John Williams

Digital VLSI Design with Verilog

A Textbook from
Silicon Valley Technical Institute

foreword by Don Thomas

Springer

Digital VLSI Design with Verilog

John Williams

Digital VLSI Design with Verilog

A Textbook from Silicon Valley Technical Institute

Foreword by Don Thomas

Springer

Dr. John Williams
SVTI Inc.
Silicon Valley Technical Institute
1762 Technology Drive
San Jose CA 95110
Suite 227
USA
john@svtii.com

ISBN: 978-1-4020-8445-4 e-ISBN: 978-1-4020-8446-1

Library of Congress Control Number: 2008925050

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

To my loving grandparents,
William Joseph Young (ne Jung) and
Mary Elizabeth Young (nee Egan)
who cared for my brother Kevin and me
when they didn't have to.

Foreword

Verilog and its usage has come a long way since its original invention in the mid-80s by Phil Moorby. At the time the average design size was around ten thousand gates, and simulation to validate the design was its primary usage. But between then and now designs have increased dramatically in size, and automatic logic synthesis from RTL has become the standard design flow for most design. Indeed, the language has evolved and been re-standardized too.

Over the years, many books have been written about Verilog. My own, coauthored with Phil Moorby, had the goal of defining the language and its usage, providing examples along the way. It has been updated with five new editions as the language and its usage evolved.

However this new book takes a very different and unique view; that of the designer. John Michael Williams has a long history of working and teaching in the field of IC and ASIC design. He brings an indepth presentation of Verilog and how to use it with logic synthesis tools; no other Verilog book has dealt with this topic as deeply as he has.

If you need to learn Verilog and get up to speed quickly to use it for synthesis, this book is for you. It is sectioned around a set of lessons including presentation and explanation of new concepts and approaches to design, along with lab sessions. The reader can follow these at his/her own rate. It is a very practical method to learn to use the language, one that many of us have probably followed ourselves. Namely, learn some basics, try them out, and continually move on to more advanced topics which will also be tried out. This book, based on the author's experience in teaching Verilog, is well organized to support you in learning Verilog.

Pittsburgh, PA — Don Thomas
2008

Preface

This book is based on the lab exercises and order of presentation of a course developed and given by the author over a period of years at Silicon Valley Technical Institute, San Jose, California.

At the time, to the author's best knowledge, this course was the only one ever given which (*a*) presented the entire verilog language; (*b*) involved implementation of a full-duplex serdes simulation model; or (*c*) included design of a synthesizable digital PLL.

The author wishes to thank the owner and CEO of ***Silicon Valley Technical Institute***, Dr. Ali Iranmanesh, for his patience and encouragement during the course development and in the preparation of this book.

Contents

Introduction

1 Course Description

This book may be used as a combined textbook and workbook for a 12-week, 2-day/week interactive course of study. The book is divided into chapters, each chapter being named for the week and day in the anticipated course schedule.

The course was developed for attendees with a bachelor's degree in electrical engineering, or the equivalent, and with digital design experience. In addition to this kind of background, an attendee was assumed familiar with software programming in a modern language such as *C*.

Someone fulfilling the course requirements would expect to spend approximately 12 hours per week for 12 weeks to learn the content and do all the labs. Of course, now that it is a book, a reader can expect to be able to proceed more at his or her own pace; therefore, the required preparation can be more flexible than for the programmed classroom presentation.

Topic List (partial):

Discussion: Modules and hierarchy; Blocking/nonblocking assignment; Combinational logic; Sequential logic; Behavioral modelling; RTL modelling; Gate-level modelling; Hardware timing and delays; Verilog parameters; Basic system tasks; Timing checks; Generate statement; Simulation event scheduling; Race conditions; Synthesizer operation; Synthesizable constructs; Netlist optimization; Synthesis control directives; Verilog influence on optimization; Use of SDF files; Test structures; Error correction basics.

Lab Projects: Shift and scan registers; counters; memory and FIFO models; digital phase-locked loop (PLL); serial-parallel (and υ-υ) converter; serializer-deserializer (serdes); primitive gates; switch-level design; netlist back-annotation.

2 Using this Book

The reader is encouraged to read the chapters in order but always to assume that topics may be covered in a multiple-pass approach: Often, a new idea or language feature will be mentioned briefly and explained only incompletely; later, perhaps many pages later, the presentation will return to the idea or feature to fill in details.

Each chapter ends with supplementary readings from two highly recommended works, textbooks by Thomas and Moorby and by Palnitkar. When a concept remains difficult, after discussion and even a lab exercise, it is possible that these, or other, publications listed in the References may help.

2.1 Contents of the CD-ROM

The CD-ROM contains problem files and complete solutions to all the lab exercises in the book. A redundant backup of everything is stored on the CD-ROM in a tar file, for easy copying to disc in a Linux or Unix working environment.

Be sure to read the **`ReadMe.txt`** file on the CD-ROM before using it.

The **`misc`** directory on the CD-ROM contains an include file required for Lab 1, plus some other side files. It contains PDF instructions for basic operation of the VCS or QuestSim simulators.

The `misc` directory also contains nonproprietary verilog library files, written by the author, which allow approximately correct simulation of a verilog netlist. These netlist models are not correct for design work in TSMC libraries, but they will permit simulation for training purposes. **DO NOT USE THESE VERILOG LIBRARIES FOR DESIGN WORK**. If you are designing for a TSMC tape-out, use the TSMC libraries provided by Synopsys, with properly back-annotated timing information.

2.2 Performing the Lab Exercises

The book contains step-by-step lab instructions. Start by setting up a working environment in a new directory and copying everything on the CD-ROM into it, preserving directory structure.

The reader must have access to simulation software to simulate any of the verilog source provided in the book or the lab solutions. Readers without access to EDA tools can do almost all the source verilog lab simulations (except the serdes project) with the demo version of the Silos simulator delivered on CD-ROM with the Thomas and Moorby or Palnitkar texts. A more functional simulator is available from Aldec (student version). Verilog simulators also often are supplied free with FPGA hardware programming kits. Netlist simulations of designs based on ASIC libraries generally will require higher-capacity tools such as VCS or QuestaSim.

The Synopsys Design Compiler synthesizer and VCS simulator should be used for best performance of the labs. Almost all verilog simulator waveform displays

were created using the Synopsys VCS simulator, which is designed for large, ASIC-oriented designs.

Keep in mind that all verilog simulators are incomplete works: Different simulators emphasize different features, but all have some verilog language limitations. Attempts have been made in the text to guide the user through the most important of these limitations.

For the professional reader's interest, the CD-ROM labs in this edition were performed using Synopsys Design Compiler (Z-2007.03 SP2) and VCS (MX 2007) on a 1 GHz *x*86 machine with 384 megs of RAM, running Red Hat Enterprise Linux 3. TSMC 90-nm front-end libraries (*typical* PVT) from Synopsys were used for synthesis.

2.3 Proprietary Information and Licensing Limitations

Publishing of ***operational performance details*** of VCS, Design Compiler, or QuestaSim *may require written permission from Synopsys or Mentor*, and readers using the recommended EDA tools to perform the labs in this book are advised not to copy, duplicate, post, or publish any of their work showing specific tool properties without verifying first with the manufacturer that trade secrets and other proprietary information have been removed. This is a licensing issue unrelated to copyright, fair use, or patent ownership. Know your tools' license provisions!

Also, the same applies to the ***TSMC library files*** which may be available to licensed readers. The front-end libraries are designed for synthesis, floorplanning, and timing verification only, but they may contain trade secrets of TSMC. Do not make copies of anything from the TSMC libraries, including TSMC documentation, without special permission from TSMC and Synopsys.

The verilog simulation library files delivered on the accompanying CD-ROM resemble those in tool releases; however, they are not produced by Synopsys or TSMC. These files should be considered copyrighted but not otherwise proprietary; they are available for copying, study, or modification by the purchaser of this book, for individual learning purposes.

Verilog netlists produced by the synthesizer are not proprietary, although the *Liberty* models used by the synthesizer are proprietary and owned by Synopsys. Specific details of synthesized netlist quality may be considered proprietary and should not be published or distributed without special permission from Synopsys.

References

(*the date after a link is the latest date the link was used by the author*)

Accellera Organization. *System Verilog Language Reference Manual v. 3.1a*. Draft standard available free for download from Accellera web site at `http://www.accellera.org/home`.

Anonymous. "Design for Test (DFT)", Chapter 3 of *The NASA ASIC Guide: Assuring ASICs for Space*. `http://klabs.org/DEI/References/design_guidelines/content/guides/nasa_asic_guide/Sect.3.3.html` (updated 2003-12-30).

Anonymous. *SerDes Transceivers*. Freescale Semiconductor, Inc. http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01HGpJ2350NbkQ (2004-11-16).

Barrett, C. (Ed.) *Fractional/Integer-N PLL Basics*. Texas Instruments Technical Brief SWRA029, August, 1999. http://focus.ti.com/lit/an/swra029/swra029.pdf (2004-12-09).

Bertrand, R. "The Basics of PLL Frequency Synthesis", in the *Online Radio and Electronics Course*. http://www.radioelectronicschool.com/reading/pll.pdf (2004-12-09).

Bhasker, J. *A Verilog HDL Primer* (3rd ed.). Allentown, Pennsylvania: Star Galaxy Publishing, 2005.

Cipra, B. A. "The Ubiquitous Reed_Solomon Codes". *SIAM News*, **26**(1), 1993. http://www.eccpage.com/reed_solomon_codes.html (2007-09-18).

Cummings, C. E. "Simulation and Synthesis Techniques for Asynchronous FIFO Design" (rev. 1.1). Originally presented at San Jose, California: The *Synopsys Users Group Conference*, 2002. http://www.sunburst-design.com/papers/CummingsSNUG2002SJ_FIFO1_rev1_1.pdf (2004-11-22).

Cummings, C. E. and Alfke, P. "Simulation and Synthesis Techniques for Asynchronous FIFO Design with Asynchronous Pointer Comparisons" (rev. 1.1). Originally presented at San Jose, CA: The *Synopsys Users Group Conference*, 2002. http://www.sunburst-design.com/papers/CummingsSNUG2002SJ _FIFO2_rev1_1.pdf (2004-11-22).

IEEE Std 1364-2005. *Verilog Hardware Description Language*. Piscataway, New Jersey: The IEEE Computer Society, 2005. Revised in 2001 and reaffirmed, with some System Verilog added compatibility, in 2005. If you plan to do any serious work in verilog, you should have a copy of the standard. It is not only normative, but it includes numerous examples and explanatory notes concerning every detail of the language. In this text, we refer to "verilog 2001" syntax where it is the same as in the 2005 standard.

Keating, M., et al. *Low Power Methodology Manual for System-on-Chip Design*. Springer Science and Business Solutions, 2007. Available from Synopsys as a free PDF for personal use only: http://www.synopsys.com/lpmm (2007-11-06).

Knowlton, S. "Understanding the Fundamentals of PCI Express". Synopsys White Paper, 2007. Available at the Technical Bulletin, Technical Papers page at http://www.synopsys.com/products/designware/dwtb/dwtb.php. Free registration and login (2007-10-17).

Koeter, J. *What's an LFSR?*, at http://focus.ti.com/lit/an/scta036a/scta036a.pdf (2007-01-29).

Mead, C. and Conway, L. *Introduction to VLSI Systems*. Menlo Park, CA: Addison-Wesley, 1980. Excellent but old introduction to switch-level reality and digital transistor design and fabrication.

Palnitkar, S. *Verilog HDL* (2nd ed.). Palo Alto, CA: Sun Microsystems Press, 2003. A good basic textbook useful for supplementary perspective. Also includes a demo version of the *Silos* simulator on CD-ROM. Our daily *Additional Study* recommendations include many optional readings and exercises from this book.

Seat, S. "Gearing Up Serdes for High-Speed Operation", posted at http://www.commsdesign.com/design_corner/showArticle.jhtml?articleID=16504769 (2004-11-16).

Suckow, E. H. "Basics of High-Performance SerDes Design: Part I" at http://www.analogzone.com/iot_0414.pdf; and, Part II at http://www.analogzone.com/iot_0428.pdf (2004-11-16).

Sutherland, S. "The IEEE Verilog 1364-2001 Standard: What's New, and Why You Need it". Based on an *HDLCon 2000* presentation. http://www.sutherland-hdl.com/papers/2000-HDLCon-paper_Verilog-2000.pdf (2005-02-03).

Thomas, D. E. and Moorby, P. R. *The Verilog Hardware Description Language* (5th ed.). New York: Springer, 2002. A very good textbook which was used in the past as the textbook

for this course. Includes a demo version of the *Silos* simulator on CD-ROM. Our *Additional Study* recommendations include many optional readings and exercises from this book

Wallace, H. "Error Detection and Correction Using the BCH Code" (2001). http://www.aqdi.com/bch.pdf (2007-10-04).

Wang, D. T. "Error Correcting Memory - Part I". http://www.realworldtech.com/page.cfm?ArticleID=RWT121603153445&p=1 (2004-12-15: there doesn't seem to be any Part II).

Weste, N. and Eshraghian, K. *Principles of CMOS VLSI Design: A Systems Perspective*. Menlo Park, CA: Addison-Wesley, 1985. Old, but overlaps and picks up where Mead and Conway leave off, especially on the CMOS technology *per se*.

Zarrineh, K., Upadhyaya, S. J., and Chickermane, V. "System-on-Chip Testability Using LSSD Scan Structures", *IEEE Design & Test of Computers*, May–June 2001 issue, pp. 83–97.

Ziegler, J. F. and Puchner, H. *SER – History, Trends, and Challenges*. San Jose, Cypress Semiconductor Corporation, 2004. (stock number 1-0704SERMAN). Contact: serquestions@cypress.com.

Zimmer, P. "Working with PLLs in PrimeTime – avoiding the 'phase locked *oops*"'. Drafted for a Synopsys User Group presentation in 2005. Downloadable at http://www.zimmerdesignservices.com (2007-04-12).

Recommended Interactive Verilog Reference Guide

Evita Verilog Tutorial. Available in MS Windows environment. This is a free download for individual use. Sponsored by *Aldec* at http://www.aldec.com/Downloads. The language level is verilog-1995 only.

This interactive tutorial includes animations, true-false quizzes, language reference, search utility, and a few minor bugs and content errors. It is very easy to run and may be useful especially for understanding the content of the first few Chapters of the present book from a different perspective.

Chapter 1
Week 1 Class 1

1.1 Introductory Lab 1

Lab Procedure

This is a stereotyped, automated lab. Just do everything by rote; the lab will be explained in a postmortem lecture which will follow it immediately. Verilog source code files have a `.v` extension. The files in the design have been listed in the text file, `Intro_Top.vcs`, for your convenience in using the VCS simulator.

An include file, `Extras.inc`, provided in the CD-ROM `misc` directory, will have to be referenced properly in the `TestBench.v` file or copied to a location usable by the simulator of your choice. To do this, you may will have to edit `TestBench.v`, changing the path in `` `include "../../VCS/Extras.inc" ``. Notice that there is no semicolon at the end of a `` `include ``. If you comment out the `` `include ``, the simulation will work, but no VCD file will be created during simulation.

Step 1. In your `Lab01` directory, use a text editor to look at the top level of the design in `Intro_Top.v`. The top level structure may be represented schematically as in Fig. 1.1.

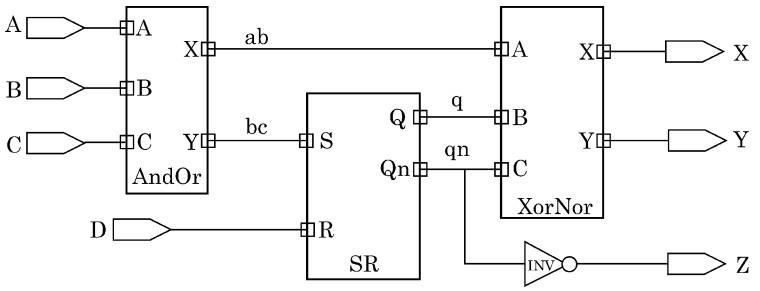


Fig. 1.1 Schematic representation of `Intro_Top.v`. The blocks are labelled with their module names

In the verilog file `Intro_Top.v`, documentation and code comments begin with "`//`". A copy of the contents is reproduced below for your convenience.

Notice that almost everything is contained in a "`module`". The top module in `Intro_Top.v` includes port declarations (I/O's), wire declarations, one assignment statement (a *continuous assignment* wire-connection statement), and three component instances. This module, combined with the component instances, represents the structure of the design.

```
// ============================================================
// Intro_Top: Top level of a simple design using verilog
// continuous assignment statements.
//
// This module contains the top structure of the design, which
// is made up of three lower-level modules and one inverter gate.
// The structure is represented by module instances.
//
// All ports are wire types, because this is the default; also,
// there is no storage of state in combinational statements.
//
// ANSI module header.
// ------------------------------------------------------------
// 2004-11-25 jmw: v. 1.0 implemented.
// ============================================================
module Intro_Top (output X, Y, Z, input A, B, C, D);
 //
 wire ab, bc, q, qn; // Wires for internal connectivity.
 //
 // Implied wires may be assumed in this combinational
 // design, when connecting declared ports to instance ports.
 // The #1 is a delay time, in `timescale units:
 //

assign #1 Z =~qn; // Inverter by continuous assignment statement.

 //
 AndOr InputCombo01   (.X(ab),  .Y(bc),   .A(A),   .B(B),   .C(C));
    SR SRLatch01      (.Q(q),   .Qn(qn),  .S(bc), R(D));
 XorNor OutputCombo01 (.X(X), .Y(Y), .A(ab), .B(q), .C(qn));
 //
endmodule // Intro_Top.
```

Step 2. Look at the verilog which will be used to run logic simulation on this design; it is in `TestBench.v`. The contents of this file are reproduced below. An equivalent schematic is given in Fig. 1.2.

The verilog simulator will apply stimuli to the inputs on the left, and it will read results from the outputs on the right. The testbench module (in `TestBench.v`) will be omitted when synthesizing the design, because it is not part of the design.

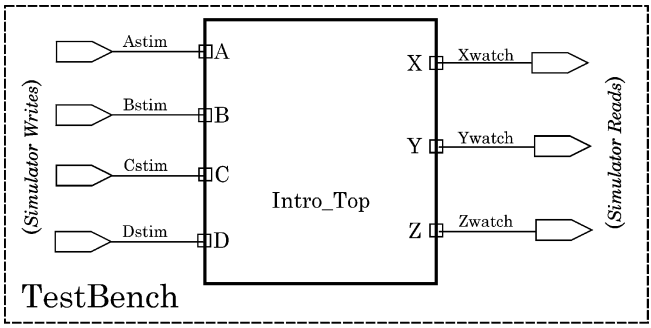


Fig. 1.2 Schematic view of the Lab01 `TestBench` module

```
// ============================================================
// TestBench: Simulation driver module (stimulus block)
// for the top-level block instance of Intro_Top.
//
// This module includes an initial block which assigns various
// values to top-level inputs for simulation. initial blocks
// are ignored in logic synthesis.
//
// No module port declaration.
// ------------------------------------------------------------
// 2004-11-25 jmw: v. 1.0 implemented.
// ============================================================
//
`timescale 1 ns/100ps     // No semicolon after `anything.
//
module TestBench;         // Stimulus blocks have no port.
//
wire Xwatch, Ywatch, Zwatch;    // To connect to design instance.
reg Astim, Bstim, Cstim, Dstim; // To accept initialization.

//
initial
  begin
  //
  // Each '#' precedes a delay time increment, here in 1 ns units:
  //
  #1 Astim = 1'b0;     // For Astim, 1 bit, representing a binary 0.
  #1 Bstim = 1'b0;
  #1 Cstim = 1'b0;
  (other stimuli omitted here)
  #50 Dstim = 1'b1;
  #50 Astim = 1'b0;
  #50 Cstim = 1'b0;
  #50 Dstim = 1'b0;
  #50 $finish;        // Terminates simulation 50 ns after the last stimulus.
  end // No semicolon after end.
//
// The instance of the design is named Topper01, and its
// ports are associated by name with stimulus input and simulation
// output wires:
//
Intro_Top Topper01 ( .X(Xwatch), .Y(Ywatch), .Z(Zwatch)
                   , .A(Astim), .B(Bstim), .C(Cstim), .D(Dstim)
                   );
//
endmodule // TestBench.
```

Notice the use of two different keyboard characters which may appear very similar:

- In the *timescale specifier*, **`timescale**, the '`' is a backquote. This character also is used in macroes and compiler directives: `define, `include, `ifdef, `else, `endif, *etc*.
- In the *width specifier* for literal constants, **1'b1** etc., the '**'** is a single-quote.

Step 3. Look briefly at the verilog for the three other modules in the design; this is in files `AndOr.v`, `SR.v`, and `XorNor.v`:

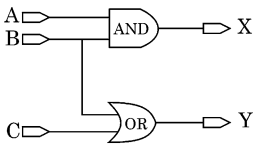


Fig. 1.3 Schematic for `AndOr.v`

```
// ============================================================
// AndOr: Combinational logic using & and |.
// This module represents simple combinational logic including
// an AND and an OR expression.
//
// ANSI module header.
// ------------------------------------------------------------
// 2004-11-25 jmw: v. 1.0 implemented.
// ============================================================
module AndOr (output X, Y, input A, B, C);
  //
  assign #10 X = A & B;
  assign #10 Y = B | C;
  //
endmodule // AndOr.
```

The only assignment statements are *continuous assignments*, recognizable by the **`assign`** verilog keyword. These are wire connection statements, mapping directly to Fig. 1.3. The '=' sign merely connects the left side to the right side. This connection is permanent and can not be modified during simulation.

assign means: "connect to this wire"

The "`#10`" literals are simulator programmed delays.

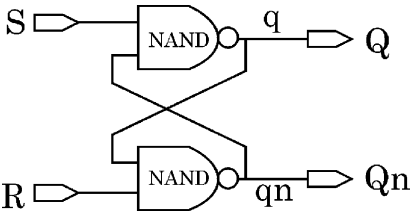


Fig. 1.4 Schematic for `SR.v`

```
// ============================================================
// SR: An S-R Latch using ~ and &.
// This module represents the functionality of a simple latch,
// which is a sequential logic device, using combinational
// ~AND expressions connected to feed back on each other.
//
// ANSI module header.
// ------------------------------------------------------------
// 2005-04-09 jmw: v. 1.1 modified comment on wire declarations.
// 2004-11-25 jmw: v. 1.0 implemented.
// ============================================================
module SR (output Q, Qn, input S, R);
  wire q, qn; // For internal wiring.
  //
  assign #1 Q = q;
  assign #1 Qn = qn;
  //
  assign #10 q = ~(S & qn);
  assign #10 qn = ~(R & q );
  //
endmodule // SR.
```

Four more continuous assignments in the `SR.v` verilog code again map directly to the schematic wiring in Fig. 1.4.

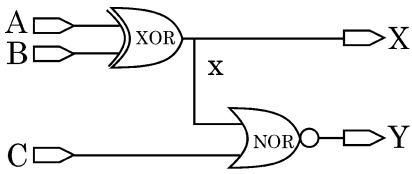


Fig. 1.5 Schematic for XorNor.v

```
// ============================================================
// XorNor: Combinational logic using ^ and ~|.
// This module represents simple combinational logic including
// an XOR and a NOR expression.
//
// ANSI module header.
// ------------------------------------------------------------
// 2005-04-09 jmw: v. 1.1 modified comment on wire declarations.
// 2004-11-25 jmw: v. 1.0 implemented.
// ============================================================
module XorNor (output X, Y, input A, B, C);
  wire x; // To illustrate use of internal wiring.
  //
  assign #1 X = x; // Verilog is case-sensitive; 'X' and 'x' are different.
  //
  assign #10 x = A ^ B;
  assign #10 Y = ~(x | C);
  //
endmodule // XorNor.
```

Three more continuous assignments map the Fig. 1.5 wiring to the verilog.

Step 4. Load `TestBench.v` into the simulator and simulate it, if necessary using the handout sheet (***VCS Simulator Summary*** or ***QuestaSim Simulator Summary***) provided in PDF format on the CD-ROM.

Don't ponder the result; just be sure that the simulator runs without issuing an error message, and that you can see the resulting waveforms displayed, as in Fig. 1.6 through 1.8, depending on the simulator of your choice.

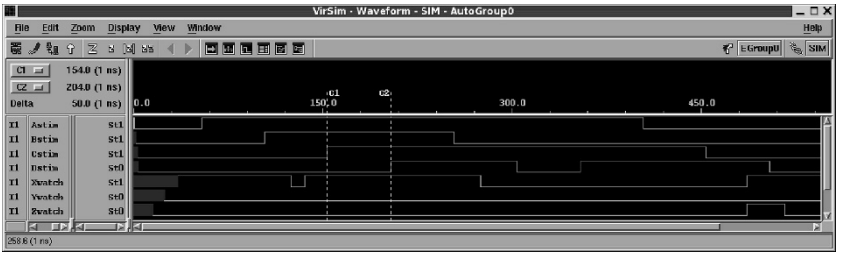


Fig. 1.6 `Intro_Top` simulation waveforms in the Synopsys *VCS* simulator

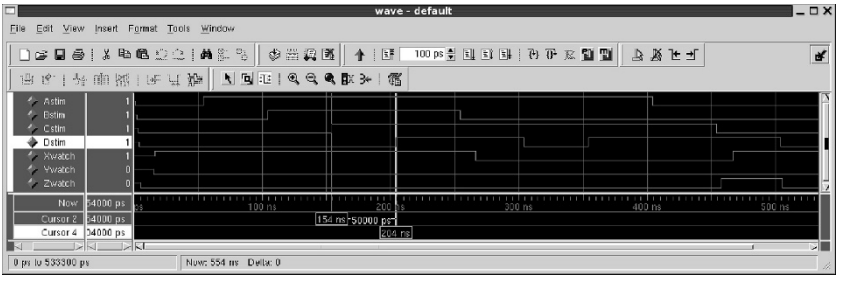


Fig. 1.7 `Intro_Top` simulation waveforms in the Mentor *QuestSim* simulator

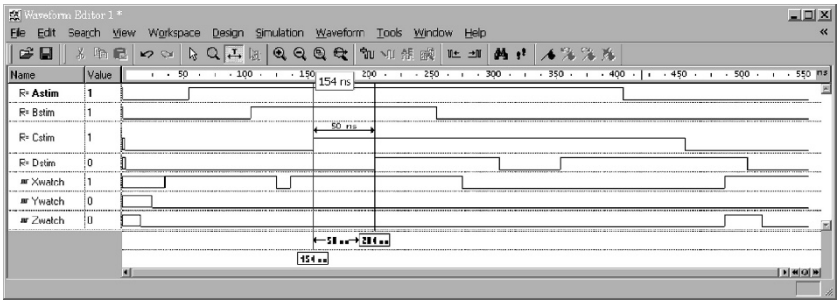


Fig. 1.8 `Intro_Top` simulation waveforms in the Aldec *Active-HDL* simulator

Step 5. The Design Compiler (DC) text interface, `dc_shell-t`, is the standard interface used by designers to synthesize, and especially optimize, digital logic. Typically, various `dc_shell` options, and directives embedded in the code, are tweaked repeatedly before the resulting netlist is satisfactory, meeting all timing and area requirements. These are text-editing exercises. There is a great advantage of text scripting in these activities. Graphical manipulations of synthesizer commands can be inefficient, undocumented, irreproducible, and error-prone when tweaking a synthesis netlist.

Our graphical interface to DC is `design_vision`, which we shall use in this course very occasionally, just to display schematics. We use the *TcL* (*T*ool *C*ontrol *L*anguage) scripting interface, indicated by "`-t`", for `dc_shell-t`; this also is the default scripting interface for `design_vision-t`.

There isn't much to synthesize in `Intro_Top`, it's already described almost on a gate level. But, continuous assignment statements aren't gates; so, their expressions will be replaced visibly by gates during synthesis.

Load the design into the logic synthesizer and synthesize it to a gate-level netlist, using the instructions below.

The gates we shall be using for synthesis in this course are provided in a 90-nm library from the TSMC fab house.

Lab 1 Synthesis Instructions

A. Invoke the synthesizer this way:

```
dc_shell-t -f Intro_Top.sct
```

There will be some text messages; then, you should see the `dc_shell-t` prompt. A `.scr` (Synopsys *scr*ipt) file holds a list of commands to be run before `dc_shell` presents its interactive prompt. Instead of .scr, we use **`.sct`** in this course to indicate *T*cl syntax.

We'll have plenty of time later to dwell on these messages. For now, you should scroll back and find where you invoked `dc_shell-t`, and then just quickly read forward, making sure there was no error message. The warnings are OK.

What the commands in `Intro_Top.sct` did, was this:

- They specified the technology and graphical symbol libraries for the synthesizer to use;
- they associated the design logical library (in memory) with a directory;
- they then analyzed and elaborated the design as disc files into that library; they also established operating conditions (`NCCOM` = *n*ominal-*c*ase, *com*mercial temperature range) and generic delay parameters (the wire-load model).

After this setup, the commands then set several specific constraints (max allowed area on chip, max allowed delays, estimated loads and drives) and then came to a stop in interactive mode. This design is named `Intro_Top`.

B. To the `dc_shell` prompt, enter these commands, waiting for each to finish:

```
dc_shell-xg-t> compile
dc_shell-xg-t> report_area
dc_shell-xg-t> report_timing
```

C. Briefly look over the results which were printed to the terminal:

The negative slack message means that the timing requirements were not met; we won't worry about this for now. Areas are approximate transistor counts, for this library.

D. Now save the result to disc:

```
dc_shell-xg-t> write -hierarchy -format ddc
```

The `write` command saved the results to a set of Synopsys `.ddc` files. These binary files are extremely efficient when saving a large design. But, we want verilog, so, write out a verilog netlist file and exit:

```
dc_shell-xg-t> write -hierarchy -format verilog -output Intro_Netlist.v
dc_shell-xg-t> exit
```

E. Take a look at the synthesized netlist schematic.

The netlist is verilog, so it may be examined in a text editor. However, because it is small and simple, we can convert it to a schematic and view it using the synthesis GUI interface. At the shell prompt, enter

```
design_vision
```

When the GUI has appeared, use the `File/Read` command to read in `Intro_Netlist.v`, which you just wrote out from DC.

Then, view the schematic by picking the "*and* gate" icon near the middle of the upper menu bar.

Notice that there isn't much change from the original design schematic above, because by default DC preserves design structure and leaves hierarchy intact.

Double-click a block to see its substructure; pick the up-arrow to return.

After viewing the schematic, use `design_vision` **`File/Exit`**.

F. Next, repeat the run of DC shell to flatten the netlist hierarchy.

Flattening typically is done to improve the timing optimization and the area. Again,

```
dc_shell-t -f Intro_Top.sct
```

When the setup commands have run, again scroll up briefly to check the messages.

Assuming no error, next,

```
dc_shell-xg-t> ungroup -all -flatten
dc_shell-xg-t> compile -map_effort high
dc_shell-xg-t> report_timing
```

The timing has been improved, and it now fulfills all constraints in the `.sct` file. Save the synthesized and ungrouped netlist, which no longer has any hierarchy, but don't exit yet:

```
dc_shell-xg-t> write -hierarchy -format verilog -output Intro_TopFlat.v
```

We'll do one more thing, while we have the synthesized netlist in memory: Write out a Standard Delay Format (SDF) file which contains the netlist timing computed from the library used by the synthesizer. We'll look at this file later in the day.

Be sure to give the new file a name with an SDF extension:

```
dc_shell-xg-t> write_sdf Intro_TopFlat.sdf
dc_shell-xg-t> exit
```

Then, after exiting DC, view the flattened and optimized netlist schematic: Invoke the synthesis GUI again. Use the `File/Read` command in `design_vision` again to read in the new `Intro_TopFlat.v`.

Examine the resulting synthetic schematic display, which should appear about the same as in Fig. 1.9. Each gate in this netlist could be mapped to a mask pattern taken from a physical layout library, layed out, and fabricated in working silicon.

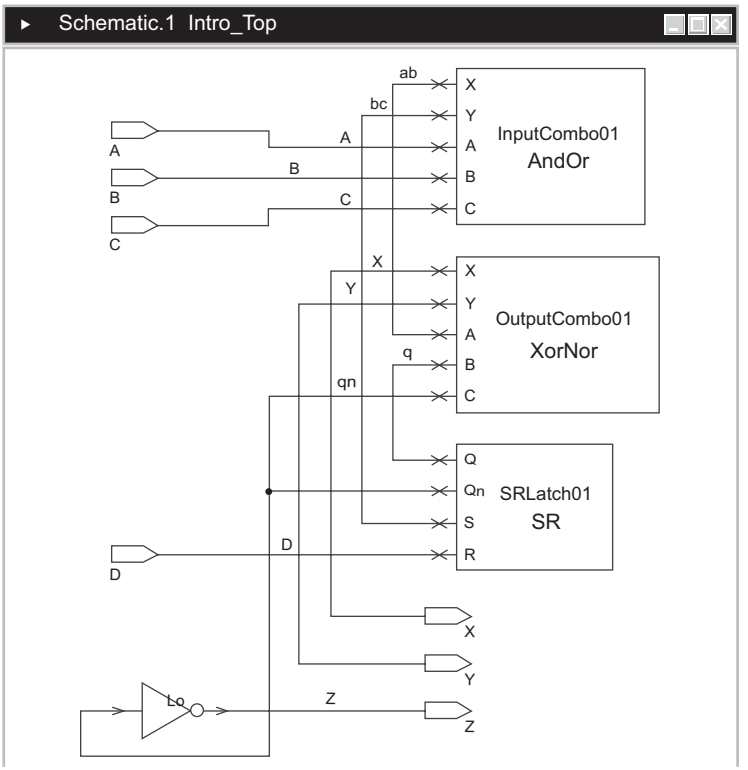


Fig. 1.9 The `design_vision` representation of the hierarchical top level of the synthesized `Intro_top` netlist

After viewing the schematic, pick `File/Exit` to finish the lab.

