippet of code:

```
reg [8*80:1] name_of_school;
$value$plusarg("MYSCHOOL=%s",name_of_school);
```

would result in

```
name_of_school = geek_factory;
```

In other words, $value$plusargs is analogous to the sprintf() function in C.

4.1.23 Can the `` `define `` be used for text substitution through variable instead of literal substitution only?

Typically the `` `define `` text macro has been used for literal text substitution only. For example,

```
`define width 8

// substitutes `width with 8

wire [`width-1:0] wire1;
```

In the above example, `` `width `` is literally replaced by 8 wherever it is used. This was the usage syntax in Verilog-1995. However, from Verilog-2001 onwards, the text substitution can also work, taking in variables, and still do text substitution as required. For example,

```
`define pos_clk(in1) @(posedge in1)

module test_define_text;

wire clk;

initial
  `pos_clk(clk);

endmodule // test_define_text
```

In the above example, wherever the `` `pos_clk `` will be called, it will be substituted by `@(posedge clk)`, with the `clk` being the argument passed to the text macro.

Note : During text substitution, it is important to pay attention to the white spaces, too. For example, in the '***define*** in above example, if a white space exists between `pos_clk` and `(in1)`, the replacement wouldn't work.

SUMMARY

This chapter discussed miscellaneous topics that couldn't be allocated into the rest of the chapters. These topics are spread across the different sections of the Verilog language.

Chapter 5

COMMON MISTAKES

COMMON VERILOG CODING MISTAKES

This chapter describes different errors that aren't detected during the compile time, but show up either as functional problems or as run-time problems during simulations. The list presented is by not exhaustive, but it captures most of the common mistakes seen during the development phase. Each of these mistakes is illustrated with an example, and possible workarounds to avoid these from occurring.

5.1 Some common errors that are not detected at compile-time

These are the mistakes that are not detected during the compile time, that is, it is a legal syntax of Verilog, but it ends up being either functionally incorrect, or causes hang during simulations, due to deadlock or live-lock, etc.

5.1.1 What are some ways a race condition can get created, and how can these race conditions be avoided?

Race condition happens when two variables are being assigned values at the same event time. Race conditions also happen due to incorrect coding style of using the blocking assignments in clocked processes. The destination variables wouldn't have been scheduled to be updated due to the bad coding styles resulting in incorrect values being updated, hence, causing a race between the source and destination variables. The receiving/retrieving agent, which could be another variable, or a function, like ***$display,*** using this variable, would display the value as per its scheduling resolution. In the following example, the same variable is initialised in two blocks:

```
module race;

reg a;
wire b;

initial begin // start at time 0
  a = 1;
end

initial begin // start at time 0
  a = 0;
end

assign b = a;

initial begin // start at time 0
  $display ("a=%0b, b=%0b",a,b);
end

endmodule
```

The above code displays either a 0 or 1 for the variable “a", and it typically follows the value that was last assigned to it. In general, this is heavily dependent on the implementation of the simulator. The value of “b" might be shown as X or 0 or 1, depending on the event ordering of the simulator.

One of the recommendations is to avoid driving variables from multiple sources. If a variable needs to be a function of information from multiple processes, then the assignment to the variable must happen in one place, with the control variables on the RHS of the assignment.

The other recommendation is to assign the variable in-line. Beginning with Verilog-2001, there is a convenient way to initialise the variable from in-line during the declaration itself. For example, the variable a, above, can be initialised in only one place as:

```
reg a = 0;
```

The SystemVerilog standard further enhances the above initialisation procedure, such that all the in-line initialisations are guaranteed to happen prior to the execution of any events before any simulations begin. See also FAQ 5.1.15 for additional side effects related to race condition.

5.1.2 Illustrate how the infinite loops get created in the looping constructs like forever, while and for.

Infinite loops are one of the common things that cause a hang in a simulation. If the loops don't have a finite end value for the looping variable, then the loop never terminates. The following are common examples:

```
reg done;

initial begin // start at time 0
  done = 0;
  while (~done) begin
    #5 $display("Entered loop at t=%0d",$time);
  end
end
```

Note that the above ***while*** loop only exits if some other process changes `done` to be 0. If this never happens, then the loop never terminates, but is syntactically correct.

The fix for the critical loops is to add a check within the ***for*** loop for a way to disable the loop when the loop variable exceeds a limit. Other way is to add a watchdog timer parallel to the ***for*** loop in a ***fork-join.*** This would stop the simulation if the loop doesn't get terminated in a specific number of iterations or a specific amount of time.

With SystemVerilog, an assertion statement can be used instead of having to write a watchdog timer.

```
reg [31:0] i;

initial begin // start at time 0
  for (i=0; i >= 0; i = i + 1) begin
    #5 $display("Entered loop at t=%0d",$time);
  end
```

```
end
```

Note that the above ***for*** loop wouldn't terminate since the termination condition of `i >= 0` is always met. The fix for this is similar to the addition of watchdog timer or use of the assertion feature in SystemVerilog.

```
reg clk;

initial begin // start at time 0
  forever #5 clk = ~clk;
  $display("After forever at t=%0d",$time);
end
```

Note that the ***$display*** statement above is never reached since the ***forever*** statement never gets completed. This makes the ***forever*** statement to be the last statement in any procedural block. The statements after the ***forever*** loop should be added into a different procedural block.