A note on "flatten" terminology: Although "flatten" often means to remove hierarchy, this word has a special meaning to Design Compiler. In DC, "flatten" refers to logical expressions, not to the design. To flatten the hierarchy in a design, the DC command is **`ungroup`**.

In DC, one may use a `set_flatten` command to flatten combinational layers of logic to a two-layer boolean sum-of-products; the `set_structure` command factorizes the logic and in a sense is the inverse of `set_flatten`.

1.1.1 Lab 1 Postmortem

Keywords. There were several verilog keywords used in the Lab 1 files: `module`, `endmodule`, `assign`, `wire`, `reg`. All were lower case.

Verilog keywords are lower case; but, *the language is case-sensitive*. Thus, one can avoid any possible keyword name conflict by declaring ones own names always with at least one upper-case character. So declaring, `module Module ... endmodule`, is perfectly legal, although a bit vague in practice.

Comments. In verilog, the comment tokens are the same as in *C++*: "`//`" for a line comment; "`/*`" and "`*/`" for a block comment. Block comments are allowed to include multiple lines or to be located in the middle of a line.

Modules. The only user-declared object type in verilog is the ***module***; everything else is predeclared and predetermined. Designers just decide how to name and use the other data types available. This makes verilog a simple language, and it permits a verilog compiler to be implemented bug-free.

A module is delimited by the keywords `module` and `endmodule`. Everything between these keywords defines the functionality of the module. A module is the smallest verilog object which can be compiled or simulated. It corresponds to a design block on a schematic. In the synthesizer, a verilog module is referred to as a "design".

Initial Blocks. The `TestBench` module contained an ***initial*** block. An `initial` block includes one or more procedural statements which are read and executed by the simulator, beginning just before simulation time `0`. Statements in an initial block are delayed by various amounts and are used to define the sequence of test-vector application in a testbench. Our Lab 1 testbench `initial` block included many statements; blocks of statements in verilog are grouped by the keywords, `begin` and `end`. Only statements end with a semicolon; blocks do not require a semicolon.

To terminate the simulation, add a `$finish` command at the end of the testbench `initial` block. Otherwise, the simulation may hang (depending on simulator configuration) and may never stop.

Module Header. A module begins with a module header. The header starts with the declared module name and ends with the first semicolon in the module. Except testbenches, which usually have no I/O, module headers are used to declare the I/O ports of the module. There are two different formats used for module headers in verilog: The older, traditional *verilog-1995* format, and the newer, ANSI-C, *verilog-2001* format. We shall use only the newer format, because it is less repetitive than the older one and is less prone to user error.

In the module header, ANSI port declarations each begin with a direction keyword, `output`, `input`, or `inout`; the keyword may be followed with bus indices when the port is more than 1 bit wide. Ports may be declared in any order, but usually it is best to declare `output` ports first (we'll see why later). Each direction keyword declaration is followed by one or more names of the I/O ports (wires). To declare additional port names, one uses a comma ('`,`'), another direction keyword, and another list of one or more names.

For example, here is a header with one 32-bit output, two 16-bit inputs, and two 1-bit inputs:

```
module ALU (output[31:0] Z, input[15:0] A, B, input Clock, Ena);
```

Assignment Statements. Our Lab 1 exercise contained two different kinds of verilog statement, ***continuous assignment*** statements to `wires`, and ***blocking assignment*** statements to `regs` (in the `TestBench` initial block). In verilog, **`wire`** types only can be connected and can not be assigned procedurally. One declares a different kind of data object, a **`reg`**, to assign procedurally. While procedural statements are being executed, `regs` can be changed multiple times, in different ways. To get a procedural result into the design, one uses continuous assignment to assign the `reg` to a `wire` or port.

For example,

```
module And2 (output Z, input A, B);
reg Zreg;
// A connection statement:
assign #1 Z = Zreg; // Put the value of Zreg on the output port.
// Procedural statements:
initial
  begin
     Zreg = 1'b0;
  #2 Zreg = A && B;
  #5 $finish;
  end
endmodule
```

This *and* gate model is rather useless, (the `initial` block terminates the simulation at time $2+5=7$); but, it shows how a `reg` and a `wire` are used together. `Zreg` is set to `0` before the simulation starts; the delay on the assignment to `Z` causes `Z` to go from unknown to `0` at time 1. At time 2, `Zreg` is updated from the module inputs, and `Z` may change then, for the last time, at time 3.

The blocking assignments in the `TestBench` and in the model above are indicated by '='. There is another kind of procedural assignment, the nonblocking assignment, which is indicated by "<=". We shall study nonblocking assignments later.

Simple Boolean operators. Verilog has about the same operator set as *C* or *C++*. In particular, the *logical* operators, '`!`', "`&&`" and "||", return a '`1`' or a '`0`', representing *true* or *false*, respectively. There also are *bitwise* operators, '&', '|', '^' (exclusive-*or*), and '$\sim$' which are the same as logical operators for 1-bit operands.

Four Logic Levels. In addition to the logically defined levels '`1`' and '`0`', verilog has two special levels to accommodate simulation states: '**`x`**' means either '`1`' or

‘0’, but that the simulator can not determine which; ‘**z**’ means “turned off” (three-state off) and generally means the same as ‘x’ except when multiple drivers on a single wire are in contention.

It usually is good practice to specify the width of every verilog literal expression, even when the width is just 1. So, we usually try to write “1‘b0” instead of just ‘0’. The width *in bits* precedes the quote (’), the number base (‘b’, ‘h’, or ‘d’) follows the quote, and the literal value is last.

Thus, 4’b000z means a 4-bit value with least-significant bit turned off. 12’b101 and 12’h5 have the same value, 5, and both are 12 bits wide.

Delay. A delay value is preceded by ‘#’ and always is in decimal base. The meaning of a delay value is determined by a ***timescale*** macro, usually at the beginning of the first file compiled for simulation. ‘timescale 1ns/100ps means that #1 equals a 1 ns delay, and that the simulator should calculate all delays to a maximum precision of 100 ps.

Component Instantiation. Module instances are the structural basis of all big designs. A module must be declared before it can be instantiated. A module instance also may be referred to as a component instance, especially when the instance is from a library of relatively small gates. Instantiations are statements and, like other statements in verilog, end with a semicolon.

The basic syntax of instantiation is: *module_name instance_name port_map*; for example, from Lab 1, the device under test was instantiated in TestBench this way:

```
Intro_Top Topper01 ( .X(Xwatch), .Y(Ywatch), .Z(Zwatch)
                   , .A(Astim), .B(Bstim), .C(Cstim), .D(Dstim)
                   );
```

The declared name of the instantiated module was Intro_Top; this instance was named Topper01, and the port map followed in the outer parentheses.

Port Map. The preceding instantiation shows how to map the port names of the module, Intro_Top, to the wires in TestBench: Each port name is preceded by a period (.), and the mapped wire, connected to that port, is immediately after it in parentheses.

1.2 Verilog Vectors

Verilog vector notation is used for ordered sets, or busses, of logical bits.

Verilog has similar *vector* and *array* constructs. We’ll look at arrays later.

A declared vector can be assigned to another one without explicit indexing; or, alternatively, a bit or part select may be made on either side of the assignment. A *select* means that part of the vector is being named explicitly. All these are legal:

```
reg[7:0] HighByte, LowByte1, LowByte2;
reg Bit;
...
HighByte = LowByte1;    // Assigns one vector to another.
...
Bit      = LowByte2[7]; // This is called a "bit select".
...
LowByte2[3:0] = LowByte2[7:4]; // Two examples of "part select".
...
```

All the assignments here are procedural, because a `reg` must be assigned procedurally. However, the module declaration and the blocks containing the procedural statements have been omitted for simplicity.

Notice the difference between verilog and *C* language declarations: The index range appears after the type name (`reg` or `wire`), not after the name being declared!

"Register" means "regular line-up". Registered data is the defining characteristic of Register-Transfer Logic (RTL), the level of abstraction most frequently used in digital simulation and synthesis.

Don't read verilog "`reg`" as *register*! Perhaps *transistor* would be better, but it still would be wrong. The "`reg`" declaration represents localization or storage of information, even if of just one bit. In verilog, a `reg` localizes a logic state; `wires` never are considered `regs`, even when regularly lined up in busses. Verilog `wires` just connect logic from one place to another; with multiple endpoints, they localize nothing. Both nets (`wires`) and `reg`'s are called *variables* in verilog, because their values can be changed by events during simulation.

The value of a `wire` can be changed only if the logic driving it changes value. The value of a `reg` can be changed by assigning it a different value in an `initial` or `always` block. An assignment to a `reg` is called a *procedural* assignment, because it can be done by a procedure of individual statements inside an `initial` or an `always` block.

Continuous assignments are done by continuous assignment statements ("`assign`" keyword) and represent permanent connections of something to a net. For example,

```
module ModuleName(...);
wire x;
...
assign x = a & b | c; // LHS must be wire; RHS wire or reg.
endmodule
```

All the assignments in the first lab's `Intro_Top` design were continuous assignments to `wires`; there were no `reg`'s in that design (excluding the testbench); therefore, there were no procedural assignments.

Procedural assignments are done in procedural blocks ("always" or "initial" keyword) and represent logical changes in a certain order. For example,

```
module ModuleName(..., input a, b, c);
reg x;
wire y;
//
assign y = a & b;
assign y = y | c; // y wired to two drivers probably is an error!
...
always @(a,b,c) // LHS must be reg; RHS may be wire or reg.
  begin
  x = a & b;
  x = x | c;    // x gets c | (a & b). No problem.
  end
endmodule
```

In any kind of assignment statement, binary, octal, decimal, and hexadecimal expressions are allowed for constant values (*literals*):

`1'b1, 1'b0, 1'bx, 1'bz. 8'b00zz_xx11. 64'h33b4_1223_1112_af01.`

`8'b1010_0010` is the same as `8'ha2` or `8'hA2` or `8'o242` or `8'd162`.

During logic *simulation*, limitations on displayed expressions tend to fail safe, in the sense that ambiguity is propagated throughout the values represented by a hex, decimal, or octal numeral. If any bit in a hex digit is 'z' or 'x', all four bits are treated as 'z' or 'x', respectively, when hex notation is used to represent the value. The same holds for decimal or octal numeral representations. Thus, assigning a variable `8'b000z_xx11` means that it will be displayed in hex as `8'hzx`. The 'x' is stronger than the 'z', so `4'b0z0x` is displayed as `4'hx`.

Timing expressions such as `#10` are ignored by the logic *synthesizer*; unknowns are treated as known by the synthesizer, but in special ways. Neither 'x' nor 'z' is meaningful as an actual value for synthesis: 'x' means the simulator can't assign a '1' or '0'; 'z' refers to three-state gate functionality, not to a logic state. The synthesizer has to apply its own interpretation to assignments of 'x' or 'z', so as to make the synthesized gates simulate the same way (except timing) as the presynthesis design.

The significance of the bits in a vector is unalterable. If `8'ha2` is assigned to an 8-bit variable declared `reg[7:0]`, bit `[7]` is the MSB, which gets a '1'; if it is assigned to a `reg[0:7]`, bit `[0]` is the MSB, and it still gets a '1'. The MSB always is on the left of a verilog vector.

```
// Some example vector operations:
`timescale 1ns/100ps
module Vector;
  reg [0:15] MyBus; // A vector of 16 bits of storage.
                    // The verilog MSB always is on the left.
  //
  wire[15:0] mybus; // Another vector, a 16-bit bus.
  ...
  MyBus[0:2] = mybus[10:8]; // This is called a "part select".
  MyBus = mybus;
  MyBus[0:15] = mybus[15:0];
  ...
endmodule
```

`` `timescale `` detail: It allows use of fractional delay expressions such as `#0.1`. For example,

```
`timescale 1ns/10ps
...
#0.05 X = Y & Z; // The delay is 50 ps.
```

1.3 Operator Lab 2

Log in and change to the `Lab02` directory; do all your work here for this lab.

Lab Procedure

Step 1. Vector exercise:

A. Create a file named `Vector.v`, and copy the "Vector" module declaration into it from the preceding lecture example (above). Fill in the missing parts ("...") of this `module` any way you want: But, use continuous assignment to drive some value onto `mybus`, and use blocking assignments in an `initial` block to schedule the assignments shown in the lecture example. Supply your own time delays. Check your work by simulating in VCS.

B. Now try to reverse bus directions by adding `MyBus[0:2] = mybus[8:10]` or `MyBus[15:0] = mybus[15:0]`. Compile in the simulator. What happens?

C. If you tried `mybus = MyBus` in your `initial` block, what kind of problem might occur?

D. Declare a 48-bit `reg` named `My48bits` and assign it in an `initial` block as follows: `#5 My48bits = 'bz; #5 My48bits = 'bx; #5 My48bits = 'b0; #5 My48bits = 'b1;` Notice that no width specification is given in these literal values. Simulate after adding `#5 $finish`. What happens? Suppose you used `1'bz`, `1'bx`, *etc.*?

Step 2. Boolean operator exercise:

A. Declare a `reg[4:0]` named `X` and two others named `A` and `B`. Try simulating assigning `A` and `B` to `X` as follows: Initialize `A = 5'b01010; B = 5'b11100;` then, run these in order: `X = A & B; X = A | B; X = A ^ B; X = ~A |~B`.
B. After that, try `X[0] = &A; X[1] = |A; X[2] = &B; X[3] = ^A & ~^B; X[4] = A[4] ^ B[4]; X = ^X`.
C. Parentheses may be used for grouping: Try `X = ((~A &^B)|(A & B))^A;`

These operators also work with unequal-sized vectors; we'll look at this later in the course.

1.3.1 Lab Postmortem

Things to have noticed:

Vector extension for logic `0`, `x`, or `z`, but not `1`
Binary *vs*. reduction operators
What happens with `#0.1` and `` `timescale 1ns/1ns ``?

1.4 First-Day Wrapup

1.4.1 VCD File Dump

The logic simulator can create a VCD file. The VCD (*V*alue-*C*hange *D*ump) file format is specified in IEEE Std 1364, section 18. Briefly, this file provides an ASCII file record of a simulation in an extremely compact format. Because it is plain, ASCII text, such a file is accessible to homemade scripts or other utilities for various purposes, for example, estimation of test coverage or detection of timing violations of rules not in force when the simulation was run.

The VCD file consists of a header with information such as a design module name, and a table of abbreviations of variable names. Ports can be dumped optionally as though they were variables in the module. Variable names are represented by one-character ASCII symbols such as '`!`', '`#`', or '`(`'. The body of the file is a list of simulation times written as #*time* (absolute; not a delay), followed immediately

by a list of variable values which were new at that time. For this reason, simulator output files of this kind often are called “time-value” files. Example:

```
...
$scope module TestBench $end
...
$var reg 1 # Cstim $end
$var reg 1 $ Dstim $end
$var wire 1 % Xwatch $end
...
#30
0#
#40
0$
...
```

The values dumped can be selected as level-only (4-state) or level plus strength. Because the VCD file format is an IEEE standard, it is portable across all tools reading or writing this format. One important use for it in a design flow is as a sample of simulation activity for an estimate of design dynamic power dissipation; this is beyond the present course.

1.4.2 The Importance of Synthesis

Although simulation might be considered enough for a front-end designer, this is quite incorrect. True, the simulator may be used to validate functionally the original design (source verilog) as well as any synthesized netlist; but, the timing used by the simulator is entirely artificial – it is entered manually by the designer, based on estimates and expectations. Only after a netlist has been synthesized, can the propagation delays of the gates, the setup, hold, and other clocking constraints, and the (back-annotated) trace delays be estimated with any accuracy.

More importantly, a netlist can be fabricated; it is a product of value in the marketplace. Simulator waveforms can’t be marketted or sold, so they are strictly for the benefit of the designer. Thus, a good netlist is a designer’s contribution to the success of the company, and the logic synthesizer is the tool which almost always is the means of creating that netlist.

1.4.3 SDF File Dump

An SDF file can be used to back-annotate delays into a netlist. The logic synthesizer can create this kind of file, but usually it is most useful when created by a tool with

access to a chip floorplan or layout. The *S*tandard *D*elay *F*ile format resembles that of lisp or EDIF, with punctuation supplied almost solely by parentheses.

For example, the SDF delays for a noninverting buffer in a netlist might be:

```
...
(CELL
  (CELLTYPE "BUFFD0")
  (INSTANCE U3)
  (DELAY
    (ABSOLUTE
    (IOPATH I Z (0.064:0.064:0.064) (0.074:0.074:0.074))
    )
  )
)...
 (INTERCONNECT U3/Z U4/A (0.002:0.002:0.002) (0.003:0.003:0.003)
 ...
```

An SDF file contains a complete timing representation of a design netlist, with delay information derived from a library and a static timing analysis or other delay extraction process. We shall look at this file again toward the end of the course, when we discuss back-annotation of layout timing into a simulation or layout netlist.

1.4.4 Additional Study

Read Thomas and Moorby (2002) chapter 1 on verilog basics.

Read Thomas and Moorby (2002) chapter 3 on synthesizable verilog through section 3.4. Ignore 3.4.4 (`casez` and `casex`).

Do Thomas and Moorby (2002) 1.6 Ex. 1.2 (different adders).

Find the `.VCD` file in the `Lab01` directory and open it in a text editor. When you ran your `Lab01` simulation, this `.VCD` file was created because of a directive, `` `include ``../../VCS/Extras.inc'' ``, which was present in the `Lab01` testbench file (`TestBench.v`). Examine the VCD file, just to see what it looks like. We won't be making use of VCD in this course, but it is good to know about. Detailed discussion is in Bhasker (2005), section 10.10.

Optional Readings in Palnitkar (2003)

Read through the exercises in chapters 1–4, and try a few if you want.

Study the S-R latch model in section 4.1; there is a runnable example on the Palnitkar CD as `Chap04/Chap_EG/SR_Latch.v`. We shall do further work with the S-R latch in a later lab.

Study section 2.6, a ripple-carry counter example. The code is available on the Palnitkar CD as `Chap02/Chap_EG/ripple.v`. We shall study various counter designs later in the course.

Read section 9.5.6 for some general information on VCD files and section 10.4 on SDF files. We'll do a lab on SDF format in one of our last chapters.

Chapter 2
Week 1 Class 2

2.1 More Language Constructs

Today we'll present enough of the verilog language to be able to design something meaningful. Also, we'll be cementing our understanding of certain basic constructs which we shall use again and again in the rest of the course. After an initial lab on conversion basics, we'll look into the essential concept of a shift register and then design one in lab.

Traditional module header format. We never shall use the traditional format for our lab designs in this course, and it is deprecated for all new design entry; however, many tools (and older textbooks) still write headers in the older format, so the 1995 format should be understood.

The traditional format follows that of K&R *C*, whereas the newer format follows that of ANSI *C*. In both verilog formats, a port name automatically is associated with an implied net of the same name, making all ports `wire` types by default.

The main difference is that in the 2001 format, header declarations are entirely contained in the header, which occupies the first statement in the module. The first semicolon in a 2001 module always terminates the header.

In the 1995 format, only the *names* of the ports are in the first statement (first parentheses); declarations of width or directionality follow and may be located anywhere in the body of the module.

This implies that, in the traditional format, it is necessary to enter the name of every port at least twice: Once in the first parentheses and once in a subsequent directionality and width declaration. In a small module, this is only a small burden; however, in a module with dozens or hundreds of I/O's, it greatly increases the risk of error.

In the traditional format, because the name of every `output` port had to be typed twice anyway, it was allowed, and actually common practice, to type it yet a third time, declaring the port to be a `reg` type if the design intent was to assign it procedurally. This saved the (trivial) effort of declaring a separate internal `reg` and using a continuous assignment statement to wire the internal `reg` to the `output` port, as is almost always done in modern design.

J. Williams, *Digital VLSI Design with Verilog*,

However, this apparently saved effort caused three problems: First, we now had three declarations of the same name, a condition which was confusing to designers who wanted their declarations each to reserve a different name for every different object. Second, it increased the likelihood of a maintenance error: The three different naming statements always had to be coordinated whenever any I/O was changed in the design. Third, it required different `output` ports to be treated fundamentally differently – with no evident difference in the initial parentheses naming those ports: Ports later named a `reg` had to be assigned procedurally, only in `always` blocks, and ports not later named a `reg` had to be assigned only in wiring statements such as continuous assignments.

Finally, in the traditional format, the "header" never had a well-defined location in the module; in principle, the header went on and on interminably, until finally, somewhere inside the module, the last I/O named in the initial parentheses was given a width and direction.

All these problems were overcome in the 2001 format, in which all I/O declarations were confined only to the header, and only were allowed to occur once.

Two design management problems with the traditional format also arose:

First, because module contents could modify the header's meaning, especially the I/O widths, at the start of a project, 1995 headers could not be sketched usefully and distributed to the design team. In 2001 format, it is easy to distribute complete headers and require that no designer modify a header without management approval.

Second, a `parameter` (= a named configuration constant; see below) declared and defaulted after the header in the 1995 format was fundamentally ambiguous: It could not be made clear which `parameters` should be overridden in a module instantiation, and which were meant to be strictly internal to the module. In the 2001 format, although `parameters` not declared in the header still can be overridden in an instantiation, design-team procedures easily can be stated so that instance overrides are restricted to parameters in the header.

Here are two alternative module declarations illustrating the header differences. The module functionality has been omitted. In 1995 header format:

```
module MyCounter
    (CountOut, CountReady, StartCount, Load, Clock, Reset);
output[15:0] CountOut;
output       CountReady;
input[15:0]  StartCount;
input        Load, Clock, Reset;
reg[15:0]    CountOut;  // Could be anywhere in the module.
...
endmodule
```

The 1995-format header never really ends.

In 2001 header format:

```
module MyCounter
       (output[15:0] CountOut, output CountReady
       , input[15:0] StartCount, input Load, Clock, Reset);
// Header has ended; module itself now begins:
reg[15:0] CountOutR;
assign CountOut = CountOutR;
...
endmodule
```

One additional aspect of the difference is that the 2001 continuous assignment to the `CountOut` net allows for specification of different delay values for rise and fall. This is a minor advantage but a real one. Procedural delays can have only one delay value, implying that 1995-style direct declaration of `CountOut` as a `reg` would mean that it could be assigned only one value for rise and fall. The different ways of assigning verilog delays will be presented later.

In 2001 format, it still is possible to declare an `output` port to be a `reg`,

```
module My2001Module (output reg[15:0] CountOut, ...);
```

however, this should be avoided in a modern, manually-entered design. Although `reg` output port declarations in the 2001 format create only a minor inconvenience, we shall not allow them in our coursework.

Side note: An `input` or `inout` port never can be a `reg`, because a `reg` on an `input` only could be changed procedurally, and there are no header procedures. Therefore, a `reg` input always would be uninitialized and would contend with anything wired to that `input` when the module was instantiated, creating a perpetual (and illegal) 'x' on that `input`.

Verilog comments. There are two different formats, the same as for $C++$:

- Line comment: "/ /" Starts anywhere on a line.
- Block comment: "/ *" begins a comment which doesn't end until "* /".

Comment examples:

```
assign X = a && b; // Put the AND of a and b on X.
//
assign Y = a /*left operand*/ && b /*right operand*/;
//
assign Z = (a>5)? a&&b : a||b;
/* The preceding assignment puts the AND of a and b
   on Z if a is greater than 5; otherwise, it puts
   the OR of a and b on Z. */
```

Comments also may be effected by nonexistent-macro regions: Refer to an undefined macro variable name in "`ifdef *undefined_name*"; this will cause everything until "`endif" to be ignored by a verilog-conforming compiler (simulation or synthesis). Thus, an undefined macro name may be used effectively to commented out code. Later, the designer may `define the name and thereby uncomment the code and put it back into the verilog. This is discussed more fully below.

Comments can be used by tools to impose constraints not part of the verilog language: Synthesis directives or other attributes referring locally to verilog code constructs can be embedded in line comments if the comment containing them includes certain tool-specific keywords. A tool looking for keywords will read at least part of every verilog comment in the code. A typical such comment would be in the form of "//rtl_synthesis ...", with a command or constraint following the keyword "rtl_synthesis" on the same line.

Always blocks. A change of value in the sensitivity list (event control expression) of an always block during simulation can cause the statements in the block to be reread and reexecuted. A logic synthesizer usually will synthesize logic for every always block. Omission of a variable from the always block sensitivity list means that the state of the logic does not change when that variable does, possibly implying latched data of some sort. We shall discuss this idea in detail later in the course.

Thomas and Moorby (2002) includes examples of always blocks without sensitivity lists. There is nothing wrong with this usage; but, in this course, we avoid it for reasons of style. Omission of an event control (sensitivity list) is not common in design practice, because location of an event control at the top of a block makes the functionality of the block more easily understood. In this course, we recommend not to use "always" except when immediately followed by an event control ("always @(*variable_list*)").

```
...
always                          // Avoid this style.
  #10 ClockIn <= !ClockIn;
...
always@(ClockIn)                // Recommended style.
  #10 ClockIn <= !ClockIn;
```

Initial blocks. An initial block has no sensitivity list and is read only once, beginning before simulation time 0, when no event control can be expressed. All initial blocks are ignored by synthesis tools. Because initial blocks can not be activated and disabled concurrently but only can schedule events after various delays from time 0, it usually makes good sense only to use one initial block in a simulation; more than one would just make it hard to determine the order in time of the events to be scheduled.

Exceptions to limiting the design to one initial block may be useful when a second initial block is reserved to implement a simulation clock (using

`forever`) or is reserved for actions unrelated to design functionality, such as to define an SDF file to load, or to set up a `$monitor` task.

Continuous assignments. We should mention these again here, for completeness. The three most common concurrent blocks in verilog are `always` blocks, `initial` blocks, and continuous assignments – in addition to structural (hierarchical) design instances. These three blocks, and hierarchical instances, are the designer's main work when entering a design in verilog. A continuous assignment is sensitive to a change of anything on its right-hand side. A few examples follow.

A wire assignment (continuous assign) is sensitive to the expression on the RHS:

```
assign #2 X = A[0] && B[0] || !Clock;
```

A procedural assignment is sensitive to changes in its event control list:
`Xreg` is not updated on changes in `Clock`:

```
  always@ (A[0], B[0]) Xreg = A[0] && B[0] || !Clock;
```

`Yreg` is updated only when `Clock` goes to `1'b1`:

```
  always@ (posedge Clock) Yreg = a && b;
```

Vectors and vector values. All vectors are read as numerical values. Most verilog vector types (`reg` or net) are read as unsigned by default. Exceptions are `integers` and `reals`, which are read as signed. `reals` generally are not synthesizable.

Assuming no overflow, the bit-pattern representing a specific number is identical, whether the storage is signed or unsigned; the difference is when the value is being used in a comparison of some sort. When the value is used, the MSB of a signed number is used to determine whether the number is positive or negative ('1' means negative, as usual in 2's complement arithmetic). The MSB of an unsigned number simply is the most significant bit representing its value; an unsigned number can not be negative, no matter what its bit pattern.

Example of the meaning of the sign bit:

```
reg[31:0] A;
integer I;
...
A = -1;  // Default is to assume decimal integers.
I = -1;  // Now, both A and I hold 32'hffff_ffff.
//
if ( I > 32'h0 )
     $display("I is positive.");
else $display("I is not positive.");  //Prints "I is not positive."
if ( A > 32'h0 )
     $display("A is positive.");
else $display("A is not positive.");  //Prints "A is positive."
```

Verilog **logical operators** `!, &&,` and || treat operands as `true` when any bit in a vector operand is nonzero and `false` for all-zero, only.

Verilog binary **bitwise operators** `&`, |, and ^ operate bit-by-bit. The unary ~ operator inverts each bit in a vector. Every binary bitwise operator can be used as a **reduction operator**: `&A` expresses the *and* of all bits in vector `A`.

Unnamed **constant** values are called ***literals***. A literal expression consists of a width specification, a delimiting tick symbol (**'**), a numerical base specifier, and finally a number expressed in that base: *`width'base(value)`*. For example, `5'h1d` defines a 5-bit wide vector literal with hex value `1d` (`= 16*1 + 1*d = 16 + 13` = decimal `29`). The expression, `16'h1d`, also evaluates to decimal `29`, but it is much wider.

If width and base are omitted, a literal number is assumed to be in decimal format; if a whole number, it is assumed to be a 32-bit integer (signed).

Vector **type conversions** are performed consistently and simply, avoiding the need for explicit conversion operators. Verilog actually has no user-defined type, so the rules can be quite simple:

1. The significance of vector bits is predefined; the MSB is always on the left.
2. All values in an expression are aligned to the rightmost bit (LSB).
3. All expressions on the right-hand side of an assignment statement are adjusted to the width of the destination before operators are applied.
4. Width adjustment is by simple truncation of excess bits, or by filling with `0` bits. When an operand is signed, the widening fill is by the value of the original MSB (sign bit).

Examples of type conversions for vector operations:

```
reg X;
reg[3:0] A, B, Z;
...
X = A && B; // X gets 1'b0 only if either A or B is 4'b0.
Z = A && B; // Z gets 4'b0 only if either A or B is 4'b0.
            // Otherwise, Z gets 4'b0001
Z = A & B;  // Each bit of Z get the and of the
            // corresponding bits of A and B.
X = !Z;     // X gets 1'b1 only if all bits of Z are 0.
X = ~Z;     // Z narrows to its LSB and is inverted; X gets !Z[0].
```

Truncation and widening examples (unsigned only, for now). Notice how a variable can be initialized when it is declared; this is for simulation, only:

```
reg       A = 1'b1, B = 1'b0;
reg[3:0]  C = 4'b1001, D = 4'b1100;
reg[15:0] E = 16'hfffe;
...
C = A | B; // C gets A=4'b0001 | B=4'b0000 --> 4'b0001.
C = E & D; // C gets E=4'b1110 & D=4'b1100 --> 4'b1100.
E = A + C; // E gets A=16'h1 + C=16'h9 --> 16'ha.
```

Verilog includes **parameter declaration**s. Parameters are named constants which must be assigned a value when declared. If declared in a module header, the value assigned is a default; this value may be overridden when the module is instantiated.

Inside a module,

```
module ModuleName ( ...  I/O's ... ); // <-- semicolon ends header.
...
parameter Awid = 32, Bwid = 16;
...
reg[Awid-1:0] Abus, Zbus;
wire[Bwid-1:0] Bwire;
...
endmodule
```

In a module header,

```
module ModuleName
      #(parameter Awid = 32, Bwid = 16) // <-- no comma!
       ( ...  I/O's ... ); // <-- semicolon ends header.
...
reg[Awid-1:0] Abus, Zbus;
wire[Bwid-1:0] Bwire;
...
endmodule
```

Parameters are unsigned by default, and are as wide as required to hold the value assigned.

Conditional commenting with ***macro***es. To comment out a block of statements or other design data, one way is to use the verilog block comment tokens, /* and */.

This can be useful when unsynthesizable code would cause the synthesizer to issue an error. But, often, a better way is to know that when (our) synthesizer reads a verilog file, it defines a global macro DC before it starts. This macro is empty, but its name is always declared. This means that you can comment out code for the synthesizer but not for the simulator by surrounding the offending lines with `ifndef DC ("if DC not defined") and `endif.

For example, the synthesizer rejects verilog system tasks. So, use the DC macro to make verilog system-task assertions invisible to the synthesizer but visible to a simulation compiler (which actually can use them):

```
always@(posedge Clk)
  begin
  Xreg = NewValue;
  `ifndef DC
  $display("time=%05d: Xreg=[%h]", $time, Xreg);
  `endif
  Yreg = ...
  end
```

The Silos demo simulator included with the books named in the *References* does not accept `‘ifndef`, but it does accept `‘ifdef` and `‘else`; so, a more portable way of writing the example above would be as follows:

```
always@(posedge Clk)
  begin
  Xreg = NewValue;
  ‘ifdef DC
  ‘else
  $display("time=%05d: Xreg=[%h]", $time, Xreg);
  ‘endif
  ...
```

A good coding rule is to use `‘define` and other macro definitions sparingly: They are global, and more than one of the same name may exist inadvertently in a large design. Then, a new value during compilation may replace an older one, possibly causing design errors.

To avoid errors because of compilation order, a defined macro should be undefined as soon as it has served its purpose, preferably in the same file as the one in which it was defined. Exception: `‘timescale`.

Example:

```
‘define BLOCKING
‘define Awid 32
module ModuleName ( ... I/O’s ... );
  ...
  reg[‘Awid-1:0] Abus, Zbus; // The ‘ is required for value!
  ...
  always@( ... )
    begin
    ‘ifdef BLOCKING
    Qreg = Areg | Breg;
    ‘else
    Qreg <= Areg | Breg;
    ‘endif
    end
endmodule
‘undef BLOCKING // Use ‘undef unless you want these to
‘undef Awid     // be seen in subsequently compiled files.
```

2.2 Parameter and Conversion Lab 3

Work in the `Lab03` directory for these lab exercises.

Lab Procedure

Step 1. Synthesis of a design with parameterized widths. The design in the `Lab03` directory consists of a top-level verilog module in `ParamCounter Top.v` and two submodules in `Counter.v` and `Converter.v`. A schematic is given in Fig. 2.1, with only those pins shown which entail a name change in the port mapping. For example, the port `OutEnable` is connected to an `Enable` pin:

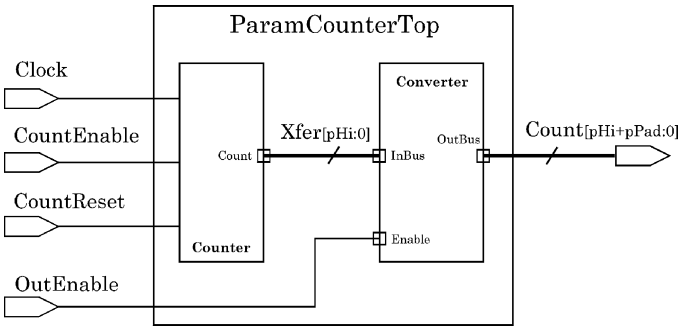


Fig. 2.1 Schematic for `Lab03`, Step 1

The design is a resettable up-counter which puts its result in the lower-order bits of an output bus of configurable width. The design is intended to show how parameters are used to configure a design. In this exercise, only bus width is parameterized.

Don't bother simulating, unless you want to do it after finishing this lab. Use the `DC` macro to comment out the entire `TestBench` module before synthesizing.

When you are ready, change `PADWIDTH` to 8, and synthesize the design; optimize it for area and then for speed. To synthesize for area, just comment out all speed-related constraints (but leave the design rules); to synthesize for speed, leave these constraints in, but keep the area goal at 0. Area and speed are somewhat correlated in practice.

After experimenting with `PADWIDTH` set to 8, increase the value of the `pPad` parameter and resynthesize to see what happens.

Step 2. Creation of a design to simulate arithmetic. Next, try some signed and unsigned arithmetic, using the simulator to view the results:

Create a verilog module in a new file `Integers.v` and check the results of the following (you may change the variable names if you wish to do them all in one `initial` block in one module, simultaneously):

A. `integer A = 16, B = -8, X; X = A + B; X = A - B; X = B - A;`
B. Same as above with all declared as `reg[31:0]` (use `32'd` to assign to this `reg`).
C. (optional) Same as above with the following successive changes: Just `X` declared as `integer`; just `A` declared `integer`, and just `B` declared `integer`.

Step 3. To see the effect of truncation and widening of various types, create a new verilog module in a file `Truncate.v` and try the following in the simulator: Declare data objects as follows: `integer Int`; `reg[7:0] Byte`; `reg[31:0] Word`; `reg[63:0] Long`; and `reg[127:0] Dlong`. Initialize each one with `'b0`. Then,

A. Assign `6'd1` and then later `-6'd1` to each of the above objects and notice what happens, both with hex and decimal radix display:
B. Assign `36'd1` and then later `-36'd1` to each as in A.
C. Assign `1` and then `-1` to `Int`, and use `Int`, not a literal, to assign each of the others as in A and B above.
D. Assign `32'h7eee_777f to Word` and then use `Word` to assign each of the others as above. This value has a `0` MSB and so should represent a positive number.
E. Finally, assign `32'hf777_eee7` to `Word` and assign to each of the others. The `1` MSB should represent a negative number.

2.2.1 Lab Postmortem

Side note: Vector negative indices are allowed, although not used in this course. The width always is given by the difference, plus 1.

For example, a declaration of,

```
reg[3:-7] CoeffA;
```

may be interpreted by a tool to represent an 11-bit decimal fraction with LSB equal to 2^{-7} = `1/2`^{7} = `1/128`. Used in DSP modelling.

2.3 Procedural Control

2.3.1 Procedural Control in Verilog

- We shall study three procedural control constructs at this point: `if`, `case`, and `for`. These only are allowed in a procedural block – `initial` or `always`.

Use **`if`** for explicit priority or for ranges of values. The `if` often is easier to use than the other control constructs when describing a short list of mutually exclusive events such as a clock and an asynchronous control.

```
if (expr) statement1;
else if (expr)  statement2;
else statement3;
```

Use **`case`** when alternatives are specific values, are numerous, or are conceptually unprioritized, for example to implement a table lookup or small memory addressing scheme. The verilog `case` breaks automatically and does not "fall through" on a match the way the *C* language `case` (in a *`switch`* statement) does. Good practice is *never* to omit the `default` of a `case` statement; we'll return to this issue, and the `case` statement, later in the course.

```
case (expr)
      alt1: statement1;
      alt2: statement2;
       ...    ...
default: default_stmt;
endcase
```

The `case` alternatives usually are constant expressions but may be variables; the `case` expression usually is a variable.

`for` is the preferred looping construct in verilog; it works about the same way as the *C* `for`. However, the *C* language unary increment and decrement expressions, `i++`, `++i`, `i--`, and `--i`, are not allowed. In verilog, one must use `i = i + 1` or `i = i - 1`.

`for` (*loop var init; loop reentry expression; loop var update) statement (*s);

2.3.2 Combinational and Sequential Logic

- The conditional expression operator, *control_expr* `?` *True_expr* `:` *False_expr*. This is used like a *C* function call returned value – in an expression. It is an expression, not a statement. The expression always is interpreted by the current synthesizer as combinational logic. If the control expression evaluates to *true* (non-`0`), the *True_expr* expression is its value; otherwise, it evaluates to the *False_expr*. Example:

```
wire[31:0] X;
integer A, B;
...
// Put the greater of A or B into X; A if they are equal:
assign #2 X = (A>=B)? A : B;
```

This is very useful for a mux connection to a `wire`, because `if` is not allowed in a continuous assignment (`if` may be used only in a procedural block).

- Simple `always@` block syntax: Use '`,`' in the event control (sensitivity list); the traditional but inconsistent "`or`" is deprecated. If any change causes the block to respond, the logic is combinational. When a `posedge` or `negedge` expression causes insensitivity to the opposite edge, the result is sequential logic:
  `always@(posedge Clk, posedge Reset)` means the same as the traditional verilog `always@(posedge Clk` *`or`* `posedge Reset)`.
  It also is possible to embed event controls inside an `always` block:

```
always@(posedge clk)
  begin
  xbus[1] <= 1'b1;
  @(posedge Enable) // Execution stops here until Enable goes to '1'.
     begin
     Dbus      <= 8'haa;
     xbus[7:4] <= 'b0;
     end
xbus[2] <= 1'b0;
...
end
```

- For complicated combinational constructs , consider using blocking assignments in an `always` block, with the result put on the right side of a continuous assignment statement. Example:

```
always@(Ain, Bin, Cin, Din, temp1, temp2)
  begin
  temp1 = Ain^Bin;
  temp2 = Cin^Din;
  ComboOut = (temp1 & temp2) | Ain^Din;
  end
assign #5 OutBit = ComboOut | OtherComboOut; // Collect the delays here.
```

- nonblocking assignments. These are statements within a procedural block. They differ from blocking assignments in two ways: (*a*) during simulation, they do not block reading of the next statement; and, (*b*) at any given simulation time, they are evaluated along with blocking statements, but they are executed only *after* all blocking assignments and net updates are complete.

- Use nonblocking assignments in `always@` block for clocked sequential logic. Nonblocking evaluation occurs when blocking evaluations do, but assignment is scheduled after all blocking assignments on a given clock event, thus ensuring that combinational input values will not have been altered by nonblocking updates before being clocked in.

Avoid: Blocking assignments for clocked sequential logic, nonblocking assignments for combinational logic.
Avoid: Mixed blocking and nonblocking assignments in one procedural block.
Avoid: Delay of `#0` to fix up race conditions caused by failure to avoid!

- Implementation of clocks and storage registers. Clocks imply sequential elements (storage), because the clocked values remain constant in value between edges. Let's ignore level-sensitive latches for now; if we do so, sequential elements are synthesized by use of the edge specifiers, `posedge` or `negedge`, in an event control. Sensitivity only to an edge implies storage on the opposite edge.

A clock generator usually appears only in a testbench and may be defined by means of a single, delayed nonblocking assignment in a level-sensitive `always` block:

```
reg Clock;
...
always@(Clock)
  #10 Clock <= !Clock; // ~Clock also OK.
```

This makes use of the concurrent `always` statement. The assignment must be nonblocking, so that the `always` block event control will be sensitive to the inversion. The clock `reg` must be initialized somewhere else.

Another way to define a clock is to use the procedural `forever` statement in an `always` or (more usually) `initial` block. Being procedural, a `forever` must be enclosed in a concurrent block in its module. For example,

```
reg Clock;
...
initial // Only for clock generation.
  begin
  Clock <= 1'b0;
  forever
    #10 Clock <= !Clock;
  end
```

This clock would work with blocking assignments. However, with blocking assignments, anything clocked by it would require special treatment to ensure proper data setup.

Be careful not to use an `initial` block to initialize things that should be synthesized or used to represent hardware initialization: This is a very important difference between coding of simulation software and coding of hardware.

2.3.3 Verilog Strings and Messages

- **Verilog string type.** This is for literal strings, only. We shall not use strings except in *system task* messages, only. String values may be stored in a `reg` vector,

or in a memory object (array of bytes); but, be careful, especially with unicode text or other nonASCII character encodings. Messaging system tasks all print to a simulator (console) text screen; they resemble *C* language *printf* functions. See Thomas and Moorby (2002) sections B.4 and F.1 for more information on messaging during simulation.

The three most useful messaging tasks :

`$display` (*format , args_for_display*) for a simulation info printout when it is encountered procedurally but *before* RHS evaluations (before nonblocking assignments) at that simulation time.
`$strobe` (*format , args_for_display*) for a simulation info printout when it is encountered procedurally but *after* RHS evaluations (after nonblocking assignments) at that simulation time.
`$monitor` (*format , args_for_display*) for print-on-change procedural simulation info *after* RHS evaluations and nonblocking assignments *whenever* one of its args changes. Usually invoked in an `initial` block, possibly under a condition or after a delay.

Assertions are routines embedded in the design, by the designer's foresight, to check that a condition holds and to announce a warning or error during simulation. Like simulation itself, they are a design activity as much as a verification activity.

Assertions *assert* that something should be true, remaining silent when it is; and, they report when it is false. Assertions bring attention to conditions which otherwise might be overlooked after a design change, or under unusual or complex simulation conditions.

Whereas the primary purpose of an assertion is to warn the designer when the asserted condition has failed, assertions also may be used to generate simulation errors and cause a running simulation to pause or terminate. We shall deal only with assertion messages for now.

A simple, homemade assertion statement using the **`$time`** system task:

```
reg X, Y;
...
if (X!=Y)
$strobe("\n***Time=%04d. X=%1b == Y=%1b failed.\n", $time, X, Y);
```

Messages not only contain text strings, but they almost always also should report values of design variables; so, they must provide readable and useful formats for those values.

The most useful format specifiers are these (* = most common):
* `%h` hex integer
* `%d` decimal integer
  `%o` octal integer
* `%b` binary integer

- `%v` strength level
- `%c` single ASCII character
- * `%s` character string
- * `%t` simulation time in timescale units
- `%u` 2-value data of `0`, `1`
- `%z` 4-value data of `0`, `1`, `x`, `z`
- * `%m` *m*odule instance name (see below)

Any of the integer formats except `%t` may be preceded by "*n*", in which *n* is an integer, to constrain the minimum number of unused leading places displayed. In addition, "`0`*n*" fills remaining leading displayed places with zeroes. For example,

```
integer x; ...; x = 5;
$display("%s = [%04b] = [%4b].", "The value", x, x);
```

prints to the simulator console, "`The value = [0101] = [  101].`". Otherwise, using "`%b`" with no *n*, the integer would be printed with a default of 29 leading blanks, because integers are 32 bits wide.

There is a special string replacement option, `%m`, which appears to be a format specifier but which actually does not format anything; instead, it is replaced in the output with the full hierarchical name of the instance in which the system task was executed. Hierarchical names are explained in detail in *Week 6 Class 2*.

2.3.4 Shift Registers

A shift register is a register of bits which shifts the bit-pattern up or down in the register. Often, "shift right" is used to describe shifting down, and "shift left" is used for shifting up. Up vs. down refers to the unsigned numerical value interpreted from the register contents. Counters count up or down by similar reasoning. The same terminology is used in software assembly-language programming, in which the contents of a register in the CPU or memory are shifted one way or the other.

Binary representation is most useful in understanding what happens during a shift. For example, suppose an 8-bit register with this content: `8'b0001_1001`. A shift up changes the contents to `8'b0011_0010`. The leftmost bit is lost, and a new '`0`' appears on the right (because a '`0`' would be shifted into the register by default). A shift down changes the original contents to, `8'b0000_1100`. If this register, in hardware, was hooked up so its MSB input connected to its LSB output during a shift down, then the original contents, shifted down, would be, `8'b1000_1100`: The LSB '`1`' would be shifted into the MSB of the register.

When a shift register is connected to other design elements, the new bit appearing on one end or the other depends on however the register has been connected. What happens to the bit shifted out also depends on the design.

2.3.5 Reconvergence Design Note

A shift register with programmable storage is a good way to introduce the problem of reconvergent fanout – in this case, of a clock.

To use a shift register as something other than a 1-bit FIFO or a latency control, the shifting should be capable of being disabled; this allows the shift register to store its current value for one or more clocks. A simple way to achieve this is to gate the shift clock, as shown in Fig. 2.2.

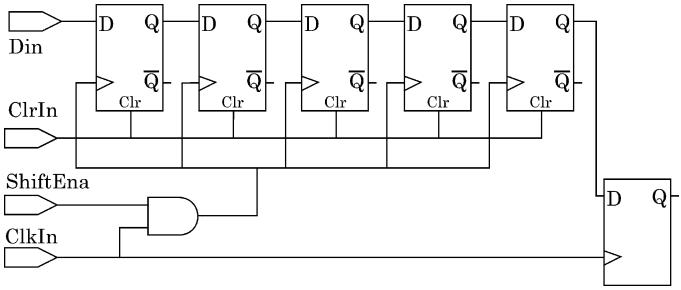


Fig. 2.2 Shift register which stores data when clock is gated off. The clock gate may cause a reconvergence error

However, in the application shown, the clock which shifts the data also clocks an independent flip-flop as shown on the far right. This clock has reconvergent fanout, because its logical effect, after traversing the shift register, reconverges on the external flip-flop. From the schematic alone, the problem in this case is likely to be setup: The *and* gate on the shift clock causes it to arrive later than the clock on the external flip-flop. Thus, even with a design allowing for setup on the individual components, the external flip-flop may be clocked too soon to capture the current value on the last shift flop.

This is a general problem in any design with a gated clock. It is overcome by using specially designed shift flops with an input-enable control pin, or by adding delay to the clock on all components which are independent of the shift register and which receive data from it.

Possible reconvergence is something always to keep in mind; however, the synthesizer is designed to accommodate reconvergence and can be expected to create a netlist from the (gated-clock) design in Fig. 2.2 without the setup error described. Of course, if the designer *wants* the external logic to receive late data, then the synthesizer must be constrained to create it that way. We shall not discuss this problem further at this point in the course. In the next lab, we avoid reconvergence by adding shift-enable logic manually to each of our shift-register flip-flops.

2.4 Nonblocking Control Lab 4

Work in the `Lab04` directory. Create subdirectories under `Lab04` if you want.

Lab Procedure

Step 1. Use the following examples to model three different D flip-flops in a single verilog module. Add an assertion that warns when preset and clear are asserted simultaneously. Check your design by simulating it; then, synthesize it, optimizing first for area and then for speed.
Simple D flip-flop:

```
always@(posedge clk1) Q <= D;
```

D flip-flop with asynchronous clear:

```
always@(negedge clk1, posedge clr)
  begin
  if (clr == 1'b1)
       Q <= 1'b0;
  else Q <= D;
  end
```

D flip-flop with asynchronous preset and clear:

```
always@(posedge clk2, negedge pre_n, negedge clr_n)
  begin
  if      (clr_n == 1'b0) Q <= 1'b0; // clear has priority over preset.
  else if (pre_n == 1'b0) Q <= 1'b1;
  else                    Q <= D;
  end
```

Step 2. Write a verilog model of a D flip-flop with synchronous clear. Simulate it to check your design.

Step 3. Use the following examples to model three different D latches corresponding to the flip-flops in Step 1. You may wish to copy your design from Step 1 and modify the flip-flop code to represent latches. Decide which "simple D latch" to use from the examples below. As in Step 1, add an assertion that warns when preset and clear are asserted simultaneously. Check your design by simulating it; then, synthesize it, optimizing first for area and then for speed. Check this netlist by inspection of the schematic to be sure that it indeed models three latches.

Simple D latch:

```
always@(D) if (ena1==1'b1) Q = D;
```

The ena1 variable is declared somewhere and controlled externally; for example, ena1 might be a module input, as probably would be D.

What would happen if the sensitivity list was always@(ena1)?

What kind of simple D latch would be the following?

```
always@(D, ena1) if (ena1==1'b1) Q = D;
```

D latch with asynchronous clear and with enable asserted low:

```
always@(D, clr, ena_n)
  begin
  if (clr == 1'b1)
       Q = 1'b0;
  else if (ena1_n==1'b0) Q = D;
  end
```

D latch with asynchronous preset and clear asserted low:

```
always@(D, pre_n, clr_n, ena2)
  begin
  if (clr_n == 1'b0) Q = 1'b0; // clear has priority over preset.
  else if (pre_n == 1'b0) Q = 1'b1;
  else if ( ena2 == 1'b1) Q = D;
  end
```

Step 4. Serial-load shift register. A shift register shifts on every clock if shifting is enabled; otherwise, it holds previous data. See the schematic in Fig. 2.3.

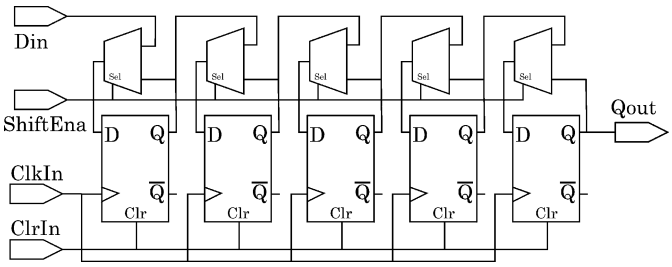


Fig. 2.3 Shift register with serial load

Write a verilog model of a D flip-flop with asynchronous clear, in its own file, and use it to construct a 5-bit shift register with serial load. Your flip-flop should have Q and Qn outputs. This serially loaded shift register should have one D input;

its contents can be loaded by shifting data for 5 clocks; or, it can be cleared asynchronously. To allow the shift register to hold its data while the clock still is running, use a multiplexor ("mux") to supply each flip-flip `D` input with one of these two possible inputs: (a) With shift enabled, the mux should connect the previous `Q` to the next `D`; or, (b) with shift disabled, the mux should connect each flip-flop's `D` input with its own `Q` output.

The schematic representation of a 2-input mux is shown in Fig. 2.4.

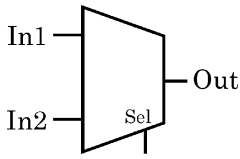


Fig. 2.4 Schematic mux symbol

Your mux model may be written this way:

```
always@(sel, in1, in2)
  if (sel==1'b0)
       outbit = in1;
  else outbit = in2;
```

A mux is combinational, not sequential, logic, so blocking assignments are fine. Notice that if either of `in1` or `in2` had been omitted from the sensitivity list, a latched state would exist, and we would not have a mux but rather a latch of some kind.

Another way to write a mux (***not*** in this Step, please):

```
assign outbitWire = (sel==1'b0)? in1 : in2;
```

The D flip-flop model should be in a separate file from the shift-register, and so should be the mux model.

Simulate your design briefly to check it for functionality.

Optional: Synthesize your design twice, optimized for area first and and then for speed.

Step 5(optional). Parallel-load shift register. Modify the design from Step 4 so that all 5 bits of the shift register can be loaded with new data on one clock.

The easiest way to do this is to add a third selection to your Step 4 muxes: The third choice should connect each flip-flop `D` input to its respective bit on a new, 5-bit, parallel-load input data bus. See Fig. 2.5.

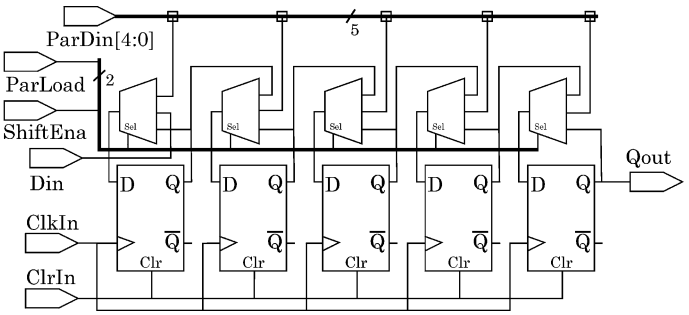


Fig. 2.5 Shift register with parallel load

In this parallel-load design, when shift is not enabled, each flip-flop `Q` is connected to its own `D`; when shift is enabled, each flip-flop `Q` is connected to the `D` of the next flip-flop; when parallel-load is selected, each flip-flop `Q` is unconnected (except the last one), and each flip-flop `D` is connected to a bit on the parallel data input bus.

Be sure that your new muxes always select exactly one of the three intended connections no matter how they are operated.

Step 6. Rewrite the serial-load shift operation of Step 4 as a ***procedural model*** in a single `always` block like this:

```
always@(posedge ShiftClock)
  begin
  QReg[0] <= Din;
  QReg[1] <= QReg[0];
  QReg[2] <= QReg[1];
  QReg[3] <= QReg[2];
  QReg[4] <= QReg[3];
  end
```

This kind of design will not require a DFF model at all. To add the Step 4 muxes, you may use conditional expressions on the right instead of unconditional `Qreg` expressions:

```
always@(posedge ShiftClock)
  begin
  QReg[0] <= (ShiftEna==1'b1)?   Din    : QReg[0];
  QReg[1] <= (ShiftEna==1'b1)? QReg[0] : QReg[1];
  QReg[2] <= (ShiftEna==1'b1)? QReg[1] : QReg[2];
  QReg[3] <= (ShiftEna==1'b1)? QReg[2] : QReg[3];
  QReg[4] <= (ShiftEna==1'b1)? QReg[3] : QReg[4];
  end
```

This shift register model can be implemented using just one always block and no design hierarchy. Incidentally, putting the same delay, say, "`#1`", in front of each assignment statement in the code above should delay each shift by that delay but not otherwise affect the logic. However, some simulators will not simulate the delayed statements correctly – another good reason not to use delays in procedural blocks.

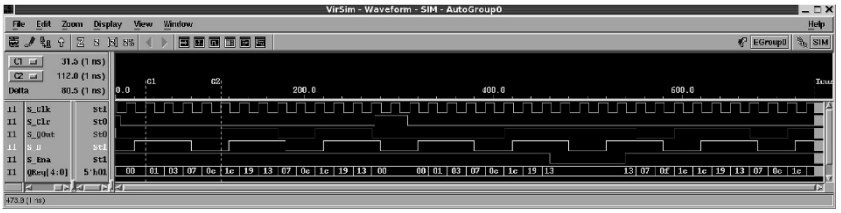


Fig. 2.6 Simulation of the behavioral shift register model above, with a 0.5 ns lumped output delay not shown in the verilog

Simulate to verify (see Fig. 2.6); then, try to synthesize the procedural model for area and speed. As it happens, the current version of the synthesizer may not synthesize nonblocking assignments correctly if they are delayed, but you should try anyway, to see what happens.

Then, just remove all the delays above and try synthesizing again.

What happens in simulation if the nonblocking assignments are replaced by blocking assignments in the code fragment immediately above? In the code fragment above, how much total time does it take for one shift?

You can synthesize a procedural shift register by removing the delays (above) or by changing the assignments to blocking ones. However, if you use blocking assignments, the statements then have to be ordered in reverse of that shown (with nonblocking assignments and equal delays as above, order is irrelevant, because the old value on the right will not be updated before it has been assigned).

To study the special features of nonblocking assignments, simulate the `Scheduler.v` model which you will find in your `Lab04` directory.

2.4.1 Lab Postmortem

Latches are a problem for the synthesizer. This is not unintentional; guess why?

Do you think the synthesizer can create better logic with a structural model than with a procedural one? Why?

How might the flip-flop models be improved? What about '`x`' or '`z`' states?

In the serial-load shift-register design, how might an assertion be added to check to make sure that shift was not disabled before at least five valid data had been loaded?

2.4.2 Additional Study

As previously assigned, read Thomas and Moorby (2002) chapter 1 on basics and chapter 3 (on synthesizable verilog) through section 3.4. Do Thomas and Moorby (2002) 1.6 Ex. 1.2.

Read Thomas and Moorby (2002) chapter 2 on logic synthesis. We shall study finite-state machine design in verilog later, so read through the FSM sections lightly.

For inferred latches, study Thomas and Moorby (2002) Example 2.7 in section 2.3.2 and do 2.9 Ex. 2.2.

Optional: Read Thomas and Moorby (2002) section 5.2 on parameters, but ignore `defparam`.

Optional Readings in Palnitkar (2003)

Section 9.2.2 on parameters.
Sections 3.2.9, 3.3.1, and 9.5.3 for details on strings and messages.
Sections 14.4–14.6 on logic optimization,

If you are puzzled by procedural controls, read through chapter 7 to see if explanations there might help. You may find the Palnitkar CD answers to exercises in section 7.11 beneficial; however, use of `forever` or `while` in the coursework, like use of `always` without an event control, generally is discouraged.

Read section 15.15.2 for an overview of assertion-based verification, which is very important in modern, large, complex designs.

Chapter 3
Week 2 Class 1

3.1 Net Types, Simulation, and Scan

3.1.1 Variables and Constants

As previously mentioned, **variables** are of two different kinds, `reg` and net. However, the situation is more complicated.

A `reg`, an unsigned type, is about the same as the corresponding signed types, `integer` and `real`. Any of these either of `reg`-like types can be assigned a value procedurally and retains the value assigned until the value is changed by a subsequent assignment. Both `integers` and `reals` are 32 bits wide; a `reg` may be any width, from one bit up to the limit the tool in use can accept. For flexibility, a `reg` may be declared *signed* in *verilog-2001*, in effect allowing declaration of integers of any width. A `real` is not synthesizable in the digital design tools we use, and we shall ignore it in most of the rest of this course.

The word *net* does not refer to a specific type but rather a generic characteristic of connectivity. The type of a net may be `wire`, `tri`, `wand`, or `wor` – or other types to be studied later. A `wire` and a `tri` are functionally identical, and the different names are just mnemonics. Multiple drivers on a `wire` may be a design error; on a `tri`, they probably are three-state drivers. A `wand` almost always is multiply driven, and it drives inputs to which it is attached with the ***wired and*** of its drivers; a `wor` drives with the ***wired or***. Thus, a `wand` or `wor` effects a logic operation in addition to a connection.

Multiple drivers on wand or wor are effected by using two or more concurrent assignments. This can be done (*a*) by wiring the net to two or more instance output pins, or (*b*) by two or more continuous assignment statements to the net. It should be mentioned that neither `wand` nor `wor` is used often in modern cmos design.

Constants may be literal numerical values, parameter values, or string values.

Because `integers` already are exactly 32 bits wide, they may be assigned constant values without specifying width. However, it usually is wise to provide a width for every literal constant; this means that operations on the literal will have unambiguous width, making the results well defined. Examples are the four fundamental 1-bit states,

`1'b1`, `1'b0`, `1'bx`, and `1'bz`; or, typical hex or binary expressions such as `7'h15` and `16'b0001_0101_1100_1111`.

A `parameter` value is by default unsigned. When initialized by a decimal integer literal, a `parameter` becomes 32 bits wide by default; however, a `parameter` may be declared of any width. Examples are

```
parameter x = 5, y = 11; // unsigned; 32 bits wide.
parameter[4:0] z = 5'b11000; // 5 bits wide.
```

There is no string type in verilog, but string literals enclosed in quotes may be used in simulation screen messages or assertions. A string of bytes also may be assigned to a `reg` of adequate width, but this usage is not common in digital design except when programming error messages into a ROM or RAM. A typical simulation example is,

```
$display("Error at time=[%04d].", $time);
```

There are no "global" variables or constants in verilog; all must be declared somewhere within a `module`.

3.1.2 Identifiers

Identifiers in verilog, which is to say, declared names of variables or constants, or names of modules, blocks or other objects, are:

- made of ASCII alphanumeric characters and '_'
- case-sensitive
- of any length allowed by the tool in use
- begin with a letter (alpha) or '\'.

Identifiers beginning with '\' are called *escaped identifiers* and may contain any ASCII character except a blank. These identifiers are terminated by a blank (' '), which is a delimiter and not part of the name. Escaped identifiers are used by tools to avoid name conflicts for translation or portability and usually are not written manually in verilog design source code.

3.1.3 Concurrent vs. Procedural Blocks

Concurrent blocks. These are blocks of code which simulate in no well-defined order relative to one another. Verilog source files, for example, may be compiled in a particular order, but the compilation order does not define the order of simulation. The most important concurrent block is the `module` instance. Likewise, continuous assignment statements, `initial` blocks, and `always` blocks are concurrent within a `module`. Other kinds of concurrent block, including `primitives` and the `specify` block, will be introduced later in the course.

Procedural blocks. These are blocks of code within a concurrent block which are read in order (sequence) during simulation and, if executed, are executed in the order read. Procedural blocks may contain:

- blocking assignment statements
- nonblocking assignment statements
- procedural control statements (`if`; `for`; `case`)
- `function` or `task` calls (later)
- event controls (‘`@`’)
- `fork-join` “parallel” statements (later)
- nested procedural blocks enclosed in `begin` ... `end`.

3.1.4 Miscellaneous Other Verilog Features

Macroes are like *C* preprocessor directives but begin with ‘`` ` ``’ instead of ‘`#`’. They are not strictly language elements, because they do not relate to other specific constructs and may appear anywhere on a new line in the code. For example,

```
`define, `timescale, `ifdef, `include
```

System tasks and system functions will be covered lightly in this course. They are simulation constructs and may appear in procedural blocks, only. For example, `$strobe`, `$display`, `$sdf_annotate`, `$stop`, `$finish`.

Timing checks will be covered in detail later. They are predefined assertions which execute concurrently. They may appear in `specify` blocks only. Examples are `$width`, `$setup`, and `$hold`.

The **PLI** (*P*rogramming-*L*anguage *I*nterface) is a library of *C* routines which allow a user to define new system tasks or functions which extend the functionality of a verilog simulator. We shall discuss this toward the end of the course but shall not use it at all.

3.1.5 Backus-Naur Format

(BNF)This is a way of defining syntax of a language and is used widely in verilog and other standards contexts. It gives an alternative view to text specifications of language syntax and helps to resolve ambiguity. We shall not use it in this course, but it is nice to know. The Thomas and Moorby (2002) appendix G treats it extensively.

The BNF rationale is extremely simple and essentially hierarchical: The allowed substructure of any primary element of syntax is provided in a list following “::=”; subelements are broken down following “:=”. For example, suppose for simplicity’s sake the BNF for a variable was,

```
variable ::= reg | net
         net := wire | tri | wor
```

Then, from this, we may deduce that a variable always will be either a `reg`, `wire`, `tri`, or `wor`. An attempt to use a `parameter` as a variable would be an error easily recognized from the BNF given in this example.

3.1.6 Verilog Semantics

Verilog is an HDL, and its meaning is hardware. So, in a VLSI context, verilog means logic gates and wires or traces. The logic gates correspond to `regs`, `module` instances, `integers`, and so forth; the wires to *net*s of various types.

Like any programming language, verilog works by expressions and statements. An ***expression*** is just something that can be evaluated (represents a value). An example of an expression is a sum or logical product, for example "`5+7`" or "`A&&B`". An expression just evaluates to something; it doesn't change anything.

However, a ***statement*** assigns a value to something and thus changes it. The changed value either is stored locally, or it is routed somewhere else. For example, in a procedural block, "`Zab = A && B;`" assigns the logical *and* of `A` and `B` (an expression) to a `reg` named `Zab`. The two equivalent concurrent statements, continuous assignment "`assign Z = A && B;`" or component instantiation "`and and01(Z, A, B);`" change the value of a net type named `Z`.

Statements have to be controlled somehow, and the usual way is by simulation of a change in a variable. For example, "`assign Z = A&B;`" will be reexecuted every time the simulator simulates a change in `A` or `B`. An `always` block may contain numerous expressions and statements, and the designer must provide such a block with an event control ("sensitivity list"); the `always` block statements all will be reread only when a variable in the sensitivity list changes. For example, a block controlled by "`always@(A)`" will be reread only when `A` changes, even if one of its statements is "`Z reg = A&B;`". However, a block controlled by "`always@(*)`" will be reread whenever any variable in an expression in the block is changed.

Latches and Muxes. Assuming only level sensitivity (not edge sensitivity introduced by keywords `posedge` or `negedge`), the meaning of an incomplete sensitivity list in an `always` block is a latch of some kind; a complete sensitivity list usually means combinational logic such as a collection of combinational gates or mux. However, if a control construct such as an `if` or `case` causes a value to be ignored, a latched state can be created even with a complete sensitivity list.