5.1.3 Illustrate the side-effects of specifying a *function* without a range.

It is common to use the ***function*** to assign a value. A mistake can happen if a range for the ***function*** return is not specified. If a range is not specified, Verilog will assume a 1 bit return value. If a multi-bit return value was calculated in the function, only the least significant bit is returned. In the following example, the value of the correct result is 7 if the range of [31:0] is specified in the ***function*** definition. Without a range, however, only the least significant bit of the value is returned, which is 1.

```
reg  [31:0] result;

function my_func;       // returns only the lsb
  input [31:0] in1;
  input [31:0] in2;
  reg tmp_reg;
  my_func = in1 + in2 ;
endfunction

initial begin
  result = my_func(3, 4);
  $display("result = %0d",result);
end
```

There would not be any compilation errors for the function definition. It is a runtime functional error. To correct the error, the function should be declared with a range of sufficient size for the return value, such as:

```
function [31:0] my_func;  // returns the correct range
```

5.1.4 Illustrate how the errors of passing arguments to a *function* in incorrect order is eliminated in SystemVerilog.

The arguments to a ***function*** call have to be exactly in the same order as the input arguments defined in the ***function.*** Passing arguments in an incorrect order to a ***function*** call, would result in incorrect functionality, although it is syntactically correct. For example, in a ***function*** call, two of the inputs, say, `in1` and `in2` could result in incorrect functionality if the variables that call the ***function*** were not passed in the right order.

SystemVerilog has an enhancement to ***function***'s that eliminates this ambiguity, by bringing in a feature to both ***task*** and ***function*** calls, wherein the arguments to the ***function*** can be passed explicitly by name, rather than implicitly by order. In the following code, the arguments of the ***function*** call are connected to their right source and destinations, although the order in which they have been passed are not in the same order the ***function*** definition has been declared.

```
module funct_output (in1, in2, out1, out2,
                     out3, out4);
input  [1:0] in1, in2;
output [1:0] out1, out2, out3, out4;

reg  [1:0] out1, out2, out3, out4;

// void, function doesn't return anything
function void arith;
  input [1:0] in1, in2;
  output [1:0] out1, out2, out3, out4;
  begin
    out1 = in1 & in2 ;
    out2 = in1 | in2 ;
    out3 = in1 ^ in2 ;
    out4 = in1 % in2 ;
  end
endfunction
```

```
always_comb
  begin
    arith (
      .in1  (in1),   // The order of arguments to a
      .out1 (out1),  // function call is immaterial.
      .in2  (in2),   // Inputs and outputs can be in
      .out3 (out3),  // any order as long as they
      .out2 (out2),  // are connected to the right
      .out4 (out4)   // source or destination
    );
  end

endmodule
```

The result of the above ***function*** call, by passing arguments by name, is the same as it was with implicit order of arguments to the ***function*** call. Hence, this eliminates possibilities of in-advertant errors during ***function*** calls.

5.1.5 Using tri-state logic inside a chip

The presence of internal tri-state logic is a critical consideration for power sensitive products. Normally a multiplexor should be used in place of tri-state logic. However, if the tri-state logic remains in the RTL, it is not an error for compilation. Synthesis tools sometimes warn the users. The linting tools also detect this condition, and report this to the user.

```
wire a, b, c;

assign b = (a == 1) ? c : 1'bz;
```

5.1.6 Illustrate the side effects of not having a final else clause in an if-else construct.

In a combinatorial block, not having a final ***else*** clause would result in a latch when synthesized. This is a fully legal construct in Verilog, and would compile without error. For example,

```
reg tmp;

always @(enable, in1) begin
```

```
  if (enable)
    tmp = in1;
  end
```

In the above code, there is no ***else*** clause in the combinatorial block and would result in a latch when synthesized. Many synthesis and linting tools detect and report this very well. If the latch is not to be inferred, then an else block is required.

If the intent is to produce a latch, any synthesis and lint warnings are false. The warnings can be avoided by using the ***always_latch*** keyword in SystemVerilog as follows:

```
always_latch
  if (enable)
    tmp <= in1;
```

5.1.7 What is the side effect of not having a default clause in a case construct

This is another common reason for the cause of un-intentional latches. The default case is necessary if all the cases are not fully specified. For example,

```
module lower (in1, in2, opcode, out1);
input  [1:0] in1, in2, opcode;
output [1:0] out1;

reg [1:0] out1;

always @(in1 or in2 or opcode) begin
  case (opcode)
    2'b00 : out1 = in1 & in2 ;
    2'b01 : out1 = in1 | in2 ;
    2'b10 : out1 = in1 ^ in2 ;
//  2'b11 : out1 = in1 % in2 ;   // uncommenting
                                 // either of these two
//  default : out1 = in1 & in2;  // lines will avoid
                                 // a latch
    endcase
  end
```

```
endmodule
```

The presence of the ***default*** statement will initialise the `out1` variable for all definitions of case items that are not be specified. Good linting tools specify the presence of the latches, and the synthesis elaboration log file also needs to be checked for the presence of latch inference.

5.1.8 Illustrate example of how unintentional deadlocked situations can happen during simulation.

When there are interactions between two processes in a handshake interface form, it is important to ensure that the implementation doesn't allow itself to become deadlocked. The deadlock situation is one in which one process is waiting for the other process to enable it, which in turn will enable the source process. The code could be a syntactically correct implementation, and still have a deadlock situation. The scenario can happen in both synchronous and asynchronous designs. A simple asynchronous example has been illustrated in the following, to demonstrate how deadlock occurs.

```
module deadlock;

reg a, b, c;

initial begin
  a <= 0; // source of deadlock
  b <= 0;
  wait(b == 1'b1);
  $display ("Variable b detected as 1");
end

always @(a) begin
  if (a == 1'b1) begin
    $display ("Variable a detected as 1");
    b = 1;
    $display ("Variable b assigned to 1");
  end
end

endmodule
```

In the above example, the displays of variable `a` and `b` being detected will never get asserted, since the variable `a` has been initialised to 0. In the above example, if the variable `a` gets initialised to 1, then all the displays get asserted. The above example is an illustration of the deadlock scenario, which can be difficult to capture in a larger implementation.

SystemVerilog eliminates the race condition with ***always_comb,*** which automatically triggers once at time 0, after all pending time 0 assignments are executed.