For example, suppose in a module we have declarations of "`wire a, b, sel;`" and "`reg z;`". Consider the following four different `always` blocks:

```
always@(a, b, sel)
  if (sel==1'b1)
       z = a;
  else z = b;
```

```
always@(*)
if (sel==1'b1)
     z = a;
else z = b;
```

```
always@(sel)
if (sel==1'b1)
     z = a;
else z = b;
```

```
always@(a, b)
if (sel==1'b1)
     z = a;
else z = b;
```

The leftmost two then represent muxes, because all the variables in expressions are in the sensitivity list, and every alternative in the `if` is assigned to the one

output (z). The other two always blocks above represent nonstandard latches of some kind which are disabled when the variable(s) in the sensitivity list does not change. The rightmost one is sensitive to every variable on the right-hand side of a statement; thus, it represents a mux which is latched in an abstract sense: The selection is latched, not the output value.

Here is a nonstandard latch which contains no procedural control construct:

```
always@(a,v)    // typo!
  Z = a | b;
```

The value of z is latched against changes in b. This is the kind of latch often created by mistake by a typo in the sensitivity list.

A simple transparent latch can be modelled correctly in an always block by omission of the sel==1'b0 alternative in what otherwise would be a mux. The code is next, with the equivalent component schematic symbol in Fig. 3.1:

```
// verilog simple transparent latch:
always@(D,Sel)
  if (Sel==1'b1)
    Q = D;
```

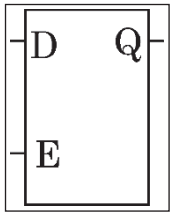


Fig. 3.1 Simple transparent latch by procedural omission. Sel renamed to E(nable)

The always block above is the preferred style for a synthesizable latch. The synthesizer will find a single component in the library for it, if the library has one.

Another way to write a synthesizable latch would be by continuous assignment:

```
assign Z = (Sel==1'b1)? D : Z;
```

Although the previous always-block model generally will synthesize to a latch library component, a continuous assignment latch typically would synthesize to explicit combinational logic with feedback.

The continuous-assignment latch might be synthesized as in Fig. 3.2:

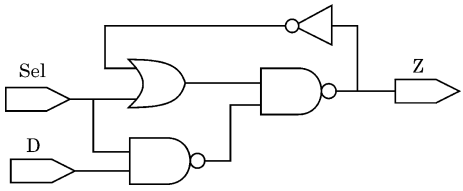


Fig. 3.2 Simple transparent latch by continuous assignment

A continuous assignment statement is treated as combinational logic by the synthesizer, so it does not attempt to find a library sequential element for the netlist. For this reason, it usually is not a good idea to use a combinational representation for a latch (although we shall use combinational S-R latches in our course work for instructional reasons).

When a latch is described functionally in combinational code, the precise gate structure is up to the synthesizer, and this structure may be changed unpredictably during optimization. Thus, the timing (including possible glitching) of a combinationally-implemented latch is more uncertain than when it has been described more directly in a simple sequential construct such as the `always` block above (represented by Fig. 3.1). It is best to avoid the whole problem, wherever possible, and write clocked constructs implying flip-flops instead of enabled constructs implying transparent latches.

3.1.7 Modelling Sequential Logic

We now turn to some aspects of model building in verilog which are intended to produce accurate synthesis results. For reasons to be expanded later, we avoid latches *per se* here entirely and discuss only *clocked* latching elements – which is to say, flip-flops.

Clocks. A clocked block in verilog must be an `always` block with an edge expression, `posedge` or `negedge`. A clocked block never should include more than one clock, and it should not include any level sensitivity. However, multiple edge expressions may appear if only one of them is applied to a clock. For example,

```
always@(posedge clk) ...
always@(negedge clk) ...
always@(posedge clk, negedge clear) ...
```

The following is not recommended and will not be synthesized:

```
always@(posedge clk, clear, preset) ...
```

Asynchronous controls. These usually are a single preset or clear but may be both a preset and a clear. If verilog is written to represent both preset and clear, the code can not avoid implying a priority for one of the controls. For example,

```
always@(posedge Clk, negedge Preset_n, negedge Clear_n)
  if (Preset_n==1'b0)
              Q <= 1'b1;
   else if (Clear_n==1'b0)
              Q <= 1'b0;
        else  Q <= D;
```

In this model, priority is given to `Preset` over `clear_n` if both are asserted in the same time interval. Thus, a correct netlist should include logic creating this priority, even though assertion of both probably would represent an operational error.

However, the designer's intent usually will be to synthesize a single gate with two pins for the controls, and either no priority (random priority) or some sort of internal gate structure effecting a priority.

Avoid more than one asynchronous control if possible: The block may not be synthesizable if the library does not include a component with both a preset and a clear pin; and, if it does, the simulation may not match the synthesized netlist if both controls ever should be asserted at once. It may be possible to pass a synthesizer a constraint or other directive which would control its library access so that cells would be chosen with specific preset-clear priority for particular instances. In the absence of such control, the synthesizer might insert logic to implement exactly the verilog simulation priority; or, it simply might ignore the verilog asynchronous control priority entirely.

Race conditions. A race condition results from code which allows an ambiguous value (? '1' or '0' ?) to be scheduled during simulation. This generally means that the corresponding hardware will not be functional.

For example, within one always block, delayed nonblocking assignments can allow concurrency and therefore a race:

```
always@(posedge Clk)
  begin
  #2 X <= 1'b1;
  #2 X = 1'b0;
  #3 Y = X; // Ambiguous value used.
  end
```

The same concurrency can occur between different always blocks:

```
always@(posedge Clk) #1 X = a;
always@(posedge Clk) #1 X = b;
```

To avoid the majority of race conditions, ***never*** mix nonblocking assignments with blocking assignments; and, ***never*** assign to the same variable from more than one `always` block. Incidentally, the synthesizer will refuse to synthesize these styles, although the simulator will permit them.

In addition, it is good design to latch all outputs on major design components (large blocks or `modules`): Flip-flops on all outputs guarantee that when the clock edge occurs, current values only, and not possibly conflicting ones, will be supplied to inputs of other components.

Finally, never schedule a delay of `#0`, except maybe in a testbench initial block. We shall study reasons for this later in the course. For now, just don't do this:

```
always@(posedge Clk)
  begin
  #0 Q1 <= a; // Likely error! Never use #0!
  #0 Q2 <= b; // Likely error!
  ...
  end
```

Synthesizable language subset. Not all the language is synthesizable. To be sure that what you simulate also will synthesize, here some rules of thumb:

- Don't mix edge expressions and level-change expressions in a sensitivity list.
- If you write a delay expression in an `always` block, use it with blocking assignments, only. The synthesizer will object to any delayed nonblocking assignment, either by reporting an error or by issuing a severe warning. We shall see why later.
- When you code delays for simulation, try to do it this way: Avoid delays in `always` blocks; instead, estimate the total delay(s) on the output of each such block, and move that estimated total to a continuous assignment statement. For example, code the way shown as follows:

```
module MyModule (output X, Y, rest of sensitivity list);
local declarations
...
assign #5 X = Xreg; // estimated total delay = 5.
assign #7 Y = Yreg; // estimate = 7.
...
always@(posedge Clock) // A very strange flip-flop!
  begin
  x1 = (a && b) ^ c;
  Xreg = x1 | x2;
  Yreg = &(y1 + y2);
  end
endmodule
```

3.1.8 Design for Test (DFT): Scan Lab Introduction

Modern tools automatically will insert scan into a design; this usually is done late in the design cycle, after the unscanned design has been well simulated. However, understanding the mechanics of scan insertion will help you to recognize and correct insertion errors or inefficiencies.

Scan is called "scan" because scan registers allow logic states at hardware test points to be ***scanned*** (shifted serially) in and out of a design.

The purpose of scan is to be able to observe changes inside a design or a whole chip. Scan registers are hardware test points; they allow a designer or test engineer to see what is going on in the hardware. Scan registers are not just simulation devices; they stay in the design after it is taped out and implemented in silicon.

There are two major kinds of scan, ***internal*** scan and ***boundary*** scan.

In *internal scan*, the inputs and outputs of all, or some selected, blocks of combinational logic in the design are changed into scan elements. This is done by replacing every flip-flop or latch with a scan flip-flop or scan latch. Obviously, every I/O of any combinational block must be either a chip pin or a sequential element such as a flip-flop or latch.

A special test port is used for controlling scan operation; this port is called a *JTAG* port ("Joint Test Activity Group") after the standards group that defined it.

The new scan components are the same as the ones they replace, except that they are muxed together into a serial scan chain, which is just a giant shift register distributed over all or part of the design. When the scan muxes are in the *operate* or *normal* mode, the design components operate as they were designed to do; when the muxes are in the *scan* mode, the design doesn't operate any more, but all logic values on the inputs or outputs of scanned blocks can be shifted out of the design and inspected for correctness. Thus, by scanning in new inputs, operating, and then scanning out the result, every combinational block in the design can be tested for correct operation, revealing design errors or (random) hardware failures, if any.

Figs. 3.3 through 3.5 illustrate how internal scan insertion is done in a design with preexisting sequential elements, in this case flip-flops.

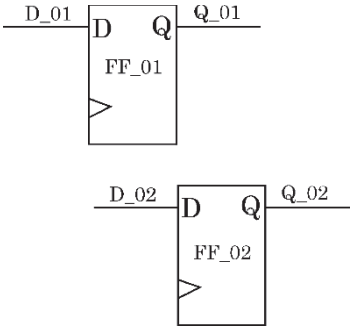


Fig. 3.3 Unscanned design

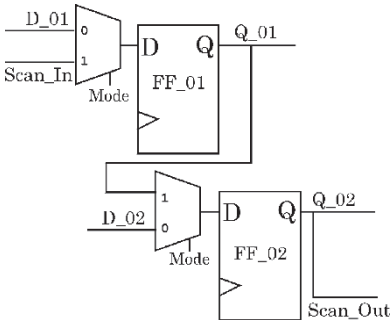


Fig. 3.4 Muxes inserted for internal scan

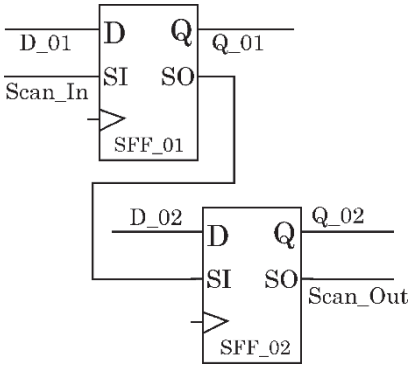


Fig. 3.5 Library scan components. Mode select not shown

In our lab exercise, we shall do this kind of scan insertion, but with some modification to allow for the fact that the original design does not include any sequential element to replace.

Boundary scan mostly is used for board-level hardware testing; it monitors the boundary of a chip. In boundary scan, the pins on a chip are connected to additional scan components which in turn connect to the gates inside the chip. This allows chip test patterns to be applied by shifting in, and the results observed by shifting out, after the chip has been mounted on the circuit board. Boundary scan thus eliminates the need for a "bed of nails", or other external hardware probe, on every test point on the board. A modern ball-grid chip package may have well over 1,000 pins spaced apart by a fraction of a millimeter; this makes a bed-of-nails approach impractical and possibly harmful to the tiny, delicate contact points.

Boundary scan usually is combined with some sort of built-in self-test protocol. In addition to a *JTAG* port, boundary scan typically requires a Test Activity Port controller (TAP controller), a built-in state machine, which is a complexity we shall ignore for now.

In the next lab, we'll use what we have learned about shift registers to insert scan flip-flops into the `Intro_Top` design of the first lab of the course. It should be emphasized that we shall be using ***two*** muxes per flip-flop, instead of the one required for normal internal scan. This is because the original design has no flip-flops, and we want to use the extra mux to remove the flip-flops functionally, not just scan them. So, we are using scan insertion as an excuse for an exercise in verilog.

Our JTAG test port will consist of just five special one-bit ports: A scan-***in*** port, a scan-***out*** port, a scan ***clock*** port, a scan ***clear*** port, and a scan ***mode*** port.

We want to insert scan into our old Lab 1 design just to see how it is done. Issues of clocking and settling of logic will be addressed again later in the course.

Before we start this lab, here is an outline of what we shall do:

First, we'll add a *JTAG* port and install flip-flops around the combinational logic in `Intro_Top`. Because `Intro_Top` was purely combinational, this means we'll install a flip-flop on every I/O. After installing the flip-flops, we'll restore functionality to the `Intro_Top` design by clocking the flip-flops. The only clock available is the *JTAG* scan clock, so we'll use that one. The rest of the *JTAG* port will remain unused at this point. The `Intro_Top` design then will become a synchronous design, clocked by the scan clock and with its original functionality intact, except for synchronizing delays.

Second, we'll install muxes, two for each flip-flop. Each mux at this point will be held in one *select* state (the *operate* or *normal* mode). For example, two `Intro_Top` outputs with muxes added to the new flip-flops would look as shown in Fig. 3.6.

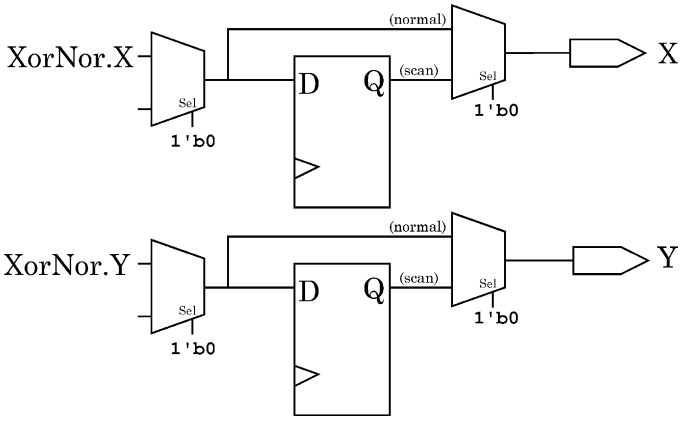


Fig. 3.6 Two outputs in `Intro_Top` after connection of muxes to the new flip-flops. Clock and clear logic omitted for clarity; '0' on *select* is assumed to select normal mode

If we put the muxes in the *scan* mode, the design would not do anything, because the scan inputs to the input muxes are unconnected at this point. So, to allow immediate verification of our verilog by simulation, we shall install the muxes with the flip-flops out of the design. This means that the `Intro_Top` design again becomes purely combinational, with no synchronizing delays, but with some small propagation delay added by the mux logic.

Notice that the design fragment in Fig. 3.6, held its *normal* mode, completely ignores the states of the flip-flops and does not respond to the scan clock at all.

Third, we'll connect the dangling mux *scan* inputs to each other in a serial chain; as will be seen in the lab, this will chain the flip-flops together and allow them to act as a shift register when the design is clocked in *scan* mode. All mux outputs will have been fully connected already; so, they need not be modified in this final step.

Now we begin the lab.

3.2 Simple Scan Lab 5

Lab Procedure

Log in and change to the `Lab05` directory.

You'll find there a copy of the original `Intro_Top Lab01` design, with minor changes. Notice the maintenance log that has been kept in each module file. This kind of commenting is recommended, not only to pass on information to other designers, but to remind the busy original designer, after a lapse of time, of what was done. The details to be logged will vary with your project coding style. There also is a synthesis script file there for use only in the final, optional Step of the lab.

Step 1. Add a *JTAG* test port in `Intro_Top`. Do this just by adding five new one-bit ports connected to nothing: Call them `ScanMode`, `ScanIn`, `ScanOut`,

ScanClr, and ScanClk. All should be inputs except ScanOut. Your module header now should be something like this:

```
module Intro_Top (output X, Y, Z, input A, B, C, D
                , output ScanOut
                , input ScanMode, ScanIn, ScanClr, ScanClk
                );
```

Step 2. Insert D flip-flops (FFs) into the path of every I/O at the top of the design (in Intro_Top.v, not TestBench.v), except the JTAG I/O's.

You may wish to use your FF design from Lab04; it should have a posedge clock and positive-asserted asynchronous clear.

To do this, work with the module ports and wires in Intro_Top; there is no need to change anything in the submodules which made up the structure of the original Lab 1 design.

Connect every preexisting design input to the D of a new FF; then, reconnect the Q output of that FF to whatever the preexisting input pin was driving. You should declare a new wire for each FF Q connection.

You may find it confusing and easy to make mistakes if you do not name your wires and component instances mnemonically. For example, suppose your FF module was named DFFC ("*D flip-flop with clear*"). For the top-level A input port, which connected in Lab 1 to pin A of a block with instance name InputCombo, you might do this:

```
wire toInputCombo_A;
...
DFFC Ain_FF(.Q(toInputCombo_A), .D(A), ...);
```

After the inputs, do the same for the outputs. In Intro_Top, connect the Q output of a new FF to every preexisting design output; then, reconnect whatever was driving that preexisting design output to the D input of the new FF. Again, declare a new wire for each reconnection.

After wiring the data for all FFs, connect the ScanClk input to the clock pin of each FF; connect the ScanClr input to the asynchronous clear pin of each FF. All your FFs then should look something like this:

```
DFFC myFFinstanceName (..., .Clk(ScanClk), .Clr(ScanClr));
```

Add drivers for the new ScanClk and ScanClr inputs in the testbench in TestBench.v; be sure to connect them to the top-level design instance there.

A good testbench convention throughout this course is to declare a reg for every design input to be driven; name this reg *stim, where '*' is the design input name. For example, Intro_Top input A would be driven by a testbench reg named Astim. Similarly, a good convention for design outputs is to declare a testbench wire for each one, named *watch. For example, the design output X would be connected to a testbench wire named Xwatch. You *stimulate* the inputs

and *watch* the outputs. Being consistent with testbench names makes it easy to recall which testbench variable was connected with each design variable without having to examine the testbench code itself.

After this, clocking the FFs after every testbench stimulus change should apply the testbench `*stim` simulation inputs to the original design and should clock out the current (not new) `*watch` outputs. After waiting enough simulation time for the combinational logic to settle, clocking the FFs a second time *without* changing inputs should clock out the new, correct `*watch` outputs determined by the current inputs and the design's combinational logic.

Compile your changed design for simulation now, from testbench on down, to check your FF and clock wiring.

Step 3. Refine the timing, if necessary. Looking at the subblocks of the `Lab05` design, there are several delays of 10 units; so, be sure to have defined the clock in your testbench with a generous period of, say, 50 units. This should allow plenty of settling time:

```
...
always@(ClockStim) #25 ClockStim <= !ClockStim;
//
initial
  begin
  #0 ClockStim = 1'b0;
  ...
  end
```

The `ClockStim` in the testbench of course will be connected to the `ScanClk` port of the modified design. And you will want a new `ClearStim` in the testbench to drive the design `ScanClr`. Just set `ClearStim` to 0; there is no need to reset the flip-flops for this lab. Check your clocking scheme by simulating the design briefly. See Fig. 3.7.

You now have a new, synchronous design consisting of the Lab 1 combinational logic surrounded by sequential logic. It is synchronous because its operation is synchronized to `ScanClk`. Inputs are clocked in on every positive clock edge; the result is clocked out on the next positive edge.

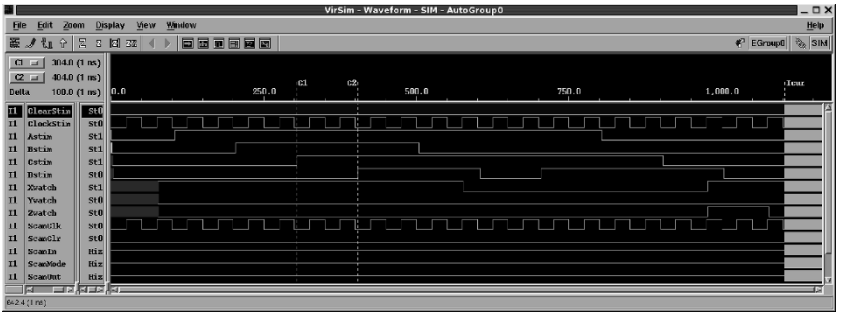


Fig. 3.7 The Step 3 synchronous implementation of `Intro_Top`

Optionally, synthesize this design, just to see what the netlist schematic looks like.

Step 4. Install multiplexers to remove all synchronous behavior again. Just the muxes; don't chain anything yet. Use an `always` block or a continuous assignment in `Intro_Top` for a mux, not a (structural) component.

Each FF will require a mux driving its `D` input, to connect its `D` either to the normal `D` input or to the scan chain. A second mux will be used to connect or disconnect the FF to its normal output. The second mux is required for this design; otherwise, the FF's `Q` output would contend with whatever normally is driving the next `D` input.

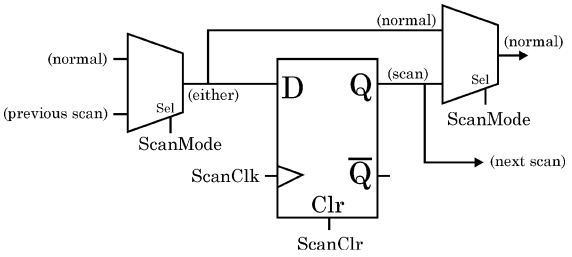


Fig. 3.8 Multiplexers around a flip-flop in `Intro_Top` to make a scan element. Usually, scan requires one mux per flip-flop

At this point, the leftmost "(previous scan)" input as shown in Fig. 3.8 should be driven with a constant such as `1'bx`; the "(next scan)" wire shown should *not* be added yet. It may be helpful to declare a wire with a noticeable name, such as `THIS_IS_X`, assign a `1'bx` to it, and use it in turn to assign the `1'bx` to the several mux inputs which will be set unknown at this stage in the lab.

As a result of all this, the FF shown in Fig. 3.8 is bypassed entirely in normal mode by its two muxes; so, in normal mode, we end up with our original, purely combinational, Intro Lab 1 design.

While adding code for the muxes, make the select input of every mux work so that when `ScanMode` is '`1`', the FF is in the scan chain; when `ScanMode` is '`0`', the FF is in the normal operating mode. Assign a small delay to each mux statement; say, `#1` or so.

Simulate the design with `ScanMode = 0` constantly (assign it in the testbench); the result should be the same as it was in Lab 1 without muxes or FFs, except for the delay added by the muxes.

We now have Lab 1 working again, but with extra, nonfunctional FFs and muxes which merely add a delay.

Notice that we have made no change at all in any submodule. Our entire original design was combinational, so we inserted the scan elements only at the top of the design. The result looks the same as a boundary scan, but it actually is an internal scan on a design which doesn't have internal sequential gates.

Optionally, synthesize this design, just to see what the netlist schematic looks like.

Step 5. Create a scan chain. At this point in the lab, the FFs are short-circuited out of the design by muxes, and they have no function. The scan inputs to all muxes are

unconnected to anything in the design. Therefore, the FFs can be connected together into a shift register, using the mux scan inputs, without affecting the design.

Refer to Fig. 3.8. Pick any FF and connect its `Q` to the design `ScanOut` port. After that, connect the `Q` of any remaining FF to the scan input pin of the mux driving the `D` input of the FF the `Q` of which just was connected. Repeat this until all FFs are chained together. The net result of this Step should be that the mux inputs which were unknown in the previous Step now are connected to FF `Q` drivers, except the first mux, which still is unknown. At this point, the `ScanOut` port should be wired to the `Q` output of the last FF in the scan chain.

Finally, connect the scan input pin of the mux driving the `D` input of the last remaining FF to the `ScanIn` port, making this the first FF in the scan chain. The result of all this may be represented as in Fig. 3.9, which omits the `ScanMode` and `ScanClr` wiring for simplicity:

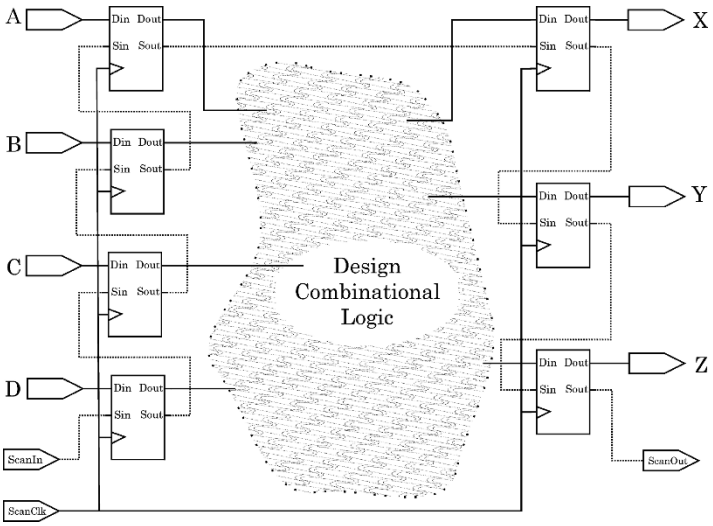


Fig. 3.9 Logical representation of Lab 5 scan insertion. In normal mode, each `Din-Dout` logically is a wire, and `Sin-Sout` is an open. In scan mode, `Sin` is a FF `D` input, `Sout` is a FF `Q` output, and `Din-Dout` is open. `Dout` and `Sout` are wired together, but this is irrelevant to the mode differences

Recognize the scan chain on the dotted line of Fig. 3.9? It's just a shift register!

Step 6. Assign useful instance names. Rename the flip-flop instances so that their order in the scan chain is represented numerically: For example, `"A_FF_s01"` might be a good name for the FF on the design `A` input, if it were set up as the first one (`"s01"`) in the scan chain; `"FF_X_s04"` might be the FF driving the `X` output, if it were fourth in the scan chain.

Optionally, synthesize this design; then, try to trace the scan chain in the netlist.

Step 7. Add a simulation safety net. It takes 7 scan clock cycles to scan in or out the contents of the entire chain; generally, depending on FF ordering, it takes

fewer clocks to scan in new stimuli or scan out new test results. Assume, then, that operation in scan mode for more than, say, 8 clocks in a row may represent an error. Write an assertion in `TestBench` which triggers a warning if scan mode is asserted for more than 8 scan clock cycles in a row.

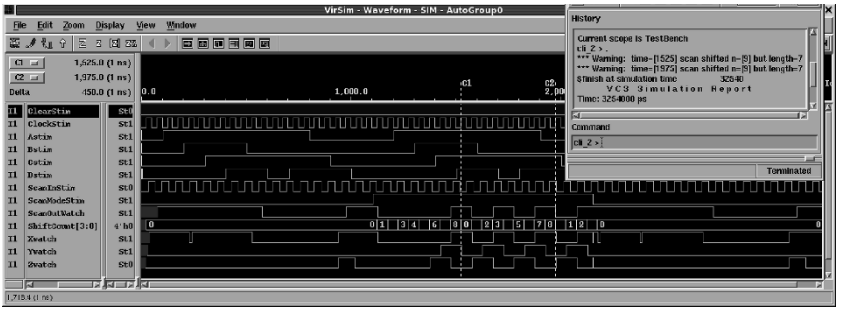


Fig. 3.10 Simulation of the completed Step 7 scanned `Intro_Top` design, with assertion safety net

Figure 3.10 shows a simulation with a testbench which invokes the safety-net assertion. This assertion, being in the testbench, will not be seen by the synthesizer and will not have any effect on a synthesized netlist.

Step 8. Simulate your design to be sure you understand it. Synthesize it (from the top level, not the testbench) for area. Examine the optimized netlist carefully; try to find and count the multiplexers.

Optionally, after synthesizing for area, without exitting the synthesizer, ungroup (flatten) the design and `compile` (synthesize) it again with an incremental-mapping option. Finally, still without exitting the synthesizer, do another compile on the flattened netlist for best area. The areas should improve progressively.

Step 9. (***optional***) The synthesizer can insert scan automatically in a design by replacing sequential component instances with scan instances, for example replacing plain flip-flops with muxed flip-flops. However, to use this feature, there must be preexisting sequential components. We shall start by discarding the manually scanned `Intro_Top` design and going back to the original combinational one.

Keep your current `TestBench.v`, but copy in a new `Intro_Top.v` from `Lab01`. Interpose a D flip-flop as an output latch between each `Intro_Top` output driver and its top-level output port (`X`, `Y`, and `Z`); use instances of your own DFFC model, but modified so that the Qn outputs are removed. These three FFs will be the required sequential elements. Add a clock input port named `Clk` and a clear input port named `Clr` to operate the FFs. See Fig. 3.11.

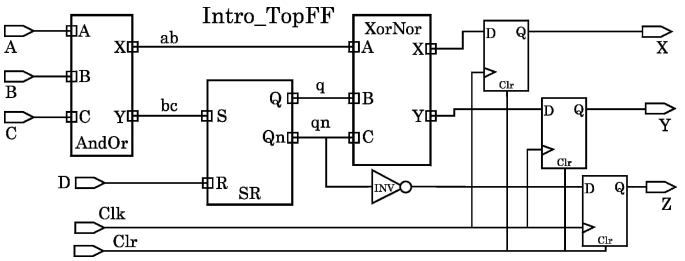


Fig. 3.11 The `Intro_Top` design with latched outputs permits automated scan insertion

Then, add a set of three unconnected JTAG ports with these standard names, `tms` (test-mode select), `tdi` (test-data in), and `tdo` (test-data out). The other two required JTAG ports, `tck` and `trst`, will be shared with the `Clk` and `Clr` ports, respectively.

After this, rename the top module `Intro_TopFF`. Then, modify `Testbench.v` as necessary to get the design to simulate normally, with outputs latched.

You should use the script originally prepared for you in your `Lab05` directory to insert scan automatically by means of the synthesizer. After scan synthesis, examine the synthesized netlist briefly in a text editor or `design_vision`.

3.2.1 Lab Postmortem

In scan mode, with our design, each shift easily might cause the inputs to the combinational logic to change, leading to a lot of logic changes and possibly glitches. Is this of any concern?

How might the operational protocol be defined so as to minimize combinational-logic changes during scan chain shifting?

How might the scan chain be modified so the design combinational logic remains in a constant state whenever it is in scan mode?

3.2.2 Additional Study

Read Thomas and Moorby (2002) sections 4.1 on concurrency, and 4.2 on event controls.

Read the example in 4.5, 4.6 on disabling named blocks, and (optionally) 4.9 on `fork-join`. In section 4.9, note that `fork` and `join` may be used to substitute for `begin` and `end`, respectively. However, adding an explicit `begin` and `end` around every `fork-join` block makes the code more readable and more easily modified; in particular, `begin` and `end` are required to use `` `ifdef DC `` to make code with `fork-join` synthesizable.

Do 4.10 Ex. 4.1.

(Optional) Look through Thomas and Moorby (2002) appendix G on BNF until you understand how it can be used.

Optional Readings in Palnitkar (2003)

Read section 4.6.1 on coding verilog for logic synthesis.

Do Exercise 1 in section 14.9, synthesis of an RTL adder.

Do Exercises 2 and 3 of section 14.9.

Look through the on-disc synthesis example that comes with the Palnitkar CD-ROM (Chapter 14, ex. 1 directory). This example includes a presynthesized gate-level netlist and a verilog library of elementary component models.

Chapter 4
Week 2 Class 2

4.1 PLLs and the SerDes Project

This time, we'll design a PLL and a parallel-serial converter.

For the remainder of the course, we'll be working off and on to complete a SerDes (***Ser***ializer-***Des***erializer) design, such as is required in the *PCI Express* specification.

4.1.1 Phase-Locked Loops

A phase-locked loop (PLL) consists of an output *ClockOut* generator, a phase comparator, and a variable-frequency oscillator (VFO), for example, a voltage-controlled oscillator. A PLL input signal *Clock* is provided, and the phase-difference comparator adjusts the VFO's frequency whenever a phase shift is detected between the *Clock* and the VFO *ClockOut*. The result is that the VFO is kept phase-locked to the *Clock*. In theory, a PLL could lock in to any *Clock* input; but, in practice, the VFO and the rest of the PLL is designed so that the lock is accurate and almost jitter-free only in some narrow frequency range.

Although a PLL always locks in to the *Clock* provided, if the VFO output is used to clock a counter, and a specific counter bit or value is used for the *Clockout*, the VFO frequency will be higher than the counted-down *ClockOut* used for the lock. The direct VFO output then may be used to provide a frequency-multiplied clock nevertheless phase-locked to the original *Clock* input.

4.1.2 A 1 × Digital PLL

An important PLL application is clock latency cancellation. In this application, there is no multiplication, but the PLL is locked to a clock from a terminal branch of a balanced clock tree. This subtracts away the tree latency (clock insertion delay) and makes available a clock from the PLL at the clock-tree terminal branches which is very close to being exactly in phase with the original clock at its entry point on chip. See Fig. 4.1.

J. Williams, *Digital VLSI Design with Verilog*,

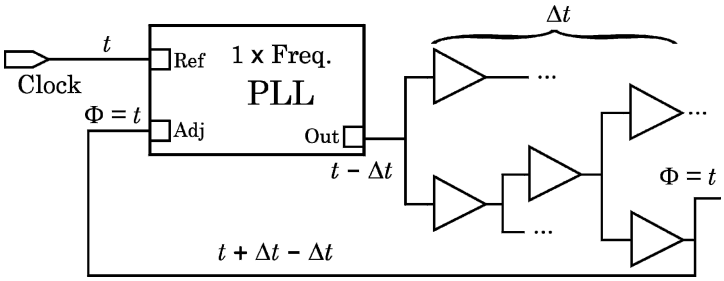


Fig. 4.1 PLL used to cancel clock-tree latency

Let us consider how we might design a verilog digital PLL for this purpose.

We don't want frequency multiplication, so we may assume two major components, a clock phase comparator and a VFO. A schematic representation is given in Fig. 4.2.

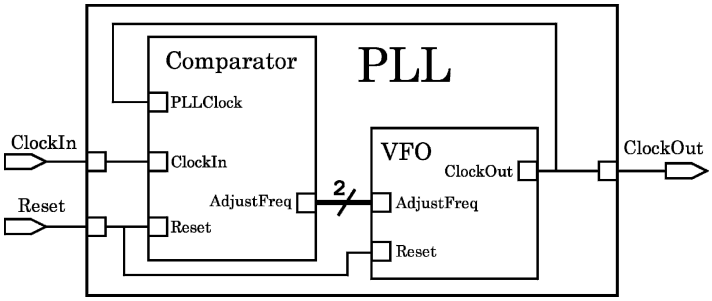


Fig. 4.2 Schematic of 1x digital PLL. The control bus from `Comparator` to `VFO` is assumed 2 bits wide

Happily, a variable holding a verilog delay value may be changed during simulation time; this makes feasible a finely-controllable VFO period derived simply from the value of a verilog `real` variable. The PLL will not be synthesizable; but, we don't have to worry right now about synthesis.

This kind of VFO frequency control may be represented by the following pseudocode:

```
real ProgrammedDelay;
...
begin
ProgrammedDelay =  some delay value;
...
#ProgrammedDelay PLLClock = ~PLLClock; // The VFO oscillator.
...
ProgrammedDelay =  some new delay value;
...
#ProgrammedDelay PLLClock = ~PLLClock; // Frequency varied.
...
end
```

Suppose now that we have decided that the Comparator adjustment code for a VFO frequency increase shall be 2'b11, and for a decrease shall be 2'b00.

Then, assuming we have defined a delay giving us the desired VFO base operating frequency, and assuming we have decided upon the size of a delay increment when the Comparator signals an adjustment, the actual verilog for the VFO would be about like this:

```
module VFO (output ClockOut, input[1:0] AdjustFreq, input Reset);
reg PLLClock;
real VFO_Delay;
assign ClockOut = PLLClock;
//
always@(PLLClock, Reset)
  if (Reset==1'b1)
       begin
       VFO_Delay = `VFOBaseDelay;
       PLLClock = 1'b0;
       end
  else begin
       case (AdjustFreq)
       2'b11: VFO_Delay = VFO_Delay - `VFO_Delta;
       2'b00: VFO_Delay = VFO_Delay + `VFO_Delta;
       // Otherwise, leave VFO_Delay alone.
       endcase
       #VFO_Delay PLLClock <= ~PLLClock; // The oscillator.
       end
//
endmodule // VFO.
```

Notice the use of blocking assignments everywhere, to ensure that new values are available immediately upon update; however, the VFO oscillator *must* use a *non*blocking assignment to oscillate. This was coded with great trepidation, and only after careful consideration of all consequences. This is a rare and unsynthesizable exception to the rule of never to mix blocking and nonblocking assignments in a single always block.

The Comparator is a bit more complicated. We require it to issue adjustment codes as described above for the VFO, but we want a decision with a minimum of logic so that we can run it very fast in 90 nm library components. A simple verilog design would just use one clock to count the number of highs of the other clock in one clock cycle; if there was just one such high, the frequencies would be approximately matched:

```
module JerkyComparator
       (output[1:0] AdjustFreq, input ClockIn, PLLClock, Reset);
reg[1:0] Adjr;
assign AdjustFreq = Adjr;
reg[1:0] HiCount;
//
always@(ClockIn, Reset)
  if (Reset==1'b1)
       begin
       Adjr = 2'b01; // 2'b01 or 2'b10 are no-change codes.
       HiCount = 'b0;
       end
  else if (PLLClock==1'b1)
            HiCount = HiCount + 2'b01;
       else begin
            case (HiCount)
                2'b00: Adjr = 2'b11; // Better speed it up.
                2'b01: Adjr = 2'b01; // Seems matched.
              default: Adjr = 2'b00; // Must be too fast.
            endcase
            HiCount = 'b0; // Initialize for next ClockIn edge.
            end
endmodule // JerkyComparator.
```

Blocking assignments again are used for immediate update. There is no reason to use a nonblocking assignment anywhere in this model. The various possible decisions by this comparator are shown as waveform relationships in Fig. 4.3.

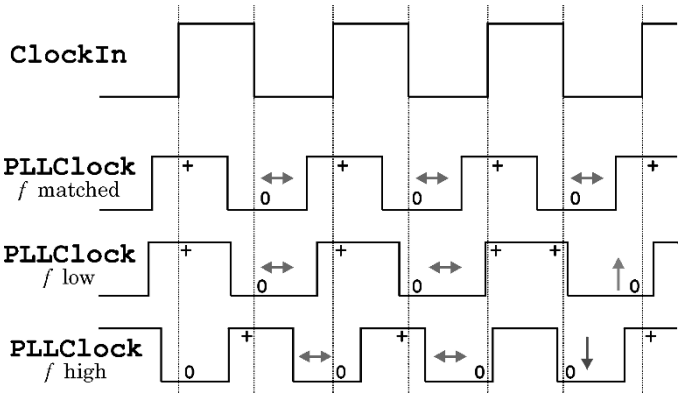


Fig. 4.3 Representative `Comparator` waveforms. Up arrow for `HiCount` $>$ `1`; down arrow for `HiCount` $= 0$; horizontal double-head arrow for `HiCount` $= 1$

This kind of model works in simulation, and we shall use one very much like it in our next lab. But, it is very slow to lock in, and it causes many spurious frequency adjustments. For the sake of a better lock-in, we can make several improvements:

- First, we should worry about a `HiCount` overflow and consequent wrap-around to `2'b00`, causing a spurious speed-up adjustment when the opposite is indicated. This is avoided easily by increasing the `HiCount reg` width from 2 to 3 bits.
- Second, we should improve the precision of the adjustment decisions. This can be done by averaging the high counts over several cycles; the expected value always should be 1 when the two clocks are synchronized.

For speed, we should take a finite-difference approach to averaging in a verilog model. We should declare a fairly wide `reg` to hold the average; and, in a separate `always` block, either add 1 to it or subtract 1 from it on every clock which, respectively, has an excess high or no high at all.

As a result of these improvements, we can obtain an optimized $1 \times$ digital PLL which does a respectable lock-in, not too much poorer than could be achieved with an analogue PLL. The code follows in two parts. The first part is the same comparator as in the previous code above – but, its result is used only internally. The second part averages the comparisons of the first in order to determine the frequency adjustment to be sent to the VFO:

```
module SmoothComparator
       (output[1:0] AdjustFreq, input ClockIn, PLLClock, Reset);
reg[1:0] Adjr;
assign AdjustFreq = Adjr;
//
reg[2:0] HiCount;  // Counts PLL highs per ClockIn.
reg[1:0] EdgeCode; // Locally encodes edge decision.
reg[3:0] AvgEdge;  // Decision variable.
reg[2:0] Done;     // Decision trigger variable.
//
always@(ClockIn, Reset)
  begin : CheckEdges
  if (Reset==1'b1)
       begin
       EdgeCode = 2'b01; // The value of EdgeCode will be used to
       HiCount = 'b0; //          increment or decrement AvgEdge.
       end
 else if (PLLClock==1'b1) // Should be 1 of these per ClockIn cycle.
            HiCount = HiCount + 3'b1;
       else begin // Check to see how many PLL 1's we caught:
            case (HiCount)
             3'b000: EdgeCode = 2'b00;  // PLL too slow.
             3'b001: EdgeCode = 2'b01;  // Seems matched.
            default: EdgeCode = 2'b11; // PLL too fast.
            endcase
            HiCount = 'b0; // Initialize for next ClockIn edge.
            end
  end // CheckEdges.
 // (continued below)
```

The second `always` block is decision-oriented:

```
// (continued from above)
always@(ClockIn, Reset)
  begin : MakeDecision
  if (Reset==1'b1)
       begin
       Adjr    = 2'b1; // No change code.
       Done    =  'b0;
       AvgEdge = 4'h8; // 7..9 mean no adjustment of VFO freq.
       end
  else begin // Update the AvgEdge & check for decision:
       case (EdgeCode)
       2'b11: AvgEdge = AvgEdge + 1; // Add to PLL edge count.
       2'b00: AvgEdge = AvgEdge - 1; // Sub from PLL edge count.
       // default: do nothing.
       endcase
       Done = Done + 1;
       if (Done=='b0) // Wrap-around.
              begin
              if ( AvgEdge<7 )
                   Adjr = 2'b11; // Better speed it up.
             else if ( AvgEdge>9 )
                   Adjr = 2'b00; // Must be too fast.
              else Adjr = 2'b01; // No change.
              AvgEdge = 4'h8;    // Initialize for next average.
              end
        end
    end // MakeDecision.
endmodule // SmoothComparator.
```

Notice the `case` statement in the `MakeDecision always` block: It lacks a default and does not cover all possible input alternatives. Therefore, a strange kind of latch is implied, and synthesis of this model should not be expected to be correct. This model would be considered a simulation-only place-holder if we were to adopt it for our class project.

This $1\times$ PLL design, with some minor changes, has been implemented for you in your `Lab06/Lab06_Ans` directory. The model was designed to be simulated with a 400 MHz clock input, which is close to the upper frequency limit possible eventually for synthesis in a 90 nm ASIC library. At this speed, the clock cycle time is 2.50 ns, with a `VFO` half-cycle delay of 1.25 ns. The testbench drifts the delay slowly upward.

Some waveforms are shown in Fig. 4.4 and 4.5:

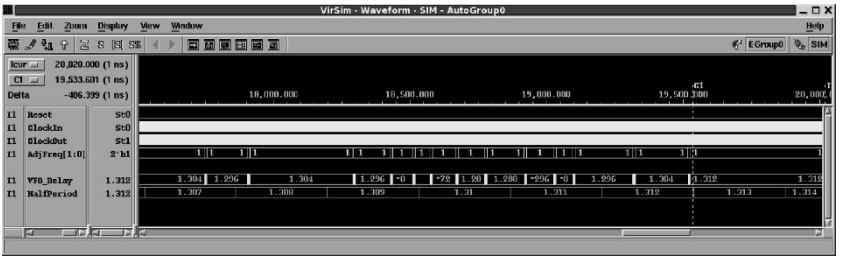


Fig. 4.4 The 1x PLL. Overview of the final 3 us of a 20 us VCS simulation

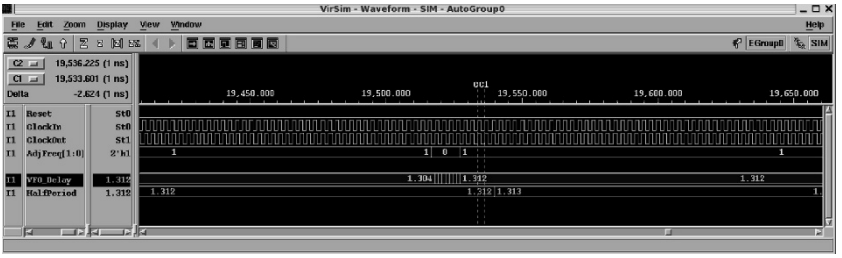


Fig. 4.5 Detailed closeup of the lock-in near the `C1` cursor position of the Overview waveform

More information on PLL's for latency cancellation, and details on static timing analysis of a design containing a PLL, may be found in the Zimmer paper in the References at the beginning of this book.

4.1.3 Introduction to SerDes and PCI Express

The *PCI Express* ("*PCIe*") bus is a serial bus meant to replace the 32-bit, parallel *PCI* (***P****ersonal* ***C****omputer* ***I****nterconnect*) bus common in desktop computers between the years of about 1995 and 2005. The serial bus sidesteps the growing *interbit skew problem* which is the main limiting factor in wide parallel busses clocked at high speed. *PCI Express* should not be confused with the *PCIx* parallel bus standard which merely doubles the clock speed of an ordinary PCI bus.

A PCI bus operating at 66 MHz can transfer $66^{*}10^{6*}32 \sim= 2^{*}10^{9}$ bits/s (2 Gb/s). By contrast, each PCI Express serial link (technically, called a ***lane***) can transfer 2.5 Gb/s in each direction simultaneously. This increase primarily is because of continual progress in the understanding of high-frequency digital operations in silicon. The past decade also has seen growing automation of the use of silicon monoliths to implement what in the past were strictly discrete analogue devices.

The first PCI Express specification, completed in mid-2002, allowed up to 32 PCI Express lanes, each at 2.5 Gb/s, for a total of 80 Gb/s in each direction. The current, second-generation PCI Express specification calls for 5 Gb/s per lane, in each direction, doubling the speed of the original generation. PCI Express, like the PCI bus, is an on-board link meant for short-range transfers of data, for example, transfers by the CPU to and from RAM or video or I/O-port controller. PCI Express not only is capable of being far faster than PCI, but it is much cheaper in terms of routing area on the board. However, data management of the serialization and deserialization makes PCI Express much more complicated to design.

For example, to see the speed advantage, a typical computer terminal has a resolution of 1280 $\times$ 1024 pixels. In full-color CMYK mode, there are 32 bits per pixel. Thus, the screen requires a buffer of 1280*1024*4 bytes, which totals about 5 MB. To refresh the screen at 70 Hz then requires a transfer rate of about 350 B/s A parallel video bus 128 bits wide, operated at 33 MHz, can transfer about 500 MB/s and thus can handle the screen update. However, such a bus would be running more slowly than a single lane in second-generation PCI Express and would occupy about ten times the area on a video card. A PCI Express video bus makes a lot of sense, both in terms of performance and economy.

There are many analogue issues involved with circuits operating in the GHz range. For example, each serial line actually is a differential pair of wires, making for four wires per full-duplex lane; for our digital purposes, calling each pair a serial line is accurate enough. The mechanical implementation of a PCI Express serial line may be a simple transmission line, a twisted pair, or even a coaxial cable. We shall ignore these issues for now and work on a serdes composed of routings and gates with all analogue difficulties assumed to have been designed away. Our serdes will include a complete, self-contained serial-parallel interface the components of which do not map exactly one-to-one with those of the PCI Express standard. Unlike a PCI Express design, when we are done, ours will be entirely digital and therefore, to that extent, technology-independent. Some optional readings have been provided in the References on the analogue side of the problem.

A serdes transfers data between two or more systems with busses of arbitrary (parallel) width. These local parallel busses are so short-ranged, that interbit skew is not specifically a design consideration. The data to be transferred are clocked off a parallel bus into a buffer of some kind, generally a FIFO (***F***irst-***I***n, ***F***irst-***O***ut stack memory), serialized (converted to a serial format), and transferred one bit at a time by the serdes to their destination. At the destination, the incoming serial data are deserialized (converted to a parallel format), buffered, and clocked onto the destination parallel bus. A great simplification is that the required precisely-synchronized common clock may be generated by a low-jitter PLL from the clock image encoded in the data. Clocking in this context is the same as the defining of data-frame boundaries. If the serdes is in a stable, on-board environment, clocking can be accomplished by using a PLL synchronized to the serial stream to generate a usable clock for buffering and format conversion at both ends. A single PLL may be shared if both ends of a lane are in the same clock domain; or, two independent PLL's may be used, one at the end of each lane.

4.1.4 The SerDes of this Course

The serdes we shall study operates, like the one in PCI Express, in a ***full-duplex*** mode; this means that transfers are possible simultaneously in both directions with no interference or sharing required. This is done simply by having two dedicated (verilog) data `wires`, each one capable of sending data in one direction.

Our serdes will convert 32-bit parallel data on each end to serial data in 16-bit frames. A full packet of data will be 64 bits. The 16-bit frame means that the embedded serial clock will change once per 16 bits transmitted. We shall transmit only one data byte per frame; this is less efficient than PCI Express, but it will allow the embedded clock to be extracted in an easily visible, orderly way. To serialize or deserialize the data, which must be processed one bit at a time, a PLL will be specified which multiplies the parallel-clock frequency by 32.

We shall do our design on the assumption of a relatively low parallel clock frequency of 1 MHz; our serial line then should transmit at 32 Mb/s, around 1/100 of the speed of either direction of a PCI Express lane. Because we require one pad byte for each data byte, a data packet containing one 32-bit word will occupy 64 bits in the serial stream. Thus, one framed data word is 64 bits wide, like this:

```
64'bxxxxxxxx000 11 000xxxxxxxx000 10 000xxxxxxxx000 01 000xxxxxxxx000 00 000.
```

The 'x' bits represent values in data bytes; the pad byte values are shown as-is.

This means that we should expect to transmit (and receive) one 32-bit word on every other clock. Packets on a PCI Express bus may be as large as 128 bits, but we shall not adopt this kind of formatting in our design.

After we complete our design, we'll see how fast it can be made to run by logic synthesis and optimization with the gate-level libraries available for course use.

Figure 4.6 gives a block diagram of the data flow in our serializer.

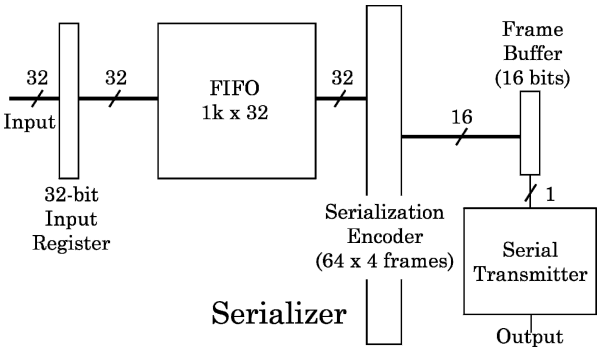


Fig. 4.6 Dataflow of the planned serializer

Our deserializer just reverses the transformations of the serializer and is shown in Fig. 4.7.

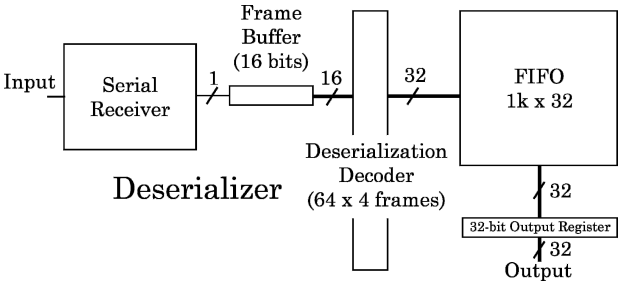


Fig. 4.7 Dataflow of the planned deserializer

In a PCI Express or other similar serdes, the input clock on the receiving end often is extracted from the serial data stream and converted to a (analogue-approximate) square-wave for input as *Clock* to the PLL. The *ClockOut* created by the PLL is controlled to match *Clock* precisely in phase, even if the PLL *ClockOut* is at a high multiple of the frequency of the input Clock.

Just as a matter of side interest, Fig. 4.8 gives an example of a real clock waveshape for a modern memory chip.

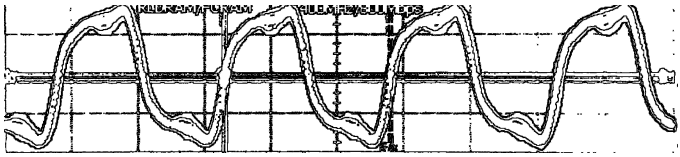


Fig. 4.8 A good 400 MHz clock. (From photo taken at the LSI Logic exhibit, *Denali MemCon 2004*)

This is a good clock running at 400 MHz; the vertical grids in Fig. 4.8 are about 300 mV/division. The waveform is the jitter envelope of many thousands of cycles. The PCI Express serial link would be a two-wire differential pair running at a peak-to-peak voltage of about 1 V. It's easy to see how analogue issues might arise, even at only 400 MHz.

4.1.5 A 32 × Digital PLL

In our next lab, we shall write a simple but unsynthesizable verilog model of a PLL designed to multiply frequency by 32.

We are constrained to keep to digital design, in this course; so, as for the 1× PLL above, we shall substitute a frequency lock for a phase lock in our serdes PLL.

Digital synchronization will provide the phase lock. To save coding time, we shall not bother with averaging for an improved lock. To cut down on comparator decisions in response to random phase misalignments, we shall generate an external sampling pulse; the VFO will run free and will adjust its frequency only when this pulse occurs.

We can force this design to be acceptable to the synthesizer just to see whether we can create of a netlist, but that netlist will not be functional. Later in the course, we shall redesign this PLL so it will be correctly synthesizable to a working netlist. In the meantime, we can use this easily-written, unsynthesizable PLL to clock the serial line while we are working on other parts of our serdes design.

The functionality of our PLL is represented by the block diagram of Fig. 4.9, with $n = 5$:

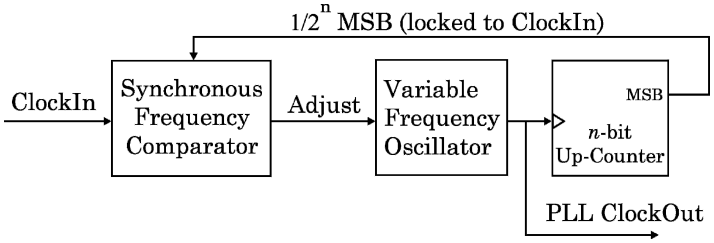


Fig. 4.9 PLL block diagram, showing `ClockIn` $\times 2^n$ frequency multiplication

A simple, resettable verilog 5-bit up-counter can be done this way:

```
reg[4:0] Count;
always@(posedge Clock, posedge Reset)
  begin
  if (Reset==1'b1)
       Count <= 5'h0;
  else Count <= Count + 5'h1;
  end
```

The lab instructions will include a schematic and further details.

4.2 PLL Clock Lab 6

Begin this lab by changing to the `Lab06` directory.

Lab Procedure

In this lab, we'll build a PLL and a related parallel-serial converter. Because a logic synthesizer can't synthesize delays (or delay differences), for synthesis purposes we normally would replace this PLL with a pre-synthesized hard macro or analogue IP ("Intellectual Property") block.

We shall introduce no #*delay* value anywhere in the PLL, other than testbench or clock-frequency delays.

To be systematic, we'll break down the PLL into three blocks, as explained above and shown in Fig. 4.10: VFO, Comparator, and Counter.

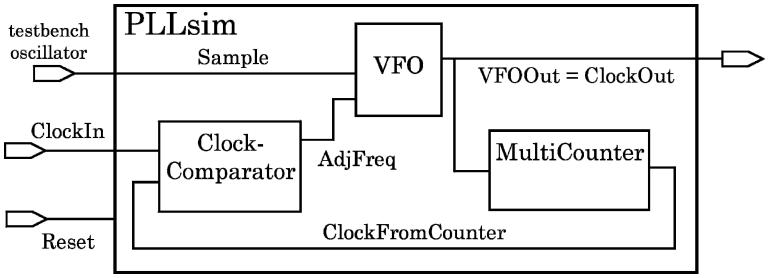


Fig. 4.10 Schematic of verilog `PLL`, showing wire names. `Reset` nets omitted

Step 1. Start by creating a top-level module named **`PLLsim`** (sim = simulation). Instantiate in it three submodules: a `VFO`, a `ClockComparator`, and a `MultiCounter`. As usual, put each module in a separate file named for that module. Give the top module just three inputs, `ClockIn`, `Reset`, and `Sample`, and one output, `ClockOut`.

Declare module header ports to match the blocks shown in Fig. 4.10:

- For the **`ClockComparator`**, declare `ClockIn`, `Reset`, and `CounterClock` input ports, and a two-bit `AdjustFreq` output port. We could use finer tuning, but two bits is enough for our purposes.
- For the **`VFO`**, declare a two-bit `AdjustFreq` input port, a `SampleCmd` and a `Reset` input port, and a `ClockOut` output port.
- For the **`MultiCounter`**, declare `Clock` and `Reset` input ports and a `CarryOut` output port.

Connect the ports in `PLLsim` with wires, consistent with the block diagram of Fig. 4.10.

You might as well add a testbench module in `PLLsim.v` and instantiate `PLLsim` in it; having a testbench early in the process will allow you to try out your submodules as you complete them.

Step 2. Model the PLL VFO. We shall start with a simple verilog model of a variable-frequency clock which adjusts itself in small increments using a two-bit correction flag on an input bus named `AdjustFreq` . A value of 1 on this bus means no change; a value of 0 means reduce the frequency; a value of 2 or 3 means increase the frequency of the VFO.

The parallel-bus clock of 1 MHz implies a VFO base frequency of 32 MHz; so, the VFO period should be 31.25 ns, and the half-period (edge delay) therefore should be 15.625 ns. These delays are fractional, but we can use only integer types (`integer` or maybe `reg`); therefore, we can't express anything less than 1 ns, so, we can expect our frequency control to be somewhat coarse.

In this preliminary, unsynthesizable PLL, we'll sample frequency occasionally (to be determined by the serial data in) using an external signal, `Sample`. We'll

design to be able to adjust our VFO by 1/16 of the half-period per sample, which comes to a little less than 1 ns per sample. So, on any `Sample`, if `AdjustFreq > 2'b01`, we'll decrease VFO period by about 1 ns; if `AdjustFreq < 2'b01`, we'll increase VFO period by about 1 ns.

It is allowed in verilog to declare a parameter to be `real` and/or `signed`, which would be useful in a PLL testbench. But, many tools won't let us use a `parameter` to store a `real` number, and declaring a `real` port would be a serious digital implementation problem, so, we'll have to be content to control the design base-point operating frequency with defined macro constants. To accomplish this, and for flexibility in compiling individual modules separately, you should put these lines in a separate include file, `PLLsim.inc`:

```
`timescale 1ns/100ps
`define HalfPeriod32BitBus 500.0 // ns half-period at 1 MHz.
`define VFOBaseDelay `HalfPeriod32BitBus/32.0 // At 32 MHz.
`define VFO_DelayDelta 1 // ns.
`define VFO_MaxDelta   2 // ns.
```

The last one is to prevent the PLL from running away or grinding to a halt: Use it in your VFO module to limit the frequency excursion from the base frequency.

Because defined macro constants can't be changed during simulation (actually, they are removed during simulator compilation and replaced by their values), these constants will have to be used to initialize verilog variables. The values above in `PLLsim.inc` will work whether the variables are synthesizable `integers` or unsynthesizable `reals`. Later in the course, we'll change this include file to use different constant values for reals (simulation) than for integers (synthesis).

Next, in any design file using these definitions, add this line above the module header:

```
`include "PLLsim.inc"
```

The only files requiring this are the top-level one, `PLLsim.v`, which propagates the timescale to the rest of the design during simulation, and the VFO one, `VFO.v`. Notice two things about the include definitions: (a) Given the 32-bit to 1-bit design itself, the values depend solely on the timescale and one assigned value, `` `HalfPeriod32BitBus ``. (b) the values involved in division include decimal points: This is required in verilog to force the delays to be calculated as real numbers. Without decimal points, the divisions and rounding would be done on integers, and the result would be less accurate in simulation. However, all variables involved have to be declared as integers for synthesis anyway, so the intermediate float calculations only have a small effect.

After this, in `VFO` declare the following variables (except `VFO_ClockOut`) as `integer` and add this reset block:

```
always@(Reset, SampleCmd, VFO_ClockOut)
if (Reset==1'b1)
     begin
     BaseDelay    = 'VFOBaseDelay;
     VFO_Delta    = 'VFO_DelayDelta;
     VFO_MaxDelta = 'VFO_MaxDelta;
     VFO_Delay    = 'VFOBaseDelay;
     VFO_ClockOut = 1'b0;
     end
else (below)
```

As a minor contribution to the lab exercise, complete the delay control of this VFO design by supplying the contents of the `else` half of the above `always` block, which is given in part below; this `else` applies the delay adjustment and determines the VFO half-cycle delay:

```
else // as above.
     if (SampleCmd == 1'b1)
     begin
     if ( AdjustFreq>2'b01
        && (BaseDelay - VFO_MaxDelta < VFO_Delay) )
            // If floor is lower than current:
            VFO_Delay = VFO_Delay - VFO_Delta;
     else if ( AdjustFreq<2'b01
              && (fill this in)
          // else, leave VFO_Delay alone.
```

Because of the `VFO_MaxDelta` limits, we have replaced the earlier `case` statement with two levels of `if`'s.

To generate the PLL clock from the delay determined above, we complete the model:

```
always@(Reset, SampleCmd, VFO_ClockOut)
  if (Reset==1'b1)
       ... (as above) ...
  else begin
       ...  (freq. adjustments) ...
       'ifdef DC
       // No delayed nonblocking assignments:
       #VFO_Delay VFO_ClockOut = ~VFO_ClockOut;
       'else
       #VFO_Delay VFO_ClockOut <=~VFO_ClockOut;
       'endif
       end // main else.
```

The oscillation inversion in the last assignment above absolutely requires a *non*blocking assignment for simulation; however, the synthesizer rejects delayed nonblocking assignments; whence the `‘ifdef` (the macro `DC` always is defined when the synthesizer runs). Some demo versions of Silos will simulate the oscillation if it was written with a blocking assignment, which actually is a verilog language error.

The reason the clock-generating `always` block is written to be sensitive to `SampleCmd` is because whenever such a command is asserted, we want the PLL clock to become synchronized to it.

After completing the VFO module, simulate it from `PLLsim`, putting a dummy clock temporarily into the comparator module to count 0 to 3 to trigger frequency adjustment events. Just do a simple simulation to check that the delay programming is working.

Step 3. Model the PLL comparator. The `ClockComparator` module, as in figure 3.10, compares an arbitrary input clock frequency to a variable clock frequency and issues an adjustment to make the variable frequency more closely match the input frequency.

In our design, as implied in this lab for the VFO model, the comparator will compare the frequencies continuously, but the VFO will not make any adjustment unless a `Sample` command requires it. We need not supply our comparator a `Sample` command, but we must supply a `Sample` command periodically to the VFO.

However, the `ClockComparator` must be supplied a reset to ensure its registers are in a known state.

How will our comparator work? There are several different ways to do a purely digital frequency comparison; we shall do it in this lab by counting variable-clock edges after every input-clock edge.

There will be three possible cases:

- If more than one variable-clock edge is found, the adjustment will be set to decrease variable-clock frequency.
- If just one variable-clock edge is found, the adjustment will be set for no variable-clock change.
- If no variable-clock edge is found, the adjustment will be set to increase variable-clock frequency.

To do this, declare a 2-bit register named `VarClockCount` to count incoming PLL clock (`CounterClock`) edges. The width of 2 bits allows a value of 3 to be counted, which exceeds the expected number of PLL clock edges per input clock, assuming that both clocks have very close to the same frequency and duty cycle. We wish to avoid a `VarClockCount` counter overflow under all reasonable conditions, but using a 32-bit verilog integer seems excessive.

It then is possible to use the following in the `ClockComparator` module to count the variable clock edges as below:

```
...
always@( ClockIn, Reset ) // This is the synchronizing clock.
    Begin
    if (Reset==1'b1)
         begin
         AdjustFreq = 2'b01;
         VarClockCount = 2'b01;
         end
    else // CounterClock is the clock to synchronize. Notice that it
         // is not on the sensitivity list; the inferred latch may be
         // expected to cause synthesis problems.
         if (CounterClock==1'b1)
              VarClockCount = VarClockCount + 2'b01;
         else begin
              case (VarClockCount)
                2'b00: AdjustFreq = 2'b11; // Better speed it up.
                2'b01: AdjustFreq = 2'b01; // Seems matched.
              default: AdjustFreq = 2'b00; // Better slow it down.
              endcase
              VarClockCount = 2'h00; // Initialize for next ClockIn edge.
              end
```

Notice that in this `always` block, we are monitoring every change of logic level of the external clock. We do not trigger a reading of this block on the clock from our PLL counter, because the PLL counter-clock is what we are using this `always` block to synchronize. It would be possible to use the `VarClockCounter` count directly as a control for the VFO, but the degree of freedom added by the case encoding allows us to modify the `ClockComparator` somewhat without making changes in the VFO, too.

We count every time a '`1`' is found for the PLL `CounterClock` when the synchronizing clock has gone to '`1`'. Whenever we find `CounterClock` to be '`0`', we check the count we accumulated since the last '`0`': If this count is 0, the `CounterClock` is assumed to be running too fast, so we request a speed-up; if it is 1, we seem to be approximately synchronized, and we do nothing; if it is above 1, the `CounterClock` is assumed to be running too slowly (its positive level was sampled twice), so we request a VFO slow-down.

Finally, complete the following lab Steps to determine the adjustment: These Steps simulate both `VFO` and `ClockComparator` together from a testbench which controls `PLLsim`. Be sure to reset `ClockComparator` to ensure correct simulation.

Step 4. Model the PLL Multiplier-Counter. This will be just a simple counter which toggles its overflow bit every 32nd clock. Thomas and Moorby (2002) describes counter modelling in detail (e. g., sections 2.2–2.6, 6.5.3, 6.7); we shall not dwell on the architecture until a later chapter.

Typical verilog for a simple behavioral up-counter is as follows:

```
reg[HiBit:0] CountReg
...
always@(posedge ClockIn, posedge Reset)
  if (Reset==1'b1)
       CountReg <= 'b0;
  else CountReg <= CountReg + 1'b1;
```

Note: A parameter is not allowed in verilog to specify the width of a literal. So, "`BitHi'b1`" would not be a legal increment expression. If we wrote, `CountReg <= CountReg + 1`, that would be OK, but it implies a 32-bit default integer increment. So, we size the increment, assuming that widening 1 bit to the width of `CountReg` would be easier on the compilers than narrowing 32 bits down to that width. This is speculative and really probably makes no difference.