5.1.9 Having a programmed loop that does not move simulation-time

When any form of loop executes without any delay in between iterations, the code without a defined termination criteria would make the simulator hang at that loop. The delay can be either through a (***posedge*** `clk`), or # delay constructs. The simulation time also doesn't move ahead, due to this issue since there is no advancement of time. For example, in the following, the ***while*** loop runs in 0 time delay between iterations forever, and would cause a hang.

```
module while_hang;

initial begin
  while (1) begin : loop_while
  // No construct to advance the time
  end // loop_while

$finish; // this line is never reached, and
         // the simulation hangs
end

endmodule
```

These kind of zero delay loop is only a problem if another process is reading the variables assigned in the loop. In the above example, the variables assigned in the loop might not have a chance to propagate to the DUT. This causes a write-write .. write-read race condition. This scenario typically occurs in a testbench.

One of the common workarounds to detect such unintentional hang scenarios is by introducing a check in difference in time between iterations just before the end of the loop. If the time didn't advance, then the loop should be exited. The following is an extension of the above example with the time check between the iterations of the loops:

```
module test_repeat;
integer i;

time intime, outtime; // time tracking variables

initial begin
  while (1) begin : loop_while
    intime = $time;  // beginning of the loop
// @(posedge clk)  // time advancer
//...  other statements within the while loop
    outtime = $time; // end of the loop
    if (intime == outtime) begin
      $display("Hang detected: intime = %0d, \
                outtime = %0d",intime, outtime);
      break; // SV feature
    end
  end
end

endmodule
```

Having checks like the above in loops that are suspicious of being hung will help in easy debugging.

5.1.10 Illustrate the side effect of leaving an input port unconnected that influences a logic to an output port

Leaving an input pin floating will cause a `bz to be propagated during functional simulation. During synthesis, it will cause the gate optimiser to optimise away the logic that propagates beyond a floating input. For example,

```
module lower (in1, in2, out1, out2);

input  [7:0] in1, in2;
output [7:0] out1, out2;

assign out1 = in1 & in2;
assign out2 = in1 | in2;

endmodule // lower

module upper1 (u_in1, u_in2, u_out1, u_out2);
input  [7:0] u_in1, u_in2;
output [7:0] u_out1, u_out2;

reg [7:0] reg1;
wire [7:0] wire1;

// Instantiating lower with in1 floating
lower U1 (
  .in1  (),  // input is left floating
  .in2  (u_in2),
  .out1 (u_out1),
  .out2 (u_out2)
);

endmodule // upper1
```

The above logic would get synthesized, such that the `u_in1` is not connected to any logic within its hierarchy, and `in1` is directly connected to `out2` in the `lower` hierarchy, (since `out2` is an or'ing function of `in1` with 'nothing').

5.1.11 Illustrate the side effect of not connecting all the ports during instantiation

Unconnected input ports evaluate to a '`z`'. If the input port is used in ***if*** conditions with logical equality operator (==), then the condition evaluates to a logical false. For example,

```
module mod1 (in1, en);
input  in1;
input  en;

always @(in1, en) begin
  #5 if (en == 1'b1)
     $display ("Reached then");
  else
     $display ("Reached else");
```

```
end

endmodule // mod1

module mod1_top;

reg in1;

initial in1 = 1'b1;

// Instantiating mod1 module and port en is
// left floating

mod1 U0 (
  .in1 (in1),
  .en ()
);

endmodule // mod1_top
```

The above example will display “Reached else”, since there was nothing connected to port en. Most of the simulators issue a warning message if input ports are unconnected or left floating.

The above mis-connections can be detected through a monitor module that would be peeking at all the inputs and outputs of the DUT. An example snippet of code to detect the floating inputs and outputs is:

```
module mod1_mon(in1, en);
input in1, en;

initial begin
  if (in1 === 1'bz)
    $display("in1 is seen floating at t=%0d",
             $time);
  if (en === 1'bz)
    $display("en is seen floating at t=%0d",$time);
end

endmodule // mod1_mon
```

Note that the check for floating input is done once through an ***initial*** block, since the state of the input is not expected to change dynamically. If the scenario of intentionally floating the input is necessary, then the above code needs to be placed in an ***always*** block. The above monitor module can be instantiated within the `mod1_top` module as:

```
mod1_mon U2 (
  .in1 (in1),
  .en (en)
); // mod1_mon
```

With the `en` input left floating in the testbench, the display outputs of the above are:

```
en is seen floating at t=0
Reached then
```

5.1.12 Illustrate the side effect of forgetting to increase the width of state registers as more states get added in a state machine.

Normally the width of the state register is the closest power of 2, that is, for a 5 state state-machine, the state register would be [2 : 0] or 3 bits wide. The state variables would be from 3'b000 to 3'b111.

As the number of states in the state machine increase and go past 3'b111, the width of the state variable also needs to be increased to [3:0], etc. Suppose the additional states were having values like 4'b1000, 4'b1001, etc., and the width of the state register remained at [2:0]. This would erroneously truncate to 3'b001 for the state value of 4'b1001, and to 3'b101 for the state value of 4'b1101.

It is syntactically correct to have a smaller width of the state variable register and larger values of the state variables, and would not cause any error during compilation. But, this would lead to functionally incorrect results. Some of the good linting tools would catch this type of problem.

SystemVerilog has a new feature of enumerated state variables that will help in resolving this issue. The keyword ***enum*** is used for this purpose, which both assigns and increments the new variables added into this. The following example illustrates the use of this feature:

```
module enumfm (clk, reset_n, rd_n, ready, done,
               out1);

input clk, reset_n, rd_n, ready, done;
output out1;

enum {idle, read, write, wait4rdy} current_state,
next_state;

always_ff @(posedge clk or negedge reset_n)
  if (reset_n == 1'b0)
    current_state <= idle;
  else
    current_state <= next_state;

always_comb
  begin
  next_state = current_state;
  case (current_state)
    idle: if (~rd_n)
            next_state = read;
          else
            next_state = write;
    read: if (!ready)
            next_state = wait4rdy;
          else if (done)
            next_state = idle;
    write: if (!ready)
            next_state = wait4rdy;
          else if (done)
            next_state = idle;
    wait4rdy: if (ready & ~rd_n)
            next_state = read;
          else if (ready & rd_n)
            next_state = write;
    default: next_state = idle;
  endcase
  end

assign out1 = (((current_state == read) ||
                (current_state == write))
             && ready);
```

```
endmodule // enumfm
```

In the above example, the state variables `current_state` and `next_state` are declared through ***enum.*** Without any assignments, the simulator and synthesis tools will assign the values linearly, incrementing with value of idle=0, read=1, write=2, wait4rdy=3, etc. As more states get added to this, the values simply increment.