Defining the "carry" bit as the MSB in the counter reg, to get a carry out every 32nd clock, our PLL's counter has to be exactly 5 bits wide ($2^5 = 32$). So, install a 5-bit counter in the `MultiCounter` module, and wire its highest-order bit to the `CarryOut` port. The carry bit will toggle every 16 clocks, giving it a period of 32 clocks. Use a behavioral counter sensitive to `posedge` clock and `posedge` reset.

Step 5. Test the complete PLL. With the counter wired into the top-level, the PLL now should be complete and functional. Use the testbench to trigger the top-level `Sample` with a positive pulse just after each positive edge of `ClockIn`. Verify the PLL by simulating from `PLLsim` (see Figs. 4.11 and 4.12). You should be able to change the frequency of `ClockIn` of Fig. 4.10 and see the `CarryClock` and the PLL `ClockOut` change (coarsely) to match it.

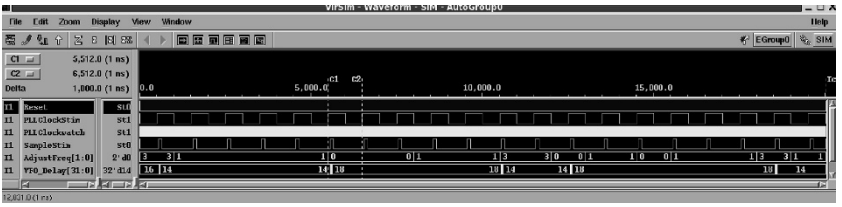


Fig. 4.11 Simulation of `PLLsim`. The "lock-in" of the VFO delay is coarse and extreme, but it does occur

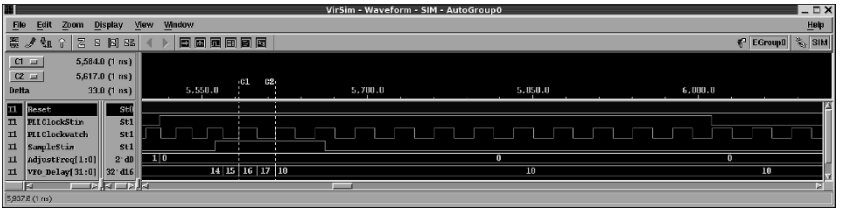


Fig. 4.12 Closeup of the `PLLsim` serial clock

This completes our work on the PLL for now; the rest of this lab is on other, but related, topics.

Note on the PLL VFO Block. In back-end design, a *hard macro* is just a block with fixed size and pin locations – for example, an IP block. This isn't a floorplanning or layout course, so we will not dwell on hard macroes. Later, to obtain a netlist, instead of substituting a macro, we can inhibit optimization of the VFO block using a synthesis don't-touch directive; or, we simply can tolerate a nonfunctional PLL in the netlist.

Any hierarchical block instance below the top level of a design can be preserved from flattening or optimization by the synthesizer by marking it with a ***don't touch*** comment directive (use your synthesizer summary pdf on the CD-ROM). Instead of a comment directive in the verilog, the ***don't touch*** command may be included in your synthesis script.

When the need for a `dont_touch` arises as a synthesis convenience, locating it in the script may be preferable to inclusion in a comment, because a directive in the script may be modified, or coordinated with other synthesis or optimization options, without editing the verilog in a design source file. However, when a `dont_touch` instance or net permanently is part of the design, locating the `dont_touch` in a comment, making it part of the design, may be the better choice.

This completes the verilog 32x `PLL`. We shall use it for serdes simulation until we redesign it for synthesis.

Step 6. Model a generic parallel-serial converter.

Call this converter, `ParToSerial`; we'll walk through the whole design. A parallel-serial converter requires knowledge of the width of the parallel bus. The clock input may be just a serial clock, `SerClock`. Assume the parallel bus contents are properly clocked and synchronized externally, and that there is an input flag, `ParValid`, to indicate when data on the parallel bus are stable and valid. See Fig. 4.13.

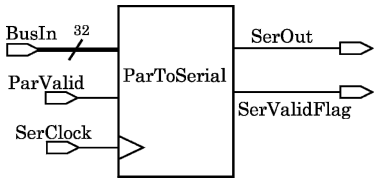


Fig. 4.13 Generic, minimal parallel-serial converter

Assume that when `ParValid` goes to `1`, the parallel data will remain valid until all serial data are clocked out, but don't clock out anything when `ParValid` is not asserted; also, don't clock out anything that has been already clocked out, no matter whether the parallel data are valid or not.

The serial protocol is simple: Clock out the data high-order bit (MSB) first, one bit per serial clock, setting a `SerValidFlag` when the first bit is on the serial bus, and clearing it after the last bit is on the serial bus.

We have to adhere to the rule that no data object should be assigned from more than one `always` block; not only is this good design practice, but it is required for synthesis. Because we must clock out by the serial clock, our one `always` block therefore must be sensitive to the serial clock.

A good plan is to use a `Done` flag to hold the state of the serialization. This makes the model an informal state machine: As soon as the `ParValid` is sampled asserted, move from a `Done` state to a *not*-`Done` state, and begin shifting out the serial data. If `ParValid` should be deasserted, or if the last parallel bit should have been processed, move back to a `Done` state. Remain in this `Done` state until a new `ParValid` is sampled (this means sampling a deasserted `ParValid` before leaving `Done`). Otherwise worded, `Done` must be cleared by sampling of a deasserted `ParValid`. We shall look into state machines later in the course.

Assuming a fixed parallel width of 32 bits, one way to do the verilog for this model is below.

```
module ParToSerial (output SerOut, SerValidFlag
                    , input SerClock, ParValid, input[31:0] BusIn);
  integer ix;
  reg SerValid, Done, SerBit;
  assign #1 SerValidFlag = SerValid;
  assign #2 SerOut = SerBit;
  always@(posedge SerClock)
    begin // Reset everything unless ParValid:
    if (ParValid==1'b1)
          if (SerValid==1'b1)
               begin
               SerBit <= BusIn[ix]; // Current serial bit.
               if (ix==0)
                    begin
                    SerValid <= 1'b0;
                    Done     <= 1'b1;
                    end
               else ix <= ix - 1;
               end // SerValid was asserted.
          else begin // No start yet:
               if (Done==1'b0)
                 begin
                 SerValid <= 1'b1; // Flag start on next SerClock.
                 ix       <= 31;// Ready to start on next SerClock.
                 end
               SerBit <= 1'b0; // Serial bit default.
               end
    else // ParValid not 1; reset everything:
         begin
         SerValid <= 1'b0;
         Done     <= 1'b0;
         SerBit   <= 1'b0; // Serial bit default.
         end // if ParValid else
  end // always
endmodule // ParToSerial.
```

Finish this up by introducing a parameter to set the parallel-bus width, and by writing a testbench. There is a copy of this model for you to modify in the `Lab06` directory, named `ParToSerial_unfinished.v`.

For this lab exercise, code the parallel-serial conversion module with a verilog `parameter` (ANSI style) which allows the width of the parallel input bus to be varied. Use the parameter inside the module, as well as in declarations, so that nothing is hard-coded for width. Make the default width 16 bits.

Also, change the declaration of `ix` to a `reg` declaration. Use a reasonable width, but also think about how you could make the `reg` width the minimum possible that will allow the model above to work correctly with a parameterized `BusIn` width. Simulate your model to verify that it works. Typical simulation results are shown in Figs. 4.14 and 4.15.

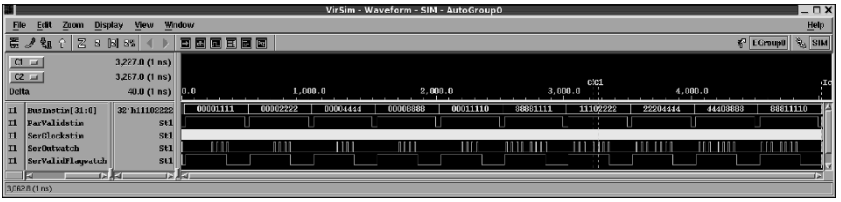


Fig. 4.14 Overview simulation of the generic parallel-to-serial converter

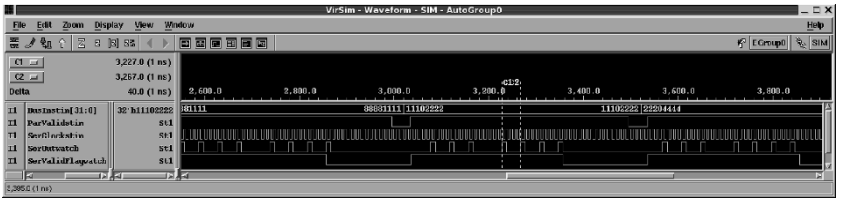


Fig. 4.15 The generic parallel-to-serial converter: Serial data closeup

Step 7. Serialization Frame Encoder. In the final part of this lab, we shall model a parallel-to-parallel encoder which takes a generic parallel bus and converts to a wider bus which includes framing and frame boundary (pad) markers. This last format then clearly can be serialized to clock our class-project serdes PLL as well as to be transmitted serially.

Refer to the block diagram of the Serializer presented at the start of this chapter. On each positive edge of a do-the-encode input clock, the parallel bus is sampled and copied to its framed format. The input bus may be assumed 32 bits wide and the framed output bus 64 bits wide, because we require that after each data byte on the input bus, we insert 8 bits of frame padding. See Fig. 4.16.

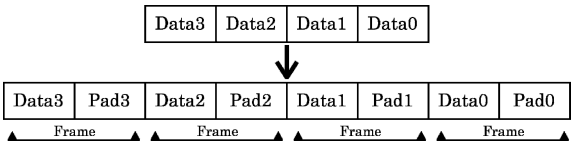


Fig. 4.16 The serdes packet format. Each data byte is framed with one identifying pad byte

Recalling the frame format for our project, to identify each byte of data, each frame boundary (8 bits) contains an ordinal number giving its place in the original 32 bits of data.

It takes 2 bits to enumerate the 4 data (bytes) in a 32-bit bus, so we shall make up each frame boundary to include a 2-bit number padded on each side by 3 binary `0`'s. Our frame boundaries then will be these binary numbers: `Boundary0` (below the lowest-order data byte) = `8'b000_00_000`; `boundary1 = 8'b000_01_000`; `boundary2 = 8'b000_10_000`; and, `boundary3 = 8'b000_11_000`.

So, one sample of our framed data will look like this in binary, with `x`'s representing the original input data:

`64'bxxxxxxxx000`**`11`**`000xxxxxxxx000`**`10`**`000xxxxxxxx000`**`01`**`000xxxxxxxx000`**`00`**`000.`

Design a module named `SerFrameEnc` which will encode an input bus this way on every positive edge of a sampling-clock input. When writing your model, use verilog `parameter` values to specify the input and output bus widths. After simulating it to check the result (see Fig. 4.17), try to synthesize your model, optimizing first for area and then for speed.

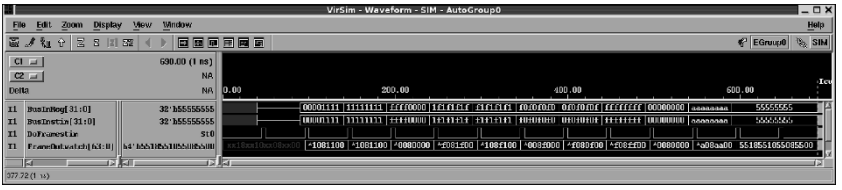


Fig. 4.17 A `SerFrameEnc` simulation, netlist optimized for speed

4.2.1 Lab Postmortem

Where should `'timescale` be set in a multimodule design?

How does verilog distinguish integer from float (`real`) numerical constants?

How could the PLL comparator be modified to compare on every clock input, rather than on every edge? How would this change the constants `'define`'d in the top module, `PLLsim`?

The frame encoder seems very inefficient, using 64 bits to encode 32 bits of data. How might this encoding be made more efficient? At 2 or more Gb/s for a PCI Express serdes, does it matter? We'll look into this question again soon.

4.2.2 Additional Study

SerDes currently (2008) is a hot design topic. Read the overview posted at the Freescale site: `http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01HGpJ2350NbkQ` (2004-11-16).

For a simple introduction to PLL's, with some history, read Ron Bertrand's, "The Basics of PLL Frequency Synthesis", in the *Online Radio and Electronics Course*, at `http://www.radioelectronicschool.com/reading/pll.pdf` (2004-12-09).

(optional) For some background on the analogue issues, try the article by S. Seat, "Gearing Up Serdes for High-Speed Operation", posted at `http://www.commsdesign.com/design_corner/showArticle.jhtml?articleID=16504769` (2004-11-16).

(optional) Two very thorough articles on serdes analogue design considerations, especially those concerning the PLL, are by E. H. Suckow: "Basics of High-Performance SerDes Design: Part I" at `http://www.analogzone.com/iot_0414.pdf`; and, Part II at `http://www.analogzone.com/iot_0428.pdf` (2004-11-16).

(optional) A very good presentation of PCI Express as it relates to system architecture was given by Kevin Edwards at ***EuroDesign Con 2004***: "PCI Express – IP for a Next-Generation I/O Interconnect". This paper is very IP and standards oriented. Available from Mentor Graphics on free registration at: `http://www.techonline.com/community/tech_group/embedded/tech_paper/37455` (2005-06-11).

(optional) The S. Knowlton paper listed in the References covers the PCI Express system architecture and the analogue speed issues but is available only to Synopsys tool licensees. Knowlton explains the individual PCI Express system components in a way very compatible with our class project. He also gives some detail on packet buffering, the retry and ECC functions, and the specific PCIe component called the serdes, which is in the analogue domain of the PCIe standard.

Chapter 5
Week 3 Class 1

5.1 Data Storage and Verilog Arrays

This time we'll study data storage and integrity, and the use of verilog arrays to model memory.

5.1.1 Memory: Hardware and Software Description

Memory retains data or other information for future use. In a hardware device context, memory may be divided into two main categories, random-access and sequential-access.

Random access memory (RAM) is most familiar as the storage in the memory chips used in computers. Any storage location can be addressed the same way, and by the same hardware process, as any other storage location. Every addressed datum is equally far from every other one, so far as access time is concerned. EPROM, DRAM, SRAM, and flash RAM or ROM are familiar names for implementations of this kind of memory.

Sequential-access memory includes tapes, floppy discs or hard discs, and optical discs such as CD or DVD discs. Sequential access requires a process or procedure which varies with the address at which the storage is located. For example, a tape may have to be rewound to reach other data after accessing data near the end of the tape; or, the read/write head of a disc may have to be repositioned and move different distances when it changes from one address to another.

In the present course, we shall not be dealing with sequential-access memory models; they have no existence in the verilog language. However, random-access memories do. Verilog has an ***array*** construct specifically intended to model RAM.

Side issue: The description of RAM chip capacity on data sheets and in the literature varies somewhat and can be confusing. A hardware databook often will describe the capacity by giving the total number of bits addressable (excluding error correcting or parity storage), and will give the word width, thus: (*Storage in bits*) $\times$ (*word*

width). So, a "**256k** $\times$ **16**" RAM chip would store 256k $= 256 \times 1024 = (2^8 \times 2^{10}) = 2^{18}$ bits. To determine how many address locations in this chip, one divides by the word width: Clearly, in this example, there would have to be an 18-bit address bus to address by 1-bit words. This means $(2^{18}/2^3) = 15$ bits of address to address by byte, or 14 address bits to address by the hardware-defined 16-bit word width for this chip.

Hardware manufacturers use this method when these chips are designed for narrow 1-bit or 4-bit words and are intended to be wired in parallel to match the computer word size. For example, 8 **256 k** $\times$ **1** DRAM chips would be required for a memory of 256k bytes. Or, 9 would be required for 256k of memory with parity. No arithmetic is necessary to know the final number of addresses; 32 of these chips would be required for 1 MB (megabyte) of memory.

In Palnitkar (2003), there is a DRAM memory model in Appendix F. The author uses a software description of this memory: It is given as (*storage in words*) $\times$ (*word width*). This memory, interpreted as a single DRAM chip, stores (256 k $\times$ 16) bits $=$ $(256 \times 1024) \times (16) = 2^{(8+10)+4} = 2^{22} = 4$ Mb (megabits), recalling that 1 Mb $=$ $1024 \times 1024 = 2^{20}$ bits. To model a memory accurately, it is necessary to understand what the description is saying.

5.1.2 Verilog Arrays

We have worked with verilog ***vector***s up until now; these objects are declared by a range immediately following the (predefined verilog) type: For example, `reg[15:0]` is used to declare a 16-bit vector for storage.

A verilog ***array*** also is defined by a range expression; however, the array range *follows the name* being declared. Historically, the array range is kept separate from the vector range precisely because it was intended that an array of vector objects should be used as a memory. For example, "`reg[15:0] WideMemory [1023:0];`" declares a memory named `WideMemory` which has an address range (array) totalling 1024 locations; each such location stores a 16-bit `reg` object (word).

Notice that the upper array index is not a bit position; it is the number of storage locations in the memory minus 1 if the lower index is 0. The address bus for `WideMemory` would be 10 bits wide.

The general syntax to declare a verilog array thus is:

`reg [`***vector*** *log indices*`]` *Memory_Name*`[`***array*** *location indices*`];`

A signed `reg` type, such as `integer`, also might be used, but this would be rare.

An example of memory addressing is,

```
reg[7:0] Memory[HiAddr:0]; // HiAddr is a parameter >= 22.
reg[7:0] ByteRegister;
reg[15:0] WordRegister;   // This vector is 16 bits wide.
...
ByteRegister       <= Memory[12]; // Entire memory word = 1 byte.
WordRegister       <= Memory[20]; // Low-order byte from the memory word.
WordRegister[15:8] <= Memory[22]; // High-order byte from the memory word.
...
```

Like hardware RAM, verilog memory historically was limited in the resolution of its addressability. A CPU can only address one word at a time, and when it does, it gets the whole word, not just a single bit or a part of the stored word. It used to be so in verilog: A memory datum (= array object) could not be accessed by part or bit, unless the words it stored were just one bit wide. This is not true any more after *verilog-2001*.

For example, suppose a memory word size was 64 bits, but the system word width was 32 bits. Then, the following code would be legal in *verilog-2001* and *verilog-2005*:

```
reg[63:0] Memory[HiAddr:0]; // HiAddr is a parameter > 56.
reg[7:0]  ByteRegister;
reg[31:0] WordRegister;    // This vector is 32 bits wide.
...
ByteRegister      <= Memory[57];       // Entire memory word (truncated).
ByteRegister      <= Memory[50][15:8]; // 2nd byte from a memory word.
Memory[56][63:32] <= WordRegister;    // To the high half of memory word 56.
...
```

To declare a mamory, vector and array sizes are given with ranges separated, but the resulting objects are referenced with ranges *all following* the object name. The memory address is immediately next to the declared name and references an entire array of bits of some kind; selects follow to the right of the memory location, as in selecting from a vector.

Verilog (*verilog-2001*) allows multidimensional arrays. For example,

```
reg[7:0] MemByByte[3:0][1023:0];
```

declares an object interpretable as a memory storing 1024 32-bit objects, each such object being addressable as any of 4 bytes. Or, it might be interpreted as storing 4096 8-bit objects arranged the same way. So, "`ByteReg <= MemByByte[3][121];`" may be used as though reading the high byte stored at location `121`. The ("`[3][121]`") is an address, *not a select*, so the declared order of the indices is used. Also, because these are addresses, variables are allowed in the address index expressions. Variables are not allowed in part-selects anywhere in verilog; they are allowed in vector bit-selects.

In a multidimensional array, any number of dimensions is allowed, but only rarely would more then three be useful. It is possible to reduce the required dimensionality by one by using a part-select on the addressed word; of course, such a part-select would be legal only if in constant indices (literals, parameters, or constant expressions entirely of them).

For example,

```
reg[7:0] Buf8;
reg[7:0] MemByByte[3:0][1023:0]; // 2-D (call byte 3 the high-order byte).
reg[31:0] MemByWord[1023:0];     // 1-D.
integer i, j;
...
i = 3;
Buf8 <= MemByByte[i][j]; // High-order byte (3,j) stored in Buf8.
Buf8 <= MemByWord[j];    // Low-order byte stored.
Buf8 <= MemByWord[j][31:24]; // Part-select; high-order byte stored.
Buf8 <= MemByWord[j][(i*8)-1:(i-1)*8]; // ILLEGAL! i is a variable!.
...
```

Thomas and Moorby (2002) discusses multidimensional arrays in appendix E.2.

Finally, it is not legal to access more than one memory storage location in a single expression: For `reg[7:0] Memory[255:0]`, the reference, "`HugeRegister <= Memory[57:56];`" is not legal, nor, for "`reg[31:0] MemByByte [1023:0];`", would be "`MyIllegalByte <= MemByByte[121:122] [31:28];`", which would seem to cross address boundaries to get 4 bits from each of two different addresses. *Only one address per read or write is allowed*; but, like a bit-select of a plain vector, an address may be given by a variable.

This last implies that an array object never may be assigned directly; it has to be accessed, possibly in a loop, one address at a time.

Verilog memory access currently is associated with these limitations:

- Only one array location is addressable at a time.
- Part-select and bit-select by constant are legal after *verilog-2001*, but implementation by tools is spotty.
- Part-select or bit-select by variable is not allowed.
- Neither *VCS* nor *Silos* (demo version) can display a memory storage waveform; however, *QuestaSim* and *Aldec* can.

Thus, currently, it is best to access memory data by addressing a memory location and assigning the value to a vector; this vector value then can be displayed as a waveform and may be subjected to constant part-select or variable bit-select as desired. This approach is portable among simulators and synthesizers. For example,

```
parameter HiBit = 31;
reg[HiBit:0] temp;            // The vector.
reg[HiBit:0] Storage[1023:0]; // The memory.
reg[3:0] BitNo;   // Assigned elsewhere.
...
temp = Storage[Addr];
HiPart = temp[HiBit:(HiBit+1)/2]; // A parameter is a constant.
LoPart = temp[((HiBit+1)/2)-1:0];
HiBit  = temp[BitNo];         // Bit-select by variable is allowed.
...
```

5.1.3 A Simple RAM Model

All that is necessary is a verilog memory for storage, an address, and control over read and write. For example,

```
module RAM (output[7:0] Obus
           , input[7:0] Ibus
           , input[3:0] Adr, input Clk, Read
           );
reg[7:0] Storage[15:0];
reg[7:0] ObusReg;
//
assign #1 Obus = ObusReg;
//
always@(posedge Clk)
if (Read==1'b0)
     Storage[Adr] <= Ibus;
else ObusReg      <= Storage[Adr];
endmodule
```

5.1.4 Verilog Concatenation

At this point, it may be useful to introduce one verilog construct we have not yet discussed: ***Concatenation***. To concatenate one or more bits onto an existing vector, the concatenation operator, "{...}" may be used in lieu of declaring explicitly a wider vector and assigning to it by part select or bit select. All this does is save declarations of temporary data; for permanent storage, the concatenated result would have to be copied to a wide-enough vector somewhere.

For example, to concatenate a parity bit to the MSB end of a 9-bit data storage location,

```
reg[7:0] DataByte;       // The 8 bit datum, without parity.
reg[8:0] StoredDataByte; // High bit will be 9th (parity) bit.
...
StoredDataByte <= {^DataByte, DataByte}; // A 9-bit expression.
```

Likewise, two bytes stored in variables could be concatenated by `Word <= {HiByte, LoByte};`.

5.1.5 Memory Data Integrity

This topic is a huge and complex one and is an active research subject. We shall code no more than parity checking in this course, but we shall introduce the principles behind error-checking and correction (ECC).

The problem is that hardware can fail at random because of intrinsic defects, or because of outside influences such as RF or nuclear radiation. We cannot address these failures in terms of verilog, but some background may be found in the supplementary readings referenced at the beginning of this book.

Error checking usually is done by computing parity for each storage location, by calculating various checksums, or by more elaborate schemes involving encoded parameters sensitive to the specific values of the data bits: If a bit changes from the value originally stored for it, a check is supposed to detect this and warn the user or initiate a correction process. The basic principle is to store the check parameter somehow so that hardware failures will change only the check or the data, but not both in a way masking the failure.

Parity checking is commonplace for RAM: The number of binary '`1`' values (or, maybe '`0`' values) is counted at each address, and an extra bit is allocated to each address which is set either to '`1`' or '`0`' depending on the count.

The parity bit makes the total number of '`1`' (or '`0`') values always an even number (for "even parity") or always odd ("odd parity"). It usually doesn't matter whether even or odd is used, or whether '`1`' or '`0`' is counted, so we shall from here on speak only of *even parity on '1'*. In this scheme, an even number of `1`'s makes the sum, the parity bit value, even – in other words, a `0`. So, a byte now takes 9 bits (8 data + 1 parity) of storage, but any change in a bit always is detected; changes in 2 bits in the same 9-bit byte will be missed.

A sum is just an `xor` if we ignore the carry, so parity may be computed by the verilog *xor* (`^`) operator. For example,

```
reg[HiBit:0] DataVector;
reg[HiBit+1:0] DataWithParity;
...
// Compute and store parity value:
DataWithParity = {^DataVector, DataVector};
// Check parity on read:
DataVector = (^DataWithParity==1'b0)
             ? DataWithParity // Parity bit is discarded.
             : 'b0; // Assign zero on parity error.
```

Parity checking is adopted because it is fast; it incurs no speed cost when done in hardware; and, it is easy, because a simple `xor` reduction (verilog ^) of any set of bits yields the binary sum automatically.

Computing parity on every access to any memory location thus is modelled easily.

Checksums usually are used with large data objects such as framed serial data or files stored on disc. The checksum often is just the sum of all '1' values (or sometimes bytes) in that object; but, unlike parity, it may be a full sum, not just a binary bit. Any change in the data which changes the sum indicates an error. If bits are summed, a change from ASCII 'a' to 'b' in a word will flag an error; if bytes are summed, a missing ASCII character in a file will flag an error.

Unlike parity values, checksums generally are stored in a conceptually separate location from the data they check. They may be used for simple error detection or for *E*rror *C*hecking and *C*orrecting (ECC) code.

The check*sum* may be calculated any of a variety of ways:

- As a sum of bytes, frames, or packets.
- As a sum of bits.
- As a sum of '1' or '0' bits (= parity, if no carry).
- As an encoded sum of some kind. For example, as a CRC (Cyclic Redundancy Check). A Linear Feedback Shift Register (LFSR) is one way to implement CRC in hardware.

To reduce the likelihood of missing a change in, say, two bits or bytes between checks, elaborate partial encodings of stored data are used. For example, for serially transferred data, a linear feedback shift register (LFSR) can be used to compute a checksum-like representative number for each such object. The LFSR computes a running *xor* on the current contents of certain of its bits with a value fed back from a subsequent bit. See this idea in Fig. 5.1.

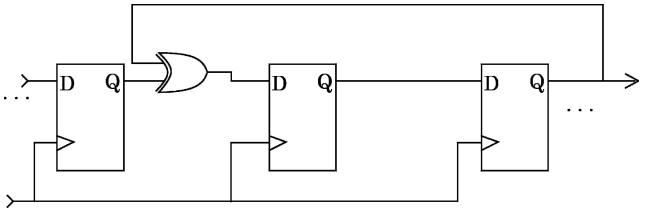


Fig. 5.1 Three stages of a generic LFSR, showing xor of fed-back data

Every time the object is accessed, it is shifted through this register, and the result may be compared against a saved value. The shift incurs a latency but no other delay.

Thomas and Moorby (2002) discuss CRC rationale in greater detail. An example they give is represented schematically in Fig. 5.2.

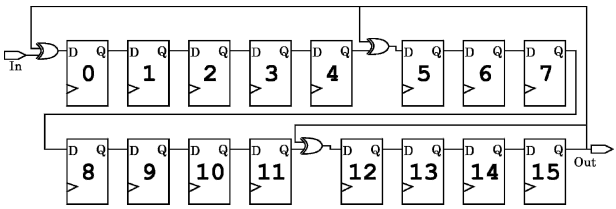


Fig. 5.2 LFSR characteristic polynomial in hardware. The value of `Q[15:0]` represents the CRC defined in Thomas and Moorby (2002) section 11.2.5 as $x^{16}+x^{12}+x^5+1$. Modulo division of valid stored data by this polynomial will return 0. The common clock to all bits, and the output taps, are omitted for clarity

5.1.6 Error Checking and Correcting (ECC)

ECC not only checks for errors, but it corrects them, within limits. All ECC processes can find and correct a single error in a data object, such as a memory storage location. Some can detect two or more errors and can fix more than one. All corrections are made by the hardware and generally incur little or no time cost. Almost all commercial computer RAM chips include builtin ECC functions. Hard discs and optical discs always include ECC.

Basically, the ECC idea depends on a checksum: A representation of the data is stored separate from it; the data are compared with this representation whenever the data are accessed; an error is localized to the bit which was changed, and the data actually seen by the computer or other device are changed to the original, correct value.

Some of the *Additional Study* reading will explain the details, but a brief, conceptual introduction will be presented now: We shall show how to do ECC using parity:

ECC from parity. Consider a parity bit pT representing the total, 8 bits of data, and *suppose just one bit could go bad*. Assume a specific parity criterion, such as even-1. If a parity check failed, all that could be done would be to recognize that this specific data object (8+1 bits) was bad; no correction could be made except perhaps to avoid using the data.

But, suppose two parity bits had been computed, one ($p1$) for the low nybble (bits 0–3) and the other (pT) for $p1$ and the other 8 bits. Then, if pT failed a check, but $p1$ didn't, the apparatus could proceed on the assumption that the error was localized in bits 4–7 or in the pT bit. If both $p1$ and pT failed, an error must have occurred in bits 0–3, or $p1$, but not pT, If $p1$ failed but not pT, one could be sure at least two errors had occurred, which we have agreed not to consider in this example.

With three parity bits, *p*1 and *p*T as before, and a new *p*E calculated from all even bits (0, 2, 4, 6), one could narrow down the error to two of the four bits in one nybble, and with fourth parity bit *p*L on the low half of each nybble, the erroneous bit could be identified unambiguously. The correction then would be just to flip the current value of that bit and go on using the resultant, corrected datum just as though there had been no error. Cost: 12 bits to represent 8 bits of data; in this simple case, a 50% overhead in size, but no cost in speed.

The process just described may be summarized this way for one byte:

Assume only one hardware failure per word, and **`reg[7:0] Word;`**

1. Define **`pT = ^Word[7:0];`**
   `pT` toggles if any bit in **`Word`** changes; system can detect this. 8'bxxxxxxxx
2. Define low-nybble **`pN = ^Word[3:0];`**
   `pN` and **`pT`** toggle if any bit in low nybble changes.
   `pT` toggles if any bit in **`Word[7:4]`** changes.
   Thus, system can determine which half of **`Word`** is reliable. 8'bxxxxxxxx
3. Define even **`pE = ^{Word[6],Word[4],Word[2],Word[0]};`**
   System can determine whether odd or even bits,
   of which half, are reliable. `8'bxxxxxxxx`
4. Define low half-nybble **`pL = ^{Word[5:4], Word[1:0]};`**
   Using **`pT`**, **`pN`**, **`pE`**, and **`pL`**, system can determine which bit changed and flip it back during a read. `8'bxxxxxxxx` → `8'bxxxxxxxx`
   This ECC costs 4 extra bits per 8 data bits.

Realistic ECC. Usually, data objects larger than one byte are adopted for ECC; and, for them, the size overhead can be a smaller percentage of the data to be checked. Instead of a binary search in a parity tree to recover single errors, it is statistically more efficient to use a finite element approach in which parity is replaced by an overdetermining set of pattern coefficients. This is done almost always by using LFSR hardware and applying algebraic field theory to encode regularities of the stored data in the checksums. Multiple bit errors in a large block of data can be recovered somewhat independently of where the errors occur in the block, and the checksum overhead for practical ECC of a 512-byte block of data can be less than 64 bytes. This overhead is about equal to that of simple, byte-wise parity checking without correction!

To illustrate the mechanics of a realistic ECC process, suppose we sidestep the complications of group theory and adopt a minimally complex method, again based on parity: We shall take an 8-bit byte of data and append to it a checksum composed of a vector of 8 parity bits composed as follows:

> The 8 data bits will be written MSB on the left. Concatenated to the right of the data LSB will be a 8-bit checksum. The leftmost checksum bit will be calculated as the parity (even parity on '1') of the whole data byte; the next checksum bit will be calculated as parity of the 7 bits of data with the MSB excluded. The next checksum bit will be parity of the least significant 6 data bits, and so on down to the 8th checksum bit, which will be equal to the LSB of the data.

A simple hardware implementation of this method could be a LFSR with one storage element fed back on itself as shown in Fig. 5.3.

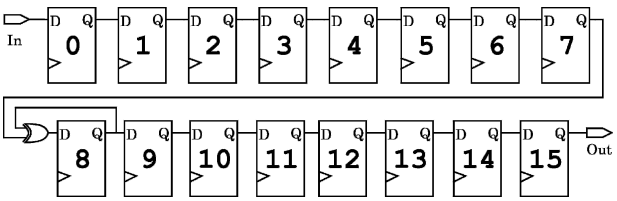


Fig. 5.3 A minimally complicated LFSR permitting ECC

To operate this LFSR, one initializes it with `16'b0` and shifts in the data LSB first, twice (16 shifts). The result will be the desired pattern of `xor`'s in the rightmost 8 bits, and a copy of the data in the leftmost 8 bits. The result could be transmitted serially with just a latency penalty; it also could be offloaded onto a parallel bus for direct memory storage.

To see how the ECC might work, suppose the data byte was `1010_1011`; checksummed, this word would become, `1010_1011_1001_1001`.

Now, suppose that a 1-bit error occurred during serial transmission; for example, suppose the data LSB flipped, making the received word, `1010_1010_1001_1001`.

The *Rx* would calculate the checksum of the received word to be `0110_0110`, clearly a gross mismatch to the received checksum. It would be unreasonable to consider the possibility that the checksum could contain an error of so many bits, making it so distant from the one calculated from the received data. Avoiding a closed-form solution in this example, the *Rx* could formulate 8 hypotheses to correct the data by calculating all possible checksums with 1 data bit flipped in each:

The 8 possible 1-bit corrections to a received word of `1010_1010_1001_1001`:

Hypothesis	Corrected Data	Computed Checksum
h0	`0010_1010`	`1110_0110`
h1	`1110_1010`	`1010_0110`
h2	`1000_1010`	`1000_0110`
h3	`1011_1010`	`1001_0110`
h4	`1010_0010`	`1001_1110`
h5	`1010_1110`	`1001_1010`
h6	`1010_1000`	`1001_1000`
h7	`1010_1011`	`1001_1001`

Hypothesis h7 generates the received checksum; so, our ECC should flip the data LSB to correct it.

Now let us assume two errors, say the data MSB and the data LSB. Again, we do not allow that the received checksum could contain so many errors.

The 8 possible 1-bit corrections to a received word of 0010_1010_1001_1001:

Hypothesis	Corrected Data	Computed Checksum
h0'	1010_1010	0110_0110
h1'	0110_1010	0010_0110
h2'	0000_1010	0000_0110
h3'	0011_1010	0001_0110
h4'	0010_0010	0001_1110
h5'	0010_1110	0001_1010
h6'	0010_1000	0001_1000
h7'	0010_1011	0001_1001

In this case, no 1-bit correction to the data yields the received checksum; however, h7' yields a checksum very close (only 1 bit away). It would be reasonable to accept h7', flip the LSB, and then try 8 more hypotheses for a second correction; this would result in a 2-bit ECC which would correct both data errors. In actual practice, the distances are quantified and minimized in closed form in the algebra of Galois fields, but this simple example shows the basic properties of a checksum permitting multibit ECC.

For more information on the algorithms and computational details of ECC checksum encoding, see the Cipra and the Wallace articles in the References.

5.1.7 Parity for SerDes Frame Boundaries

A simple parity value might be used to improve greatly the efficiency of our planned serdes serial data framing. However, we shall not use it in this course. We are interested in design in verilog, and our inefficient but obvious 64-bit packet makes it both easy and instructive to recognize verilog design errors during simulation. We do not wish to obscure a possible design error to ensure hardware we never intend to manufacture.

However, let's ignore our own project once more, for the moment. Consider the following way of determining clock synchronization of a local PLL clock with the embedded clock in the serial data. Instead of padding the data with 8 bits of encoded order information per byte, as we shall do in our project, suppose we added just a parity bit to each datum, extending it to 9 bits per byte. Then, a packet of 32 bits of our serialized data will look something like this:

```
36'bXxxxxxxxPXxxxxxxxPXxxxxxxxPXxxxxxxxP
```

The parity for each byte follows that byte, in the sense that we are assuming that the MSB is sent first over the serial line. Each byte's MSB is represented by an

upper-case X, and the parity by P. Compare this with the Step 8 representation in our previous lab. With underscores to emphasize byte boundaries, we may write,

```
36'bXxxxxxxxP_XxxxxxxxP_XxxxxxxxP_XxxxxxxxP
```

Now, suppose we try to synchronize a PLL clock with a stream of such frames: If we know we are on a byte boundary and are worried about a 1-bit jitter, we can calculate parity: If we should shift by a bit, the parity *might* change, and *maybe* we could adjust our PLL to resynchronize. If we are using even-1 parity, a '1' in the wrong frame will trigger a parity error, but a '0' won't. Detection of a framing error then would be about 50% accurate.

But, this isn't good enough: We want reliable synchronization. So, assuming even-1 parity, let's guarantee that a parity error will occur on a framing error, at least in one direction. We simply add another new bit, a trigger bit, which always is the inverse of one of the bits in the frame, to the end of every frame.

Because even-1 parity always implies that the *xor* of the parity bit and its word must be 0, the receiver's (*Rx*) parity hardware will verify that the parity of the expression, (*word* xor *parity bit*) always is 0. So, if the MSB of the word is toggled, this must toggle the parity bit, or a parity error will occur.

So, let's add our new trigger bit but position it in the frame at position (MSB-9) and require that it always will be ignored in the transmitter's (*Tx*) parity calculation. Our data packet now consists of 10 bits to represent every 8 bits of data, with parity bit *P* following the LSB, and an extra bit, our trigger bit, indicated by underlined lower-case *x*, following the parity bit and set to the inverse of the MSB of the preceding data byte. The trigger bit never is counted in the *Tx*'s calculation of the value of *P*:

```
40'bxXxxxxxxxPxXxxxxxxxPxXxxxxxxxPxXxxxxxxxP.
```

The MSB of each byte is capitalized, X. The *Tx*'s parity bit is bit P; each P ends a 10-bit data frame. Each *x* is a trigger bit; the *x* following the first P from the left, for example, is set to the inverse of the first X from the left.

Now we have achieved some progress: The correct framing would be the following, with underscores to indicate the receiver's (*Rx*) detected frame boundaries. The first bit in each frame is the *x* inverted MSB X value and is ignored for *Rx* parity:

```
40'bxXxxxxxxxPxXxxxxxxxPxXxxxxxxxPxXxxxxxxxP.
```

A 1-bit *Rx* framing error lagging would be this,

```
40'bxXxxxxxxxPxXxxxxxxxPxXxxxxxxxPxXxxxxxxxP.
```

and, leading, it would be this,

```
40'bxXxxxxxxx_PxXxxxxxxx_PxXxxxxxxx_PxXxxxxxxx_P.
```

The lagging error clearly causes the *Rx* to ignore the real (*Tx*) MSB and replace it by its inverse; this forces incorrect parity and guarantees an error which always will

be detected. The leading error can be detected reliably some of the time, whenever the transmitted-data LSB happens not to equal the parity value of the now-garbled data (which has x in the place of its MSB). The resultant raw framing-error detection rate then should be expected to be about 75%.

A PLL biased very slightly to lag exact synchronization thus can be designed to achieve a very low rate of undetected 1-bit framing errors. The approach above would allow a designer to adjust a receiving PLL reliably with data in a frame no larger than 10 bits per byte. The ratio of 10 bits per byte is the assumption usually made in actual PCI Express designs, a numerical coincidence because we have ignored many serialization complexities, such as Manchester encoding.

5.2 Memory Lab 7

Lab Procedure
Work in your `Lab07` directory.

Step 1. Try the following memory access statements. Initialize the RHS variables with literal constants, and then see which ones work:

```
reg[63:0] WordReg;
reg[07:0] ByteReg;
reg[15:0] DByteReg;
reg[63:0] BigMem[255:0];
reg[3:0]  LilMem[255:0];
...
BigMem[31]      <= WordReg;
WordReg         <= BigMem;
LilMem[127:126] <= ByteReg;
LilMem          <= ByteReg[3:0];
DByteReg        <= ByteReg;
ByteReg         <= DByteReg + BigMem[31];
WordReg[12:0]   <= BigMem[12:0][0];
```

Step 2. Design a verilog $1\text{k} \times 32$ static RAM model (32×32 bits) with parity. Call the module, "`Mem1k x 32`". Check this model by simulation as you do your work on it.

This RAM will require a 5-bit *address bus* input; however, use verilog `parameters` for address size and total addressable storage, so that quickly, by changing one parameter value, you could modify your design to have a working model with more or fewer words of storage. Parity bits are not addressable and are not visible outside the chip.

You may model your RAM after the Simple RAM Model given preceding presentation. Use just one `always` block for read and write; but, of course, the module will have to be considerably more complicated than the Simple RAM.

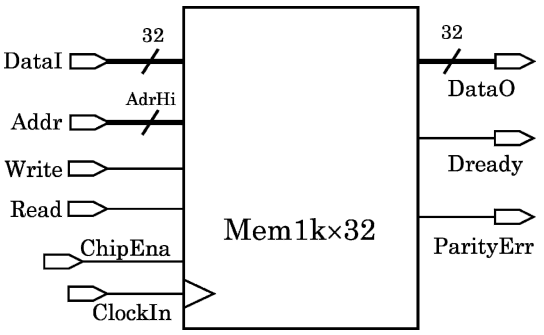


Fig. 5.4 The `Mem1k x 32` RAM schematic

Use two 32-bit data ports, one for *read* and the other for *write*. Supply a *clock*; also an asynchronous *chip enable* which causes all data outputs (read port) to go to 'z' when it is not asserted, but which has no effect on stored data. The clock has no effect while chip enable is not asserted.

Supply two direction-control inputs, one for *read* and the other for *write*. Changes on read or write have no effect until a positive edge of the clock occurs. If neither read nor write is asserted, the previously read values continue to drive the read port; if both are asserted, a read takes place but data may not be valid.

Assign reasonable time delays, using delayed continuous assignments to the outputs. Supply a *data ready* output pin to be used by external devices requiring assurance that a read is taking place, and that data which is read out is stable and valid. Don't worry about the case in which a read is asserted continuously and the address changes about the same time as the clock: Assume that a system using your RAM will supply address changes consistent with its specifications.

Also supply a *parity error* output which goes high when a parity error has been detected during a read and remains high until an input address is read again. Refer to the block diagram in Fig. 5.4.

Because this is a static RAM, of course omit DRAM features such as ras, cas and refresh. Design for flip-flops and not latches. Put the model in a file named after the module.

Include an *assertion* to announce parity violations to the simulator screen. Of course, your simulation model can't possibly experience a hardware failure, but this message may tell you if you make a design error with the parity bit. You can force

an error by putting a temporary blocking assignment in your model to confuse the *xor* producing the parity value.

Step 3. Check your RAM. Write data to an address and simulate to verify that it is read correctly (see Fig. 5.5).

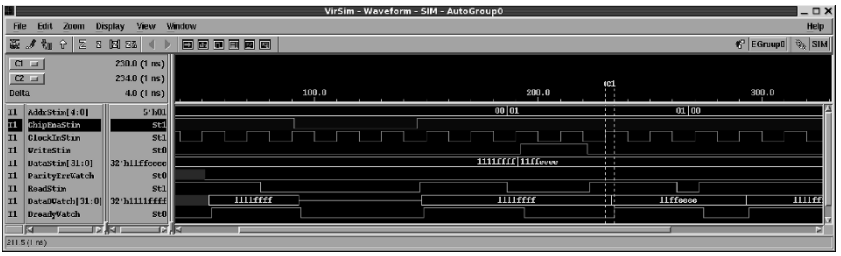


Fig. 5.5 Cursory simulation of single-port `Mem1kx32` with separate read and write ports

Step 4. After completing the previous step and doing any simulation necessary to verify your design superficially, add a `for` loop in a testbench `initial` block to write a data pattern (`e. g.`, an up-count by 3) into the memory at every address and then to display the stored value. Use `$display()` for your display. Pay special attention to the "corner cases" at address 0 and address 31. Your parity bit would be in bit 32 at each address. An example of the loop is given just below. Notice how to address a data object ("`MemStorage`") in an instance, here named `Mem1kx32_inst`, in the current testbench module:

```
for (...)
  begin
  ...
  #1  DbusIn  = (some data depending on loop);
  #1  Write   = 1'b1;
  #10 Write   = 1'b0;
     SomeReg  = Mem1kx32_inst.MemStorage[j];
     $display(``...'', $time, addr, SomeReg[31:0], SomeReg[32]);
end
```

Step 5. Modify your RAM design so it has just one bidirectional data port, Do this by copying your working model (above) into a new file named "`Mem1kx32Bidir.v`". Then, declare a new, empty module in this file, named after the file. Use the exact same new module ports as in the old `Mem1kx32` model, except for only one `inout` data port. See Fig. 5.6.

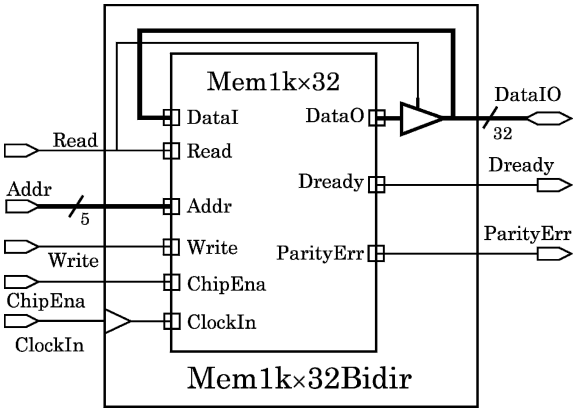


Fig. 5.6 Schematic of wrapper to provide `Mem1kx32` with a bidirectional data bus

Instantiate your old RAM in the new `Mem1kx32Bidir`. Connect everything 1-to-1, but leave the data unconnected.

All you have to do now to complete the connection is to add a continuous assignment in the wrapper module which turns off the `DataO` driver when `Read` is not asserted. Also, wire `DataI` to the new `DataIO` driver as shown above. Verify your new RAM by a simulation that does write and then read from two successive addresses, then reads again from the first address (see Fig. 5.7).

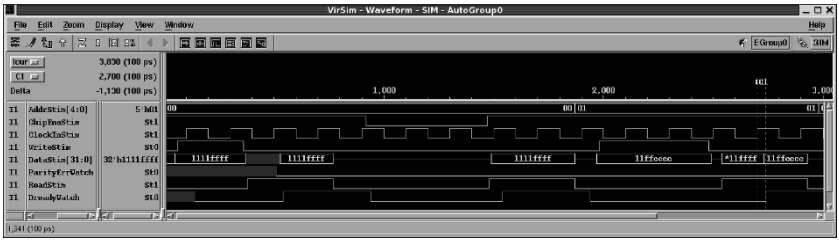


Fig. 5.7 Cursory simulation of single-port `Mem1kx32Bidir` with bidirectional read-write port

If this were part of a much larger design project, you would separate the bidirectional and original RAM modules into different files. However, this exercise is simpler if you allow yourself to keep as many as three modules in one file: The original RAM model, the new bidirectional "wrapper" for that RAM model, and the testbench.

Step 6. Synthesize your bidirectional-data bus memory design, optimize for area, and examine the resulting netlist in a text editor. Resynthesize for speed and examine the netlist again.

5.2.1 Lab Postmortem

Concatenation: When can it be useful?

How are hierarchical references made to module instances?

What's the benefit of a bidirectional data bus?

How might the spec for the `Dready` flag be improved? What about when a read remains asserted while the address changes? Shouldn't the RAM be responsible for all periods during which its output can't be predicted?

How would one change the RAM flip-flops to latches?

Do we really need a `ChipEna`? Why not disable outputs except when a read was asserted?

5.2.2 Additional Study

Read Thomas and Moorby (2002), Section 5.1, on verilog rules for connection to ports.

Read Thomas and Moorby (2002) section 6.2.4, pp. 166 ff. to see how our parity approach can be adapted easily to a Hamming Code ECC format (at 12 bits per byte).

Read Thomas and Moorby (2002) appendix E.1 and E.2 on vectors, arrays, and multidimensional arrays.

(optional) Read "The Laws of Cryptography: The Hamming Code for Error Correction", by Neal R. Wagner, a 2002 web site posting at `http://www.cs.utsa.edu/~wagner/laws/hamming.html` (2004-12-15). This is a brief and very nice treatment of ECC, extending and improving the present coverage. Unhappily, the posting is flagged by the author as "obsolete"; it will become part of a book which is downloadable from the web site but which mostly is irrelevant to this course.

Optional Readings in Palnitkar (2003)

Section 4.2.3 discusses the verilog rules for connection to ports.

Look through the verilog of the behavioral DRAM memory model in Appendix F (pp. 434 ff.). It uses several verilog language features we haven't yet mentioned, so you may wish to put it aside for a few weeks. It may not work with the Silos simulator on the CD.

Chapter 6
Week 3 Class 2

6.1 Counter Types and Structures

6.1.1 Introduction to Counters

A ***counter*** is a data object which represents a value that is incremented or decremented in uniform steps. So, a counter may contain successive values of 0, 1, 2 …; or, 2, 4, 6, …, etc. A 1-bit counter alternates between '0' and '1'. A *n*-bit register for a binary, unsigned up-counter would count as shown in Fig. 6.1, in successive count values starting from 0 on the top row (first register state):

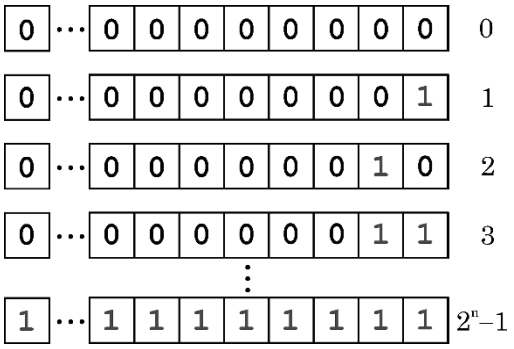


Fig. 6.1 Binary up-count register with MSB on left and successive count values Between 1 and *n* bits switch on each count

The above storage of the count value is very compact spatially, *n* bits for 2^n different values, but every carry to a new bit causes simultaneous switching of several, often very many, bits at a time. This carrying and bit-switching can cause settling delays to vary from clock to clock; all the switching may generate on-chip cross-talk noise or power supply glitches, causing errors in the counter or in nearby devices.

Values of, say 1, 2, 4, 8, 16, ... don't represent a count as such. However, if the value of a counter is interpreted in terms of bit position, a register content of $2^0, 2^1, 2^2, 2^3, \ldots 2^n$, may be considered a count, in that the position of the '1' in a register holding those successive values increments in uniform steps, by one location at a time. Essentially, the $\log_2$ of the value in the register is being counted. An example of this kind of counter is the ***one-hot*** counter, because exactly one bit position always is "hot" with a '1'.

The hardware implementation of a one-hot counter is just a shift register which is initialized with a single '1' in it. A one-hot count would look like that in Fig. 6.2, in terms of successive bit-patterns in the counter register:

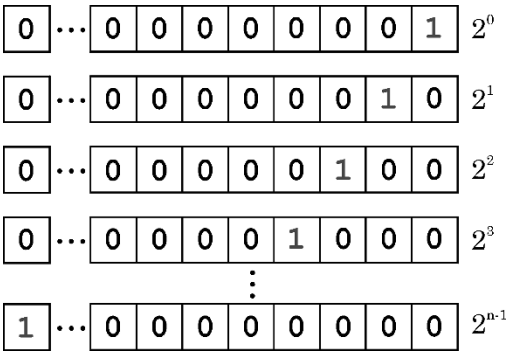


Fig. 6.2 One-hot up-count. Two bits switch on every count

Notice that there can't be any value of 0 represented this way, if the bits in the register are to be interpreted as ordinary, verilog binary (2^n) numbers. Also, the storage capacity is very limited, requiring n bits for just n different values. This kind of count is spatially inefficient, but it is very fast and causes little (but not minimal) noise, because just two bits switch per clock. We trade storage space for time, in a sense.

6.1.2 Terminology: Behavioral, Procedural, RTL, Structural

There are at least three different counter approaches possible in verilog: *behavioral*, *procedural*, and *structural*. Sometimes the word, *RTL*, "*R*egister *T*ransfer *L*evel", is used to describe design activity; RTL overlaps behavioral and procedural.

Behaviorally, the verilog language permits elementary operators to be used for counting, with no concern for the hardware that might result. A simple count may be written as, `Count <= Count + 1`, `Count <= Count - 1`, `Count <= Count + Incr`, etc. Behaviorally, we leave it up to the synthesizer to decide how to put these statements into our netlist. Notice that in the preceding

statements, there is no way of knowing which `Count` bit might change, or which way, because of the statement alone.

Procedural assignments are distinguished by the possibility that a statement might be superceded by another one in simulation time, with no implication that multiple drivers were involved. Procedural assignments only can occur in procedural blocks. The possibility of changing the assigned value is most obvious with blocking assignments. For example,

```
initial // Cycle the reset by time 10:
  begin
  #0 Reset = 1'b0;
  #1 Reset = 1'b1;
  #9 Reset = 1'b0;
  end
```

Behavioral statements may include procedural or continuous assignments. Procedural statements may be bit-specific and thus not behavioral.

Procedurally, we may assign values to counter registers in procedural statements, but the level of detail may be anywhere from that of the whole counter register, as in the behavioral example above, down to that of individual parts or bits. Composing a counter by `xor` expressions and bit-assignments could be procedural, but not behavioral. For example, in an `always` block, `Count[2] <= Count[1]^Carry` would be an example of a bit-level procedural statement. Note that a continuous assignment, `assign Count[2] = Count[1]^Carry`, would make the statement nonprocedural. However, counters are sequential devices; and, whereas a continuous assignment statement might represent an addition, it can not represent a count.

The border between procedural and behavioral coding is not crisp, and often the same code may be described as ***RTL*** (usually a special case of procedural) or ***behavioral***, depending on context. Recall the simple counter for our PLL in *Week 2 Class 2*: That was an RTL counter not quite behavioral because we wrote a bit-width expression for the incrementing literal, `5'h1`.

> The present author feels justified in using the term ***behavioral*** when the code does not explicitly assign identifiable bit values to a port or register; thus, a structural interpretation of the statement is not specified.
>
> This is consistent with usage in the Thomas and Moorby (2002) (section 1.2) and Palnitkar (2003, chapter 7). Very little in this book is purely behavioral, in that it could not be called RTL.

Structurally, verilog allows us complete control over the netlist if we wish to instantiate the gates by hand from a library compatible with our intended technology. In this context, continuous assignment statements are structural connections representing simple wiring or combinational logic. Palnitkar (2003) distinguishes

dataflow modelling as a special style including continuous assignments; however, this terminology is nonstandard. One sure thing is that continuous assignments are not procedural (they can't be included in a procedural block in modern verilog).

Here is another example to clarify the terminology. Suppose the goal is to represent a 2-bit, binary up-count. After a few declarations, we may write verilog as shown next:

behavioral	`Count <= Count + 1;  // Count is reg.`
RTL:	`Count[1:0] <= Count[1:0] + 2'b01;  // Count is reg.`

RTL:	`always@(posedge Clock) Count[0] <= ~Count[0];`
	`always@(Count[0])`
	`  if (Count[0]==1'b0) Count[1] <= ~Count[1];`

structural:	`// Count is net:`
	`DFF Bit0( .Q(Count[0]), .Qn(Wire0), .D(Wire0), .Clk(Clock) );`
	`DFF Bit1( .Q(Count[1]), .Qn(Wire1), .D(Wire1), .Clk(Wire0) );`

Gate-level structural design entry should be avoided when using a synthesizer. Sometimes, when tuning a synthesized netlist, structural control may be necessary; but, for complex structures, the result may not be optimal. Furthermore, the synthesizer's logic optimizer may not be able to improve suboptimal structures which have been built by hand. Finally, gates instantiated from a specific library make the design technology-specific; porting the design to a new technology will be more complicated with hand-instantiation than when the synthesizer is allowed to choose all gates from the new technology library.

6.1.3 Adder Expression vs. Counter Statement

It is important to point out the difference here between an *adder* and a *counter*; An adder performs an expression evaluation, for example, `X <= A + B`. Even if included in a function call, the adder is contained entirely on the right side of this statement. A counter makes a statement of the current value in a register. Thus, addition can appear in a combinational or a sequential procedural statement; counting can occur only in a sequential procedural statement. Although a count increment expression can be implemented as an addition of 1, a count can not be kept in a combinational form; it has to be stored so that the sequence of different values, *count* vs. *next-count*, can be distinguished in order of occurrence. The next *count* must depend on the previous value of *count*.

Counting requires assignment of a sum expression in a statement, which is shown schematically in Fig. 6.3.

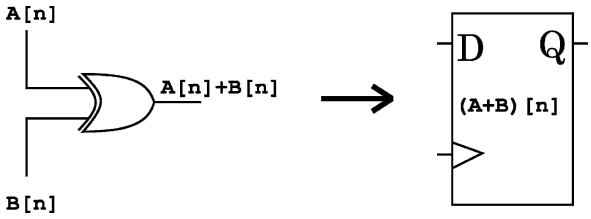


Fig. 6.3 Sequential summing permits counting

6.1.4 Counter Structures

Next, we discuss a few examples of different counter structures. In all the counters to be presented below, assume the count is the binary value taken from the `Q` ports of the flip-flops shown. We shall use only D flip-flops here, because they are the most common kind in current use, either in manual design entry or in synthesized logic.

The element of any binary counter based on D flip-flops is the ***toggle flip-flop***, sometimes called a T flip-flop, which is just a D flip-flop with its ∼`Q` port wired back to its `D` input.

For D flip-flops modelled behaviorally, toggle assignments from `D` to `Q` should be *non*blocking, not blocking, to ensure that updated values are based solely on assignments from the previous clock.

```
//
// Basic toggle flip-flop:
//
wire Qn_D;
...
DFF Toggle01( .Q(Q), .Qn(Qn_D), .D(Qn_D), .Clk(ClkIn) );
...
```

The schematic for this device hookup would be as shown in Fig. 6.4:

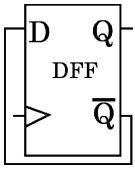


Fig. 6.4 Basic toggle (T) flip-flop

A toggle flip-flop component may be constructed by putting a simple wrapper module around a DFF wired to toggle:

```
module TFF (output Q, input Clock, Reset);
// DFFC is the DFF above, with clear (reset) added.
wire Qn_D;
DFFC Toggler( .Q(Q), .Qn(Qn_D), .D(Qn_D), .Clk(Clock), .Rst(Reset) );
endmodule
```

6.1.4.1 Ripple Counter

This kind of counter is minimal in size and easy to implement structurally. The bits in this kind of counter are flip-flops which are not clocked but are triggered by the data edges (carry out) of previous stages. They are wired to toggle on every active edge on their clock-pin input. The count must be reset to a known value; but, otherwise, no logic is required other than that of the flip-flops themselves. For example, a 3-bit ripple counter could be as simple as in Fig. 6.5.

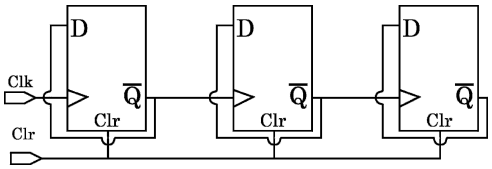


Fig. 6.5 Schematic of 3-bit ripple counter constructed of `D` flip-flops. Outputs not shown. MSB is farthest right. Only the LSB is clocked

Because the higher-significance stages in a ripple counter see no clock and can not change state until all earlier bits have changed state, this is a linearly progressively slower device as the count register widens. A variety of glitch values will appear in the output bit pattern during the time after each clock, while the ripples triggered by the clock settle. However, carry from all stages does not propagate immediately on the clock edge, so the power consumption and noise are reduced when compared with that of a synchronous counter.

6.1.4.2 Carry Look-Ahead (Synchronous) Counter

This kind of counter completes an update of all register bits on each clock input, at the cost of additional carry look-ahead logic. Stated otherwise, all bits always are clocked and would toggle together, all at once, on every clock, except that the toggle for each bit is disabled until all lower-order bits are '`1`'. Thus, although every bit is clocked, no bit toggles to '`1`' until it becomes the carry from the previous bits.

A synchronous counter can operate at almost the same clock speed regardless of register width. However, because all bit switching is synchronous with the clock,

a synchronous counter occasionally briefly will consume power in big spurts, and such a spurt will create noise which might cause logic errors, as mentioned above.

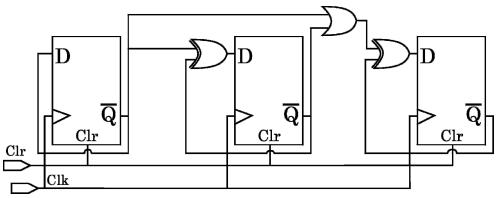


Fig. 6.6 Schematic of a 3-bit synchronous counter. Every bit is clocked. Previous 1's (inverted) are *or*'ed, and an *xor* prevents each toggle until carry overflows into that bit

Notice in the synchronous counter of Fig. 6.6 that each storage element (flip-flop) is driven by a 1-bit adder (*xor* gate). Thus, in a general sense, the flip-flops merely latch, on each clock, the sum composed by the connecting combinational logic.

6.1.4.3 One-Hot and Ring Counters

Unlike the ripple or synchronous counters, these counters do not use all possible bit states of a register.

The one-hot counter was described above.

A ring counter is just a one-hot counter in which the MSB is shifted back into the LSB. In a ring counter, a single '1', or other constant pattern established by reset logic or a parallel load, circulates through the register endlessly. Ring counters may be used to generate clock pulse patterns of arbitrary duty cycle. When used to create a clock, a ring counter in a sense is the inverse of a PLL; it creates a clock of lower frequency than its input, rather than a clock of equal or higher frequency.

6.1.4.4 Gray Code Counter

Of all common counter hardware structures, the gray code is maximally efficient in terms of switching power and noise. It is named after a designer whose name was F. Gray.

In gray-code counting, as in the simple binary up-counter, all possible register patterns are used; but, successive count patterns are such that just one register bit changes on any clock, as shown in Fig. 6.7. This efficiency is at the cost of some extra encoding logic which is necessary to sequence the register patterns properly. Also, determining the equivalent 2^n binary bit count value from the gray code value requires some decoding logic.

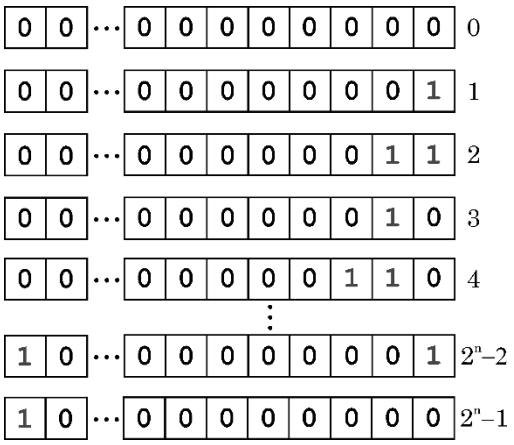


Fig. 6.7 Unsigned gray-code up-count. MSB is on the left. With the bottom pattern (`2^n-1`), one more count restores the `0` pattern

We shall not be using gray code in this course, but it is a good pattern to keep in mind when minimal switching power or noise is a concern.

Another application of gray code is in synchronizing the value of a count across different clock domains; for example, this would be required for generating consistent, successive addresses or state-machine state encodings. As explained in the Cummings papers listed in the References, because only one bit changes at a time, the probability of invalidly sampling an intermediate counter state (by logic in a domain other than that of the counter) is lower for gray encoding than for any other kind of count commonly used.

6.2 Counter Lab 8

Work in the `Lab08` directory for this lab.

Lab Procedure

Note: For a counter clocked on the positive edge, a ***miscount*** in this lab is defined as the wrong number when the count is sampled on the opposite (negative) edge of the clock.

Step 1. Build a ripple counter. You should have a working positive-edge flip-flop (FF) from `Lab04`. If necessary, modify your FF model so it has two outputs, `Q` and `Qn`. The `Qn` logically will be `!Q` (or, ∼`Q`), obtained perhaps by clocking `Qn <= ~D` in the model. If not in your model already, assign a `#3` delay (3 ns) to changes in `Q` and `Qn`.

Then, create a verilog structural model of a 4-bit ripple counter, using your FF component and the design in Fig. 6.5 above. Name the counter module `Ripple4DFF`.

Simulate the model to determine how much simulation time it takes your model to count from `0` to `4'b1100` (hex `4'hc`) at its fastest possible speed. Do this by repeatedly simulating with shorter clock periods until there is a miscount. Record the time for later reference. Use a testbench in a separate file.

Note: The miscounts you find probably will depend only on the delays, and your flip-flops may toggle in simulation no matter what your clock speed. This is because the simulator may not implement *inertial delays* for delayed statements. So, although real hardware would freeze up if clocked too fast, we may have to wait until we study path delays in `specify` blocks before our simulator will reveal this kind of problem.

Synthesize `Ripple4DFF` and try optimization for area and speed. Save the speed-optimized netlist for later in this lab.

Step 2. Build a synchronous counter. Use the design in Fig. 6.6 above to build a structural 4-bit synchronous counter model using the same FF module as in the previous Step. Name the synchronous counter module `Synch4DFF`. Use verilog continuous assignments to represent the connecting `or` and `xor` "gates", and assign a delay of `#2` for each such gate assignment.

Simulate `Synch4DFF` with the same testbench to compare it with the ripple counter in the previous Step; record the shortest possible simulation time to count from `0` to `4'b1100`, as before. Use the same basic testbench file as you did for the ripple counter.

Also, synthesize for area and speed and save the speed-optimized netlist.

Then, instantiate both the ripple-counter speed-optimized netlist and the synchronous-counter netlist modules in a testbench module and see how fast you can run them in simulation. You will have to compile the verilog *Library Name*`_v2001.v` file for this simulation, because you will require verilog models for the components in the synthesized netlists. If you use VCS, add `-v` before the library file name in your `.vcs` file.