5.1.13 Illustrate the side effect of an implicit 1 bit wire declaration of a multi-bit port during instantiation.

This is a common problem seen during connecting blocks with multi-bit port sixes. Since Verilog has the feature to define implicit 1 bit wires during port connections, the multi-bit port will be connected to single bit wires. It is a WARNING and not ERROR during compilation for most of the simulators. If the WARNING messages are turned off, or if there are too many of these WARNING messages, this issue can go undetected at the simulator level, and become an error during functional simulation. For example,

```
module lower (in1, in2, in3, out1, out2);

input  [7:0] in1, in2, in3;
output [7:0] out1, out2;

assign out1 = in1 & in2;
assign out2 = in1 | in2;

endmodule // lower

module upper1 (u_in1, u_in2, u_out11, u_out12,
               u_out21, u_out22);
input  [7:0] u_in1, u_in2;
output [7:0] u_out11, u_out12;
output [7:0] u_out21, u_out22;

reg [7:0] reg1;
wire [7:0] wire1;

// Instantiating lower
lower U1 (
  .in1 (u_in1),
  .in2 (u_in2),
  .in3 (in3),     // causes an implicit 1 bit wire
//.in3 (wire1),   // genuine 8 bit wire
  .out1 (u_out11),
  .out2 (u_out12)
); // lower

// Instantiating lower
lower U2 (
  .in1 (u_in1),
  .in2 (u_in2),
  .in3 (in3),     // causes an implicit 1 bit wire
//.in3 (wire1),   // genuine 8 bit wire
  .out1 (u_out21),
  .out2 (u_out22)
);

endmodule // upper1
```

This issue is detected as an ERROR by the synthesis tools. Also, some of the editors, like EMACS, can be commanded to declare the intermediate wires of the correct width.

SystemVerilog provides enhancements that can prevent this implicit 1-bit wire error. The implicit named port connection during module instantiations will not permit connections where the net is a different size than the port.

5.1.14 Same variable used in two loops running simultaneously

Sometimes accidentally, a loop variable is used in two different blocks (often the ***for*** loops), and would be modified in both places. Although this is syntactically correct, it would cause functional problems. For example, in the following code, the same variable, "i", is being modified in two different loops, which could be difficult to detect in large pieces of code:

```
module twoloops;

integer i;

initial begin // start at time 0
  for (i=0; i <= 7; i = i + 1) begin
    #5 $display("Entered 1st loop at t=%0d, i =
                    %0d",$time, i);
  end
end

initial begin // start at time 0
  for (i=0; i <= 7; i = i + 1) begin
    #2 $display("Entered 2nd loop at t=%0d, i =
                    %0d",$time, i);
  end
end

endmodule // twoloops
```

The output of the above code produces displays as:

```
Entered 2nd loop at t=2, i = 0
Entered 2nd loop at t=4, i = 1
Entered 1st loop at t=5, i = 2
Entered 2nd loop at t=6, i = 3
Entered 2nd loop at t=8, i = 4
Entered 1st loop at t=10, i = 5
Entered 2nd loop at t=10, i = 6
Entered 2nd loop at t=12, i = 7
Entered 1st loop at t=15, i = 8
```

Note that the iteration from 0-8 is shared between the two loops. In a few rare occasions, this could be a genuine requirement in behavioural coding, in which case the user needs to verify that the intended functionality is correctly met.

However, SystemVerilog has a good feature of declaring the variable *within* the ***for*** loop so that the variable is *local* to that loop only. The same for loop in the above example, in SystemVerilog would be:

```
module twoloops;

// integer i; // is now local within the for loops

initial begin // start at time 0
  for (int i=0; i <= 7; i = i + 1) begin
    #5  $display("Entered  1st  loop  at  t=%0d,  i  =
%0d",$time, i);
  end
```

```
end

initial begin // start at time 0
  for (int i=0; i <= 7; i = i + 1) begin
    #2  $display("Entered  2nd  loop  at  t=%0d,  i  =
%0d",$time, i);
  end
end

endmodule // twoloops
```

The output of the above SystemVerilog code would produce all the 8 iterations from **both** loops as follows:

```
Entered 2nd loop at t=2, i = 0
Entered 2nd loop at t=4, i = 1
Entered 1st loop at t=5, i = 0
Entered 2nd loop at t=6, i = 2
Entered 2nd loop at t=8, i = 3
Entered 1st loop at t=10, i = 1
Entered 2nd loop at t=10, i = 4
Entered 2nd loop at t=12, i = 5
Entered 2nd loop at t=14, i = 6
Entered 1st loop at t=15, i = 2
Entered 2nd loop at t=16, i = 7
Entered 1st loop at t=20, i = 3
Entered 1st loop at t=25, i = 4
Entered 1st loop at t=30, i = 5
Entered 1st loop at t=35, i = 6
Entered 1st loop at t=40, i = 7
```

Some of the considerations in using a variable declaration within the ***for*** loop are:

- The local declarations of the variables within the ***for*** loop cause the variable to have ***automatic*** properties, that is, will not be overwritten when used in multiple loops, as illustrated in above example.
- The loop variable is visible only within the ***for*** loop, and not outside it. If the variable needs to be accessed outside the ***for*** loop, it must be declared explicitly outside the loop

5.1.15 Illustrate the side effects of multiple processes writing to the same variable.

When the same variable is assigned in two different processes, it not only creates race conditions during simultaneous assignments, it becomes non-synthesizable. Most of the simulators allow compilation to proceed, since it is syntactically correct. For example,

```
module blknonblk (clk, in1, in2, in3, out1);
input clk,in1, in2, in3;
output out1;

reg reg1, reg2, reg3, out1;

always @(posedge clk) begin : proc1
  reg1 <= in1;
  out1 <= reg1 & reg2;
end

always @(in1 or in2) begin : proc2
  reg1 = in3;
// reg1 is assigned in proc1 and proc2. Incorrect.
  reg2 = in2;
end

endmodule
```

Most of the linting tools are able to detect this, and is also a compilation error seen during synthesis.

5.1.16 Illustrate the side effect of specifying delays in assignment's.

Specifying any kind of delay before an assignment, or within an assignment, in a blocking or nonblocking procedural assignment is ignored by synthesis tools. If the functionality depends upon the presence of the delay, then a mismatch in functional simulation will be seen between the model and the synthesized netlist. For example,

```
reg1 = #3 reg2;    // #3 will be ignored
#6 reg3 <= reg4;   // #6 will be ignored
```

Since the above construct is syntactically legal, the synthesis tools will issue a WARNING and not an ERROR.

SUMMARY

This chapter discussed the common functional mistakes that happen during the coding using Verilog. Most of these errors go undetected, as they will be syntactically correct, and, hence, get past the compilation. While many of these are un-intentional errors, a preview of these scenarios will help the readers towards debugging more easily in the different stages of the project cycle. Any workarounds that could help in avoiding these mistakes have also been discussed.

Chapter 6

VERILOG DURING SIMULATION REGRESSIONS

INTRODUCTION

This chapter discusses using Verilog for regression. testing In particular, we discuss the requirements for pre-release regression testing, and the issues encountered during such regression simulations. As the design sizes continue to grow, so does the complexity of verification and the regression runtime. We discuss some special constructs of Verilog that help in meeting some of the needs of pre-release regression simulations. In most cases, Verilog alone is not sufficient in constructing the regression infrastructure. Regression environments are typically wrapped in programming languages like C, and scripting languages like Perl, TCL, make or csh. Scripting languages typically constitute the control/logistical flow of the regression environment. This chapter discusses specifically how these logistics can be aided in implementing the release infrastructure efficiently, and how the constructs in Verilog help in achieving the same.

The development of a good regression environment is not something that can be deferred until the end of code development and testing. It is an essential infrastructure that needs to be built right at the architectural definition phase. This way, the changes done during the development phase can also make use of this infrastructure to validate the changes.

6.1.1 Illustrate a few important considerations on simulation regressions, and how Verilog can be useful for achieving the same.

While the regression environment is quite unique to each product, the following are a few generic requirements that are useful for a release simulation:

1. The user must have provision to turn off the waveform dumps during the regression. This will not only save disk space, but also improve the overall runtime. In Verilog, this can be controlled by a variety of ways, as illustrated in the following:

   a. Controlling the dump operation via a command line argument. This can be implemented through the ***`ifdef*** compiler directive and the ***$dumpvars*** system task. An example to illustrate the same is:

```
`ifdef DUMP_ON
   initial $dumpfile ("waves.dump");
   initial $dumpvars;
`endif
```

   During the command invocation, the following needs to be appended at the end:

```
+define+DUMP_ON
```

   b. In the above method, either the waveforms of all the variables from/below the hierarchy from where the ***$dumpvars*** command is invoked are recorded, or there is no waveform recording at all. If the waveforms need to be dumped at specific hierarchical levels, this can be controlled by specifically mentioning the hierarchy in the first argument of the ***$dumpvars*** command. For example,

```
$dumpvars(0, testtop.U1);
```

   This specifically dumps all hierarchies of `U1` at and under `testtop` only, and no other modules.

```
$dumpvars(1, testtop);
```

This specifically dumps all hierarchies at `testtop` only, that is, level 1, and none below level 1.

Note that more than one module can be mentioned after the first argument, for specifying additional hierarchies to be dumped.

The value of the depth can also be passed from the command line argument, using the Verilog ***$value$plusargs*** construct. An example of how to use this is explained later this chapter.

c. In the above method, the dumping of the waveform happens right from time 0 onwards, until the end of simulation. Sometimes it is necessary to capture the waveform only for a small window of simulation. This could be the duration in the zone of interest in the entire simulation. Verilog provides a mechanism to capture a specific window, too, using the ***$dumpon*** and ***$dumpoff*** commands, as illustrated in the following example:

```
module test_dumping;

reg clk;

initial begin
  clk=0;
  $dumpvars;
  $dumpoff;
#50;
  $display("Dumping ON at t=%0d",$time);
  $dumpon;
#100;
  $display("Dumping OFF at t=%0d",$time);
  $dumpoff;
#75;
  $display("Simulation   actually   ends   at
t=%0d",$time);
  $finish;
end

initial begin
  forever clk = #5 ~clk;
```

```
end

endmodule // test_dumping
```

If the dump file is viewed through a waveform viewer, it will be evident that there was no dumping until time unit 50, and after 150, until the end of simulation.

This approach is useful for one other purpose, too, as follows:

- In long simulation runs, the user is interested to see the dump only for a few transactions/scenarios before the erroneous time stamp. During the next iteration of debugging, it is useful to specify the ***$dumpon*** at a timestamp a few appropriate transactions before the timestamp of the bug/error scenario in simulation. The bug/error timestamp will be known during the previous iteration. This will help in loading the dump files **faster** in the waveform viewers, rather than the large dump files.
- This approach also helps in creating smaller dump files for debugging and, hence, lessen disk space.
- Many times, in order to update the product specifications, that is, either the functional specification or the user documentation, it is necessary to add timing diagrams. In that case, the transactions can be run on the DUT, and the VCD dump can be captured for the zone of window, depicting the transaction scenario with the signals of interest. This can capture the waveform for that specific simulation time, and be useful for the waveform capturing into the product specifications.

d. During regressions, it is necessary to store the dump files of the runs *optionally.* For example, if a test PASSED in a self-checking testbench, it is very unlikely that the dump file is further required for debugging purposes, other than viewing the waveform for an explicit check. In that case, it is not required to have the dump file of a passing test, which would unnecessarily occupy large disk space. In such a case, the dump file can then be conditionally deleted, or retained if the test did not pass. This feature can be made configurable through specially implemented configuration commands, or by passing specific arguments from command line.

2. The provision to generate or not generate the log files (transcripts of the simulation run) should be controllable at the command invocation level, or by specifying this in as a specialized configuration command, or as a parameter in an ***`include*** before the simulations begin. Just like the waveform dump files scenario explained above, the log files sometimes take a lot of disk space and runtime. These factors will not be significant if one or two tests are run. But these add up when there are several thousand tests to be run. The following are a few ways to control the dumping of the log file:

   a. For command invocation, the mechanism is exactly the same as described in the above description for the dumping of waveform. This mechanism is usually the most convenient.

   b. For non-command invocation control, the facility to dump or not dump could be controlled by a user defined configuration command `configure`, which is basically a Verilog ***task*** for the user. For example,

   ```
   configure (log_file_dumping, true);
   ```

   This can be used within the messaging commands to use ***$display*** or ***$fdisplay***, depending upon whether the `log_file_dumping` is false or true. For example,

   ```
   task display_msg( …)
     if (log_file_dumping) begin
       $fdisplay(int_log_file, <string>);
     end else begin
       $display("<string>");
     end
   endtask
   ```

3. The provision to add/modify/delete the tests must be possible to be done easily by the user, and preferably modifying just one file. Typically, during the development phase, the regressions can fail on a particular tests. In that case, the test infrastructure should have the facility to run just a single test case, or a subset of testcases, with ease, typically by just modifying the list of testcases from a file. The

other way could be to specify all the subset of tests to be run in the command line argument itself. While the latter is okay for a smaller number of tests, with small names of the tests, it may not work in many shells with limited capacity of the characters on the command line. When the file approach is used for a group of tests, a special file parser is required for this purpose, and invokes the tests mentioned in each line.

An example of the file parser approach is discussed in FAQ 6.1.2 of this book.

4. If multiple CPUs are present, then the provision to schedule these optimally for the regression purposes should be used. This is not a Verilog feature, but a useful functionality done by batch-scheduling software. This way, the sessions get queued on multiple CPUs and each CPU is used optimally, to complete a given sub-task of the full run. Batch-scheduling software is also useful when the number of licenses of the tools is limited, and requires the jobs to be sequential, based on availability of the licenses.

5. The provision to display the results of the full regression must be available to the user at any time during the regression. This should not be considered a post-processing task at the end of a long simulation. This will help in providing the user feedback early on if there is a problem in the regression. This will help decide to terminate the regression, if it is not worthwhile to proceed further. The resolution of the results should be at a single test level, and can be until the last test completed. Some of the key features to be displayed are:

    a. Hostname of the machine where the test was run
    b. Number of tests passed
    c. Number of tests failed
    d. Number of tests timed out
    e. Seed value of the test run (in case of random testing)
    f. Date/time test started
    g. Date/time test ends
    h. Total number of tests run
    i. List the tests passed/failed/timeout into separate files
    j. If Failed, what was the string of failure, that is, data mismatch?

k. Which was the source/destination that encountered the failure, that is, whether the DUT was involved, or was the error between agents involved in a traffic test without DUT?
l. Runtime taken for each test and an accumulated summary
m. If memory is critical, then the memory required for the tests

The mechanism to display the summary can be done through a Perl script or csh script, or even through Verilog, through the file I/O capabilities. The file parsing/interpreting mechanism discussed in the earlier sections can be useful for this. As an extension, it would be useful to display the result in a HTML format, which will be useful for all interested team members for viewing the results through a web browser.

6. Sometimes it is useful to pass the name of the test as an argument to the simulation session. This will be useful to print the test name during some print messages, or to create the log file based on the name of the test. The Verilog ***$value$plusargs*** command line inputs can be used for this purpose. This system function searches the command line argument for certain patterns, and assigns the value to the pattern into a variable within the testbench. Note that you can give multiple such command line inputs, and the unique string will assign the destination variable. For example,

```
module tmp_task ;
reg [8*31 : 0] test_name;
initial begin

if   ($value$plusargs("test_name=%s",   test_name))
begin
  $display("Starting test %0s at %0d",test_name,
                                      $time);
end

end
endmodule
```

When the above module is simulated with the command line input of:

```
% <simulator_executable> +test_name=mytest
```

the output produced is:

```
Starting test mytest at 0
```

Similar to the above, multiple such arguments can be *communicated* into the testbench environment. The command line arguments can be passed from a wrapper script that launches these tests.

Note that, if the sufficient number of ***$value$plusargs*** are not defined in the testbench, the excess arguments will be ignored. That is, if the testbench code has implemented checks for 4 plusargs, and additional plusargs are mentioned in the command line, they get ignored by the testbench.

7. All the inputs to the regression should be checked before the launch of the long regression. These inputs could be any or all of the following variables:

   - Parameter values for the regression have to be specified. These values can be specified through ***$value$plusargs,*** as explained earlier, or through the parameter files. If a specific parameter is not specified, then a default value should apply.
   - Presence of sufficient number of tool licenses to launch single or parallel runs. This check needs to be done by the launching script, whether it is in PERL or TCL or csh.
   - Checking availability of system memory for the regression. This check needs to be done by the launching script, whether it is in PERL or TCL or csh.
   - Checking availability of disk space, considering the outputs of the log and dump files that gets produced during the regression. This check needs to be done by the launching script, whether it is in PERL or TCL or csh.

8. Since the product could be implemented in multiple platforms too, it is necessary to involve the scripts for multiple platforms, like Solaris, Linux, HP-UX, etc. As a first order requirement, simulation scripts should be platform-independent to the extent possible. If platform-specific constructs are used in scripts, then these should be multiply customized for all the supported platforms. The script should be intelligent enough to detect the platform on which it is being run. Sometimes, the version number of the Operating System (OS) also matters, as also the necessity for certain patches. The script should automatically detect the platform and the version number, and do everything appropriate to the particular platform of execution.

9. If the design has multiple parameters, it is required to run the regression across multiple parameters, which influence the functionality of the DUT dramatically. Examples of such parameters include varying bus widths, varying clock frequency ratios between externally accessible clock domains, endian of the data path, widths and depths of FIFOs used, etc. The scripts should have the capability to cycle through all legal combinations of parameters, and run simulations for each combination.

10. There should be facility within the regression to switch between automatic command generation, using random stimulus generation, and executing specific sequences of commands, using a directed flow. This should preferably be controlled by a single flag, through a specialized user defined ***task*** like `configure`. For example,

    ```
    configure (auto_command_generation, `true);
    ```

    This will cause the automatic command generation through random command stimulus generation until all its constraints are met. The flag can be set to `` `false `` by default. The objective of this mechanism is that there should eventually be only one testbench that switches between the directed and random command generation with ease. The user shouldn't have to maintain two testbenches, that is, one for random stimulus generation only, and one for directed stimulus generation flow.

11. During regressions when all other messages except ERROR/FATAL are disabled during the runtime, it is useful to get some "heartbeat" message, to know that the simulation is still in progress, and not stuck at some point. This needn't be for every transaction, but it is found to be good enough if a heartbeat is seen, for example, every 10 transactions. At such instants, it would be useful to optionally display a message like:

    ```
    Simulation in progress after 10 transactions at t=100
    Simulation in progress after 20 transactions at t=200
    ...
    ```

    A simple logic to implement the above is illustrated below:

```
module heart_beat;
```

```
parameter max_timeout_limit = 500;

reg clk, start_addr_ph;
integer num_xns, timeout_track;

initial begin
  clk = 0;
  num_xns = 0;
  start_addr_ph = 0; // testing purpose only
  forever clk = #5 ~clk;
end

initial begin
  $dumpvars;
//  #1000 $finish;
end

// this always block detects addr phase every
// alternate clock. This is for illustration only.
// Actual DUT may give addr phase at varied clock
// intervals

always @(posedge clk) begin
  start_addr_ph <= ~start_addr_ph;
end

always @(posedge clk) begin
  if (start_addr_ph) begin
    num_xns <= num_xns + 1;
    timeout_track <= 0;
    if (((num_xns % 10) == 0) & (num_xns !== 0))
      $display ("Simulation in progress after %0d
transactions",num_xns);
  end else begin
    timeout_track <= timeout_track + 1;
    if (timeout_track == max_timeout_limit) begin
      $display ("WARNING : Unusually long time of %0d\
                clocks taken after start_addr_ph. \
                Maybe the test hung. TIMEOUT",\
                max_timeout_limit);
      $display ("INFO : Executing $finish in the file\
                ./testbench/test_top.v");
```

```
        $display ("INFO : Change the max_timeout_limit \
                  parameter in this file to increase \
                  the limit");
      end
    end
  end

  endmodule // heart_beat
```

A few salient points regarding the above example are:

- The value of timeout can be changed through the parameter to the appropriate acceptable limit.
- In the above example, the heartbeat rate is 10 transactions. It can also be easily changed to 20, or 50, or to any preferred value.

12. A constant monitoring by the various bus monitors in the testbench should be incorporated. The monitors can *communicate* to the testbench by assertion of an output port, or by setting of a flag in the testbench, or by the testbench having access to these flags within the monitors. In the following example, two different monitors, with instance names `U1` and `U2`, are monitoring the bus activity in the testbench with their output ports `status`:

```
wire [1:0] status1, status2;
mod1_mon U1 (
  .in1 (in1),
  .en (en),
  .status (status1)
);

mod1_mon U2 (
  .in1 (in1),
  .en (en),
  .status (status2)
);

assign error_det = ((status1 == 2'b11) |
                    (status2 == 2'b11));
```

The outputs of the multiple monitors are OR'ed to assert a critical error_det signal in the testbench. This can cause the simulation to terminate with a ***$finish***, as illustrated in this simple example:

```
  always @(posedge clk)
  begin
    if (error_det) begin
      $display("Error detected in testbench at
t=%0d");
      if (stop_on_error) begin
        $display("Terminating regression");
        $display("Test FAILED");
        $finish;
      end
    end
  end
```

13. The error messages should be identifiable, whether it is caused by an intentional error, or not. Many times, post processing of the log files is done, to search/grep for strings like ERROR. In order to distinguish between an intentional error and a real error, there should be suitable string displayed out before the launch of the intentional error. This will help in isolating any false alarm during the post processing of these error messages. The assertion of the global error flags or signals, as explained earlier, can be gated with the unintentional error criteria.

14. It is useful to inform the user, through INFO messages, as to where these values can be changed, and which module caused the exit of simulation. When the regressions are run during the development phase, it is likely that some tests will hang. There should be some kind of logic to detect this hang situation, and have a graceful exit in the form of a TIMEOUT. This has been illustrated in the example above.

15. If more than one copy of licenses of the simulation tool is present, it is useful to plan the regression, such that the execution happens in different directories. Typically, the launch of the simulation happens from a common directory like the .../sim (or its equivalent directory name in your project). Trying to run multiple runs from the same directory could cause dump, log, or any other output files to be overwritten. In order to avoid such potentials of overwriting to

happen, it is good to design the execution of the runs from different directories. This will help in multiples simulation runs to take place in parallel.

6.1.2 What coding constructs of Verilog can be used during the various stages of designing a regression environment for simulations?

Verilog has the following constructs built-in, which help in the regressions:

1. ***$readmemh/$readmemb*** : These constructs help in loading memory data from a file. These constructs will be useful during microprocessor simulations, or in supplying vectors for a DUT, or simply customized commands encoded into the various fields of the line. Examples to illustrate the above have been discussed earlier in an earlier FAQ 3.5.1 in the Verification chapter.

2. Sometimes, a file parsing is required, to know the arguments from certain lines of a file. Instead of writing a PLI just for this purpose, the Verilog language provides the ***$fscanf*** system task inbuilt. This function scans the lines that it reads sequentially, until a carriage return is obtained, and assigns the values of the arguments to the destination variables. Each argument is typically separated by a white space within the input file. This is a very easy way to specify the tests to be run within a file, along with its associated arguments.

   For example, in the following code, the `infile.txt` is a file containing three fields. The `arg1` is a decimal variable, `arg2` is string, and `arg3` is hexadecimal. The ***$fscanf*** task scans this file, all the way until the EOF is reached, and currently displays what it sees. It can be modified further into the different simulation launching sessions.

```
module tmp_task ;

reg [8*31 : 0] arg2;
// arg2 can be string variable in SystemVerilog
reg [7:0] arg1, arg3;
integer infile;

initial begin
```

```
  infile = $fopen("infile.txt", "r");

  while ($fscanf (infile, "%d %s  %h", arg1, arg2,
                  arg3))
  begin
    $display("arg1 = %0d,\targ2 = %0s,\targ3 =
               %0h",arg1, arg2,arg3);
  end

  $fclose(infile);

end // initial
endmodule // module
```

Suppose the infile.txt file contained the following:

```
10 test1 0a
11 test2 0b
12 test3 0c
13 test4 04
```

The following would be the output of the above code:

```
arg1 = 10,    arg2 = test1,  arg3 = a
arg1 = 11,    arg2 = test2,  arg3 = b
arg1 = 12,    arg2 = test3,  arg3 = c
arg1 = 13,    arg2 = test4,  arg3 = d
```

In the same way as illustrated above, the `arg1,2,3` variables can actually be used internally for test launching and initialisation purposes.

3. Some environments use a "reference-file" based approach for the vector comparison for PASS/FAIL criteria of a release regression. This is typically useful for a multi-simulator product regression. This method is useful, since it is HDL independent output for comparison of the responses across the simulators of Verilog. The ***$fstrobe*** command can be used for this purpose. An example to illustrate this is as follows:

```
module test_fstrobe;

parameter cycle_time = 10;
parameter clk2q_time = 9;
integer out_vec_file;
reg clk;
wire out1;
wire out2;
wire [3:0] out3, out4;
reg [3:0] count;
reg capture_vec;

initial begin
  out_vec_file = $fopen ("out_vec_file", "w");
  forever # (cycle_time/2) clk = ~clk;
end

always begin
  if (capture_vec) begin
    # clk2q_time;
    $fstrobe (out_vec_file,"%0t %b %b %0h %0h", $time,
                out1, out2, out3, out4);
  end
  @(posedge clk);
end

initial begin
  capture_vec = 1;
  count = 0;
  clk = 0;
  #50 capture_vec = 0;
  $finish;
end

always @(posedge clk) begin
  count <= count + 1;
end

assign out1 = count[0];
assign out2 = count[2];
assign out3 = count;
assign out4 = ~out3;

endmodule // test_fstrobe
```

The output of the above code produces the file `out_vec_f ile`, with contents as:

```
14 1 0 1 e
24 0 0 2 d
34 1 0 3 c
44 0 1 4 b
```

The disadvantage of the vector-based approach is the fact that the vectors do not carry information of functional correctness. It is useful if the vectors have been inspected a priori by some other method, and declared to be "golden". Another disadvantage of this method is the following: If a design is changed during the course of its life-cycle (as for example, for bug-fixes or enhancements), then, depending on the nature of the changes, the originally captured golden vectors may no longer be valid. They would need to be re-captured, re-inspected and re-certified to be "golden". The vector doesn't carry information of functional correctness, except for clock cycle accurate reproduction of response, provided it has been verified once before.

4. When the same testbench is being used for multiple configurations of the DUT, that is, for running RTL simulations, Gate level simulations, or the behavioural model simulation, the instantiation of the DUT must be easily selectable from the command line itself. This can be achieved by the use of the command line argument of ***$value$plusarg.*** An example to illustrate this is as follows:

```
   `ifdef RTL
     initial $display("Instantiating RTL\n");
   // Instantiate the top level of RTL module
   `endif
   `ifdef GATE
     initial $display("Instantiating Gate level
netlist\n");
   // Instantiate top level of Gate level netlist
   `endif
   `ifdef BEHAV
    initial $display("Instantiating Behavioral
level\n");
   // Instantiate top level of Behavioral level
   `endif
```

During the simulation invocation command line, the following can be appended:

```
% <simulation invocation> +define+RTL
% <simulation invocation> +define+GATE
```

Note that the Gate level simulations could be slow, due to the presence of system timing check commands, like ***$setup*** or ***$hold***, built in within the simulation models of the cells of the technology library. If running the gate-level simulation is a requirement, some Verilog simulators have switches that will ignore these timing checks, and only the functional simulations are run, to make sure the logic is okay. The timing checks are now being done more through Static Timing Analysis (STA).

SUMMARY

This chapter discussed how the Verilog constructs could be used for the product simulation regression purposes. The different Verilog constructs that influence the simulation during invocation and runtime have been illustrated. The chapter also discussed how the simulation session can influence the log and the dump file generation.

References

1. IEEE Std 1364-2001, Language Reference Manual (LRM)
2. IEEE Std 1364.1-2002 Standard for Verilog Register Transfer Level Synthesis 2002
3. SystemVerilog 3.1a Language Reference Manual
4. SystemVerilog Synthesis User Guide : Version V2003-12 Dec 2003 from Synopsys Inc
5. The Verilog Hardware Description Language, 5th Edition by Donald Thomas and Philip Moorby
6. Reuse Methodology Manual by Michael Keating and Pierre Bricaud, third edition
7. Verilog 2001: A Guide to the New Features of the Verilog Hardware Description Language by Stuart Sutherland
8. Writing Testbenches: Functional Verification of HDL Models, Second Edition by Janick Bergeron
9. Verilog HDL Synthesis : A Practical Primer by J Bhasker
10. http://www.eedesign.com/editorial/1997/test9708.html

Index