Step 3. Write a behavioral counter model. In a new module file named `Counter4.v`, declare a 4-bit `reg` and model a counter behaviorally this way:

```
reg[3:0] CountReg;
...
always@(posedge ClockIn, posedge Clear)
  begin
  if (Clear==1'b1)
       CountReg <= 0;
  else CountReg <= CountReg + 1;
  end
```

Assign a delay to a continuous assignment statement which transfers the `CountReg` value to an output port, `CountOut`. The count delay should match your #3 FF module delay (Step 1 above). Now, simulate and compare times with the two preceding structural counters.

Then, synthesize for area and speed.

Finally, do a little experiment: Make a copy of `Counter4.v` and change the code to `CountReg <= CountReg - 1`. Simulate; and, notice what happens when the count wraps below `0`. A `reg` is an unsigned type, and the value next after `0`, in a down-count, is the maximum positive value expressible in the `reg`.

Step 4. Use a verilog special wire type for logic. Make a copy of your `Synch4DFF` model in a new file named `Synch4DFFWor`. Replace the verilog `or` expressions in your synchronous counter with independent assignments to one or more wired-or (`wor`) nets. Don't assign any delay to the new `or` logic (we'll look into differentiating rise and fall delays later in the course). Simulate again and synthesize for area and speed, to compare with `Synch4DFF`.

Keep in mind that `wor` may not be available in a CMOS technology library, where pull up and pull down strengths usually are equal. When you code a `wor` net, then, the synthesizer either should replace its drivers with special library `wor` gate buffers or other components, or should drive the net with a library `or` gate.

Step 5. Use your PLL as a counter clock. Make a new directory under `Lab08` and copy into it your entire `PLLsim` model from `Lab06`. Rename the `PLLsim` module and its file to "`PLLTop`" to avoid confusion. Instantiate the PLL and all your counters (ripple, synchronous, behavioral) in a single module named `ClockedByPLL` and connect the PLL output clock to them.

Clock your PLL with an input (`ClockIn`) at 1 MHz. See the block diagram in Fig. 6.8. Be sure to remove your timescale and `` `define `` code from all the old module files and put them *only* in the new top of the design (`ClockedByPLL`), and in the testbench instantiating it. Simulate briefly to be sure all counters will count (see Figs. 6.9 and 6.10).

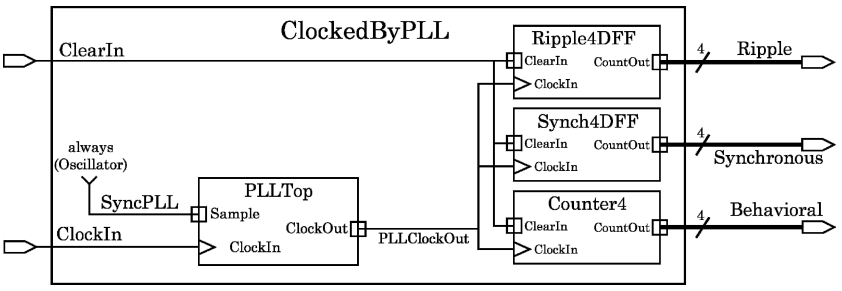


Fig. 6.8 The Lab 6 PLL adapted to clock three counters. Resets omitted

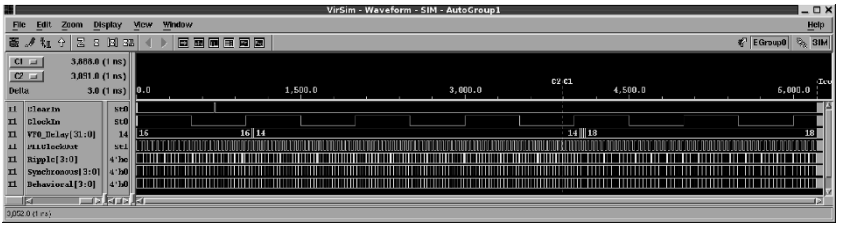


Fig. 6.9 Simulation of the three `ClockedByPLL` counters

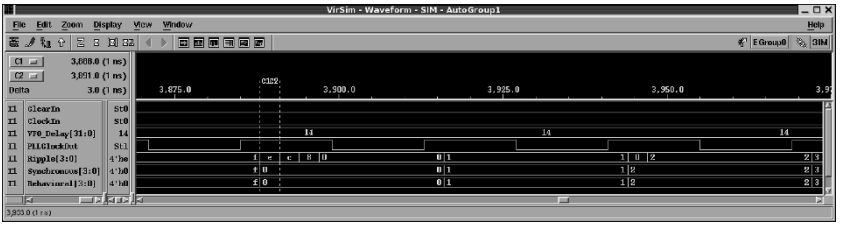


Fig. 6.10 Closeup of the `ClockedByPLL` counters, showing ripple glitches

6.2.1 Lab Postmortem

How do synch vs. ripple counter sim times compare?

If the DFF delay is around 3 ns, why won't the ripple counter count correctly at a clock period of 10 ns?

The intermediate states which are output by a ripple counter before it settles can be confusing during simulation. They also represent brief pulses or glitches which in a real design might be current-amplified by bus drivers, or otherwise might cause unnecessary noise. How might these states be kept off an output bus?

The behavioral counter (`Counter4`) was trivial to change from an up-counter to a down-counter. How easy would it have been to change similarly the ripple counter or the synchronous counter?

6.2.2 Additional Study

Read Thomas and Moorby (2002) section 1.4.1 to see a counter model used in context.

Read Thomas and Moorby (2002) appendix A.14–A.17 to see a counter modelled as a state machine.

Optional Readings in Palnitkar (2003)

Palnitkar models counters at several different places, to make various points about the verilog language. If you are interested in counters *per se*, you may wish to work the examples in section 6.7. Solutions are given on the Palnitkar CD.

The ripple counter is described in sections 2.2 and 6.5.3.

A synchronous counter built from `J-K` flip-flops is described in section 12.7, exercises 5 and 6. Note: A switch-level `J-K` flip-flop is modelled in section 9.4 of Thomas and Moorby (2002).

Chapter 7
Week 4 Class 1

7.1 Contention and Operator Precedence

This time we shall study some new verilog: drive strengths, race conditions, operator precedence, and named blocks. We then shall move on to a discussion on how to lock in our PLL to a serial data stream.

7.1.1 Verilog Net Types and Strengths

Strength means nothing in verilog except when a net has more than one driver.

We've briefly mentioned some of the special net types (`tri`, `wand`, `wor`) in *Week 2 Class 1*. In the present lecture, we merely introduce the different kinds of strength; we shall study switch-level modelling later.

Except the high-impedance state (one of the four basic logic levels in verilog, implied by assignment of '`z`'). there is little use to strength in RTL or behavioral modelling for VLSI design.

According to the verilog standard (IEEE Std 1364, 4.4 and 7.9–7.13), there are two kinds of strength, ***drive strength*** and ***charge strength***.

7.1.1.1 Drive Strength

The drive strength levels, in decreasing order, are called *supply*, *strong*, *pull*, and *weak*. Recall here that '`z`' is considered in the language to be a logic level, not a strength. However, a '`z`' level is effectively an '`x`' asserted at *lower than weak* strength.

Drive strengths of *strong* ('`1`'; '`0`'; '`x`') or weaker-than-*weak* ('`z`') are the only ones associated with the usual functions encountered in RTL or gate models. Drive strength proper in verilog only applies to (a) nets in continuous assignment statements or (b) gate outputs of the builtin verilog primitive logic gates.

Drive strength as such is useless in association with `reg` type declarations; after a `reg` value has been assigned to a net type, strength possibly may become relevant.

7.1.1.2 Charge Strength

The charge strength levels, in decreasing order, are called *large*, *medium*, and *small*; they should be considered as representing the sizes of on-chip capacitors. Switch-level models are used to represent current flow through transistors, which depends in turn on device-element capacitance. The only data objects allowed to be declared with a charge strength are `trireg` nets. They are called `tri`*`reg`*, because they are viewed as charge storage elements, somewhat the way `reg`'s can be viewed as logic-level storage elements.

As said already, except the use of '`z`', strength generally is of no concern in gate-level or behavioral design; strength is intended to be invoked when simulating at the switch level. Switch level models include pass transistors, single nmos or pmos devices, or other gate substructure useful in simulating the elements which are combined to make up a logic gate in a technology library.

Switch-level models exist in something of a vacuum in modern verilog. Accurate validation of models at this level usually requires analogue simulation. Switch-level models can be functionally accurate but not timing-accurate. However, the major concern in verilog modelling of individual technology-library gates is timing, because other factors such as noise, crosstalk coupling strength, and leakage current, are beyond the scope of the language. Because timing usually can be expressed adequately in verilog without switch-level reference, switch-level verilog models are not commonly used in commercial library work in the industry.

Also, library models usually are not written in verilog but in ALF (IEEE Advanced Library Format) or in vendor-specific languages such as *Liberty*. This because a complete library model has to include analogue-related simulation data such as slew rate under load, as well as nonsimulation data such as layout geometry, placement rules, wire-loading, and other quantities beyond the verilog language.

When it occurs, contention is resolved in verilog by strength level. Ambiguous results only occur when different logic levels of '`1`' or '`0`' contend and are of equal strength. In all other contentions, the resultant strength and logic level is that of the higher strength level. Ordinary logic gates, module instances, or verilog expressions act in this context as amplifiers in the sense that they output *strong* logic regardless of the strength of their (resolved) inputs.

Here's an ordered table of verilog strengths (see also Thomas and Moorby (2002), section 10.10.2.2):

Strength Level	Keyword(s)	Logic Level(s)	Strength Number
supply	supply1		7
strong	strong1	1, x	6
pull	pull1		5
large	large		4
weak	weak1		3
medium	medium		2
small	small		1
highz	highz1	z	0
highz	highz0	z	0
small	small		1
medium	medium		2
weak	weak0		3
large	large		4
pull	pull0		5
strong	strong0	0, x	6
supply	supply0		7

The verilog logic *levels* (not *strengths*) normally used in RTL or behavioral code apply only to net assignments and are defaulted to the strength numbers shown. This default can be changed by verilog macroes as explained below.

The strength number in the preceding table is what is used to resolve contention: Higher numbers win over lower ones, and equal numbers at different levels result in unknowns.

Strengths may be declared in parentheses preceding the instance or net name, or may precede the target of a continuous assignment to the net; drive strengths must be in pairs (*high*, *low*), and charge strengths must be single (*size*). Examples are,

```
wire OutWire;
...
and (strong1, weak0) and01( OutWire, in1, in2, in3);
nor (pull1, pull0)   nor31( OutWire, in1, in2, in3);
trireg (large)          LargeCapOnNet;
wire (supply1, supply0) TiedTo1 = 1'b1;
// ---
wire UpClock;
  ...
assign (strong1, weak0) UpClock = ClockIn;
```

In the example code above, a truth table for multiply-driven `OutWire` would be:

in1&in2&in3	and01 **output**	nor31 **output**	Outwire
1	strong1	(pull1 *logically impossible*)	(strong1)
1	strong1	pull0	strong1
0	weak0	pull1	pull1
0	weak0	pull0	pull0

We are interested only in the idea of *strength* at this point. We shall take up switch-level modelling in detail later.

Two relevant procedural assignment statements are ***force*** and ***release***. These are simulator overrides, rather than drive strengths, and they should not be used in a design – only for testbench or debug. The statement, "`force` *object* = *level*`;`", in which *object* is any `reg`, forces the design object to the given logic *level* until "`release` *object*`;`" is executed.

There also is a similar procedural ***assign*** and a ***deassign***; we shall not study these and do not recommend them for design work.

7.1.2 Race Conditions, Again

In verilog, every block in a module has its inputs (or right-hand side – RHS) evaluated concurrently in simulation time; this includes instantiations and `initial` blocks. Ordering of simulation events can be made meaningful (a) by cumulative delays from sequences of events which happen to combine to cause events to occur at unique simulation times; or, (b) by location of statements within procedural blocks (`always` or `initial` blocks).

There is an important ordering of evaluations within each time step, but it is not visible in simulation waveforms. We shall study this ordering, which is enforced in the form of the *verilog simulator event queue*, later in the course.

Synthesis tools currently can not use concurrency in procedural blocks very well; they also ignore design delay expressions. To create a synthesizable sequence of events, it is necessary to make the later events depend on the earlier ones by defining the earlier ones as inputs (RHS) and the later as outputs (LHS), or by making the desired sequence depend on the sequential state of design variables, primarily `reg` types.

Order can be considered either unique or multiple. If an event (such as a hardware reset) occurs only once in the simulation lifetime of the device being modelled, it may be ordered uniquely with respect to other events. Obviously, something which happens on every clock, or on every number of them, can't be ordered uniquely; but, it may perhaps be ordered within the clock cycle or relative to some other repetitive event.

A *race condition* occurs when some state depends on the relative order of two or more preceding events, and the order of them can't be predicted. Clearly, this implies that both of the preceding events can occur at the same simulation time. The preceding events are said to be in a race to determine which one comes before the other(s). This is a very undesirable condition for the hardware, because unpredictable hardware usually is not functional. If the synthesizer doesn't reject such constructs as errors, it may synthesize the wrong hardware.

The simplest kind of race condition is repetitive and caused by assignments to the same data object from two or more concurrent blocks. If such blocks were not concurrent, the race would not occur, and the hardware would be functional. For example,

```
always@(posedge clockIn)
  begin
  #1 a = b;
  (other statements)
  end
...
always@(a, b, c) #2 ena = (a & b) | c;
...
always@(a, b)    #2 ena = a ^ b;
```

In each of the three statements in the example, the delay should be interpreted this way: If a variable in the event control expression changes to '1' or '0' (only to '1' for a *posedge*), the `always` block is read and thus the delay is initiated; after the delay has lapsed, the RHS is evaluated and the result is assigned.

The race in the code above is to assign `ena`. We assume other statements or blocks not shown are assigning to `a`, `b`, and `c`. This race is fairly obvious; but, it's not hard to imagine a bigger module which might cause confusion and conceal a race, because of complicated concurrency.

One might imagine that a designer would think of the simulation of the race example above this way: Start time = 0 at the clock edge. Then, at time 1, `a` gets the value of `b`. Suppose `a` thus has been changed to '1' or '0'; then, this triggers reading of the two other blocks. At time 3, `ena` goes to some value because of the second `always` statement – then again, depending on what was the value of `c`, perhaps `ena` has its value replaced by `0` at time 3, because of the third `always` statement? Hmmm ... may be a problem?

But, worse: What if `c` changes after time 1? Then this triggers the second `always` but not the third! Thus, `ena` may have its value changed arbitrarily either to the value of `(a&b)| c` or to `a^b`. Not only that, but the clock might intrude at any time to sample any arbitrary value of whatever is controlled by `ena`.

Also, what if `a` has been set equal to `b` before the clock edge? The clocked block won't change the value of `a`; then, neither of the other blocks shown will be read right after the clock; rather, they may be read at any arbitrary time because of changes in `a` or `b` from other blocks not shown. Yuk! This won't work.

The way to avoid this kind of race condition within a module is never to assign any data object in a module from more than one `always` block. When orderly execution is required, a simple solution is to have the `always` blocks read on opposite edges of the same clock, and to have all delays total less than half of a clock period in every block involved. Another way would be to provide two or more well-defined clocks derived at different phases from the same original clock.

Returning to the example, the first step in correcting the preceding race condition without using clocking schemes becomes obvious: Put all assignments to `ena` in a single `always` block, for example like this:

```
always@(posedge clockIn)
  begin
  #1 a = b;
  (other statements)
  end
...
always@(a, b, c)
  begin
  #2 ena = (c==1)? 1 : a & b; // An if could be used here.
  #2 ena = a ^ b;
  end
```

The sequencing of the two assignments to `ena` now is obvious, whereas it might have gone ignored or unnoticed in a big collection of independent `always` blocks. One here might question why the exclusive-or assignment should occur after the one above it, causing at most a glitch ignored by anything reading the outputs of this module. But, at least now there is no race condition. Perhaps the (inconsistent) intent of the original code would be realized better this way:

```
always@(posedge clockIn)
  begin
  #1 a = b;
  (other statements)
  end
...
always@(a, b, c) #2 ena = (c==1)? 1 : a ^ b;
```

The `initial` kind of block shouldn't be overlooked in a discussion of race conditions. Verilog allows any number of `initial` blocks in a module, and the same races can occur as with `always` blocks, except that usually such races will be nonrepetitive. The synthesizer ignores `initial` blocks, so they can cause no synthesis problem – except a simulation/synthesis mismatch!

`initial` blocks should be used only in testbenches or, in a design, for special purposes, such as denotation of an SDF back-annotation file. However, we should nevertheless point out the following:

Both `always` blocks and `initial` blocks are equally concurrent, and, at time 0, either one might be the first to cause evaluation of what should be assigned to something. Because it is unavoidable to assign some objects both in one `always` block and in an `initial` block, brief race problems may be overlooked or tolerated in simulation as a minor nuisance. In the following example, if the `always` block is read before the `initial` block, the `Reset 1'b0` will be missed at time 0, and `Abus` may not be set to `0` until `Reset` is toggled later:

```
initial
   begin
   Reset    = 1'b0;
   #0 Reset = 1'b1;
   ...
   end
always@(ClockIn, Reset)
  if (Reset==1'b0)
       Abus <= 'b0;
  else Abus <= Inbus;
```

Another problem can be caused by mixing blocking and nonblocking assignments in a single procedural block: Don't do it. Determine whether any of the logic will be clocked; if so, use only nonblocking assignments. For combinational logic, we want setup time for clocking, and, any change has to be propagated throughout the whole block. Because nonblocking assignments will update with the earliest, not the immediately new, values, if all the logic will be combinational, use blocking assignments. When the logic is sequential but unclocked (latched), be very careful of correct synthesis, and use either blocking or nonblocking assignments, depending on setup requirements. If necessary, repartition your design (within a module) so the sequential and combinational logic is more cleanly separated.

In closing this discussion of race conditions, it should be mentioned that verilog simulators use the same pulse-propagating process as most of the other available digital simulators; it is called *inertial delay*. An input pulse has to have a certain amount of "inertia" to get through a gate and influence the output. By default, if a module has been assigned a propagation (or path) delay, and a new level in an input lasts for less time than the propagation delay, the change will be ignored. This is a form of glitch filtering; only glitches wide enough will make it through the gate. Physically, it is a coarse model of energy response: A brief pulse is assumed not to have enough energy to switch the gate; and, the path delay value being available, it is used to decide this energy. Unlike other HDL's, verilog allows the threshold inertial pulse width to be controlled by the designer; we shall study this later in the course.

Simulators designed for VLSI design often do not simulate inertial delay properly, unless the delay is given in a `specify` block (to be covered later). So, unless you have tested it, do not depend on your simulator to swallow glitches based on hand-entered delays in continuous assignments or procedural blocks.

7.1.3 Unknowns in Relational Expressions

We shall return to this later in more detail, but, for now, it is enough just to say that relational expressions always evaluate to '`x`', which is not *true* (*false*) when a bit neither is '`1`' nor '`0`'. Thus, any '`x`' or '`z`' causes failure of *true*.

For example,

```
reg[3:0] A, B;
...
A = 4'b01x1;
B = 4'b0001;
if ( A > B )
     X = 1'b1;
else X = 1'b0;
```

The result is that `X` goes to `1'b0`. Perhaps surprisingly, with values assigned as above, the same result occurs for this:

```
if ( A == A )
     X = 1'b1;
else X = 1'b0;
```

To handle the four verilog logic levels literally and individually, one may use a `case` statement. With `A` equal to `4'b01x1` as above,

```
case (A)
4'b0000: X = 1'b0;
4'b0011: X = 1'b0;
4'b01x1: X = 1'b1;
default: X = 1'bx;
endcase
```

we get `X` set to `1'b1`.

The `case` statement only does exact matches; it does not permit wildcards. There exist two special wildcard variants of the `case` statement named `casex` and `casez`; we shall ignore them for now.

7.1.4 Verilog Operators and Precedence

We've already used the verilog bitwise operators most often seen in a design. Here is a list of all the operators, as given in the Thomas and Moorby (2002) (appendix C), and in IEEE 1364 (5.1).

Symbol	Type	Symbol	Type	Symbol	Type
~	bitwise	*	arithmetic	>	relational
&	bitwise	/	arithmetic	<	relational
~&	reduction	+	arithmetic	>=	relational
|	bitwise	−	arithmetic	<=	relational
~|	reduction	%	arithmetic	==	equality
^	bitwise	**	arithmetic	!=	equality
~^,^~	reduction	!	logical	===	equality (case)
>>	shift	&&	logical	!==	equality (case)
<<	shift	||	logical	? :	conditional
>>>	shift (arith)	{ }	concatenation		
>>>	shift (arith)	{ *n*{ } }	replication		

Any bitwise operator except ~ also is a reduction operator. There are two alternative ways of writing the *xnor* reduction operator; the present author prefers ~^.

Corresponding logical operators and bitwise operators yield the same results on one-bit operands. '1' for true is the same as `1'b1`, for all purposes. When manipulating bits, busses, or registers, or when expecting to synthesize gates to perform operations, it is best to provide an explicit width, just to keep in mind what one is doing.

The arithmetical shift right operator (>>>) shifts in the value of the preoperation sign bit instead of a '0'. The other shift operators shift in a '0'.

Synthesizable verilog generally imposes limits on the exponentiation operator (**); for example, both operands must be constant; or, the expression must evaluate to a power of 2. As a rule, wherever practical, avoid exponentiation by writing "1<<*n*" to evaluate 2**n*.

The case equality operator (=== or !==) is not used in a `case` statement; it may be used in an `if` or a conditional expression. It works like the comparison done in a `case` (but not `casex` or `casez`) statement, which distinguishes 'x' from 'z'. It can not return a value of 'x', as can the other equality, the bitwise, or the relational operators. The case equality operators do not allow wildcarding and do not interpret 'x' or 'z' as wildcards. Case equality is ***case*** equality, not ***casex*** or ***casez*** equality.

The replication operator may be used to assign a value or pattern to a part select or a whole vector. For example, "`Xbus[15:0]<= {Abus[4:1], 6{1'bz, 1'b0}};`", to set the 12 low-order bits to an alternating pattern of 'z' and '0'.

All operators except the conditional operator associate left to right when of equal precedence. All unary operators are of the highest precedence. Here is a table of operator precedence, after IEEE Std 1364:

Precedence	Operator
lowest = 0	? : (conditional) { } (concatenate)
1	||
2	&&
3	|
4	^ ~^ ^~
5	&
6	== != === !===
7	< <= > >=
8	<< >> <<< >>>
9	+ - (binary)
10	* / %
11	**
highest = 12	+ - ! ~ & | ^ ~& ~| ~^ ^~ (unary)

This ends our discussion of verilog operators. Now you know them all ...

7.2 Digital Basics: Three-State Buffer and Decoder

Before starting the next lab, let's look at a couple of design basics as they are realized in verilog.

First, let's introduce the verilog three-state buffer component shown in Fig. 7.1:

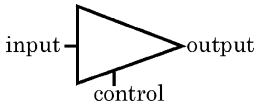


Fig. 7.1 `bufif1`, with pin functionality labelled

The **`bufif1`** is a verilog primitive gate which, when on, simply amplifies, or buffers, its input logic level. When off, it outputs a '`z`'. Its port list includes one *output* bit, one *input* bit, and one *control* bit, in that order from the left.

`bufif1` *optional_InstName* (*out*, *in*, *control*);

When the control bit is a logic '`1`', the gate is on; so, it is a ***buffer if*** control is '**`1`**', and this explains its name. We shall use this gate to exercise what we have learned about contention among different driving strengths.

Next, how do we do a verilog decoder? For a 2-to-4 decoder, the simplest way is to use a `case` statement. We can use a `case` as though it were a lookup table to assign the one-hot '`1`' depending on which 2-bit count is in the `case` expression.

With a binary count in `Sel`, to run through all possible bit patterns, and the decoded result in `xReg`, this is our `case` decoder for the *X* buffer selection in the next lab:

```
reg[3:0] Sel, xReg;
...
always@(Sel)
    begin
    case (Sel[1:0])
      2'b00: xReg = 4'b0001;
      2'b01: xReg = 4'b0010;
      2'b10: xReg = 4'b0100;
      2'b11: xReg = 4'b1000;
    default: xReg = 'bx; // e. g., to handle an 'x' in Sel.
    endcase
    end
```

This will be our first explicit use of the `case` statement in a lab. It is very important not to leave open an alternative; this means always to add a `default` statement to cover leftover alternatives. Setting the default to assign '`x`' values tells the logic synthesizer you don't care about leftover alternatives, and this permits better area optimization than otherwise. Also, overlooking an alternative may create a latch of some kind, and this may not be what you want. We'll look later at other implications of this very useful verilog construct, the `case` statement.

7.3 Strength and Contention Lab 9

Do the nonoptional work of this lab in the `Lab09` directory. If you have either the Thomas and Moorby or the Palnitkar books, they come with a demo version of the Silos simulator, which may be used for the optional Steps of this lab, a wire strength exercise. The demo version of Silos is not intended for large designs.

The optional steps of this lab probably will not work as described in a simulator which is optimized for use in CMOS VLSI design; such a simulator never would be used to resolve strengths; instead, speed and capacity for synthesizable verilog would be its goal. Strength is not synthesizable as a netlist gate output, and contention, except for '`z`', typically is assumed in a big design to imply only unknowns.

Lab Procedure

Work in your `Lab09` directory.

In this lab, we shall create a module named `Netter`. This module will contain logic as shown in the block diagram of Fig. 7.2.

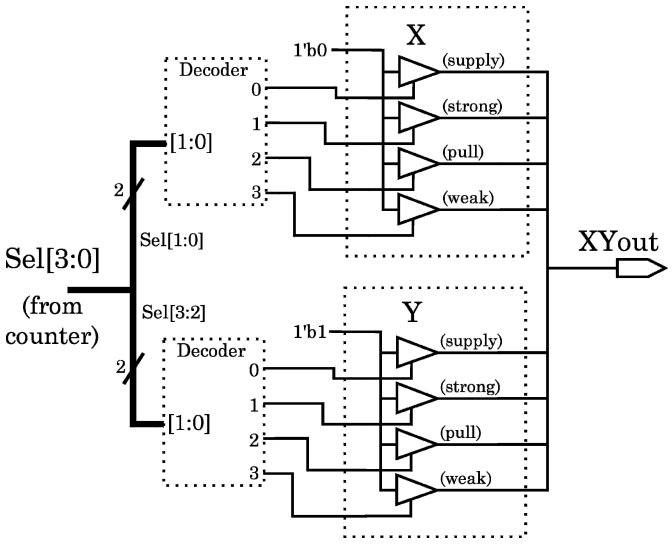


Fig. 7.2 `Netter` functionality. Blocks with dotted borders indicate logic clouds, not submodules

Notice that the `Netter` output drivers are all three-state buffers. It is illegal in verilog to drive an output with a simple wired logic net (`wor`, `wand`, etc.), for example, one connected directly between the module inputs and outputs. This probably is because no localized delay can be associated with the logic of such a net (as opposed to a path on a net from a gate output).

We wish to run a simulation which will demonstrate the result of contention of each of the *four drive strengths* against the others, including itself. To do this, we shall connect together some buffers and turn them on and off selectively so that only two are on at a time. One of the two always will be at logic level '`1`', and the other at '`0`'. Thus, we can see during simulation which strength wins a contention of two opposite levels.

This means $4 \times 4 = 16$ different pairwise connections. So, to explain the schematic of Fig. 7.2, we define four `bufif1`'s as the X buffers and four others as Y. All the X buffers receive a '`0`' input, and all the Y's a '`1`'. Each buffer in X or Y is assigned a different one of the 4 drive strengths. Then, we may use a 4-bit binary counter and assign the lower two bits (2 bits $= 4$ choices) to control X and the upper 2 bits to control Y. If we decode the count, we then can control each of X and Y to have just one buffer on at a time. By decode, it means here that the binary count is represented "one-hot" – in other words, a 2-bit count is represented by a logic '`1`' in one of 4 positions on a 4-bit bus.

Because a 4-bit binary counter goes through all possible combinations as it counts, if we tie all the X and Y buffer outputs together, we shall get all possible combinations of one buffer on among the X buffers, and, correspondingly, one on among the Y buffers.

Step 1. Enter two decoders. Begin the `Netter` module by writing the header. Use the schematic in Fig. 7.2. Declare a module named `Netter` with one 4-bit input port and wire a 4-bit internal net `xChoice` to the `xReg` of the *X* decoder as shown in Fig. 7.2. For the decoder, be sure to use an `always` block with a `case` statement as in the code immediately above. Then add another `case`, switching on `Sel[3:2]`, in the same `always` block, for the *Y* buffers. Wire that output (`yReg`) to a new 4-bit internal net named `yChoice`. Don't bother to connect the *x*`Choice` wires to anything yet.

The incomplete `Netter`, at this point, should be as represented by Fig. 7.3.

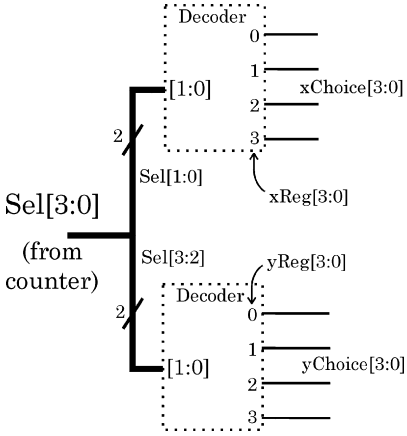


Fig. 7.3 Output wiring of the `Netter` decoders

Because the decoded outputs will be assigned from constants, there is no reason to make the `always` block sensitive to anything but the `Sel` bus.

To check your model, connect a counter and simulate it in VCS or QuestaSim; use a testbench which sometimes provides an '`x`' bit in the input. These simulators are meant for large CMOS designs and can not resolve strengths usefully, but they will find syntax errors for you.

Optional Step 2. Using your module from Step 1, you should add `bufif1`'s as shown in the code below, tying their inputs to logic '`1`' or logic '`0`' and their controls to the decoder outputs.

Define the strengths by assigning them in the `bufif1` instantiations; you should instantiate two each of the following `bufif1`'s in `Netter`. See Step 3 for a hint on how to fill in the wire names and the instance names:

```
bufif1 (supply1, supply0)  inst_name( , , );
bufif1 (strong1, strong0)  inst_name( , , );
bufif1 (pull1, pull0)      inst_name( , , );
bufif1 (weak1, weak0)      inst_name( , , );
```

Optional Step 3. After declaring the 8 `bufif1`'s, enable one buffer at a time by wiring them this way to the decoder outputs:

```
wire[3:0] xChoice; // Just to rename xReg.
 ...
assign Xin = 1'b0;
assign Yin = 1'b1;
 ...
assign xChoice = xReg;
//
bufif1 (supply1, supply0) SupplyBufX(SupplyOutX, Xin, xChoice[0]);
bufif1 (strong1, strong0) StrongBufX(StrongOutX, Xin, xChoice[1]);
  ... (2 more X and 4 more Y)
```

Optional Step 4. After this, tie all the buffer outputs together; this may be done by a set of continuous assignments all to the same wire. Such assignments of course are concurrent, so there is no need to worry about their order in the verilog source code:

```
  ...
assign XYwire = SupplyOutX;
assign XYwire = StrongOutX;
  ...(6 more) ...
```

Then, the contention may be examined by looking at `XYwire` in a simulator such as Silos, which can display strength differences; the result also may be assigned to a `Netter` output port, but if it is assigned to a `reg` type, the strength will be lost and only the logic level at strength = *strong* will remain.

Don't clock `Netter`, because all you want is combinational (muxed and decoded) nets. You had to declare `reg`'s to use the `case` statement, but these `reg`'s will be assigned to nets continuously and never will be allowed to save state when one of their drivers changes; so, they will represent combinational logic – if you have all `case` alternatives covered!

Optional Step 5. Clock a counter in your testbench to assign the value of `Sel` and step through all contention alternatives.

After seeing the simulation, you may synthesize `Netter` with no area optimization and examine the netlist. What did the synthesizer do to implement the decoders? Was strength preserved? Optimize for area and examine the netlist again.

This completes our exercise in wire strength and contention, and it ends our dependence on the use of the Silos simulator for this lab.

Step 6. Race condition exercise. Create a new module named `Racer` in a new file and put in it two `always` blocks as follows:

```
always@(DoPulse)
  begin
  #1 RaceReg = 1'b0;
  #1 RaceReg = 1'b1;
  #1 RaceReg = 1'b0;
  end
//
always@(DoPulse) #1 RaceReg = ~RaceReg;
```

Instantiate `Racer` in a testbench which toggles the input `DoPulse` a few times. Simulate: (Note: This will not synthesize). The first block should cause a 1 ns positive pulse lagging every `DoPulse` edge by 2 ns. But, what effect has the second `always` block? If it inverts after the first '`0`' assignment, then it should advance and widen the pulse; if it inverts before the first '`0`', it should be superceded by the '`0`' and should not be noticed. Reverse the locations of the two `always` blocks in `Racer.v`. Does this change the result? It should, because concurrent blocks may be read in any order, according to the simulator developer; and, generally, the order of appearance in the file will be used consistently, one way or the other.

Figures 7.4 and 7.5 show the waveforms in VCS or QuestaSim; what about Silos or Aldec?

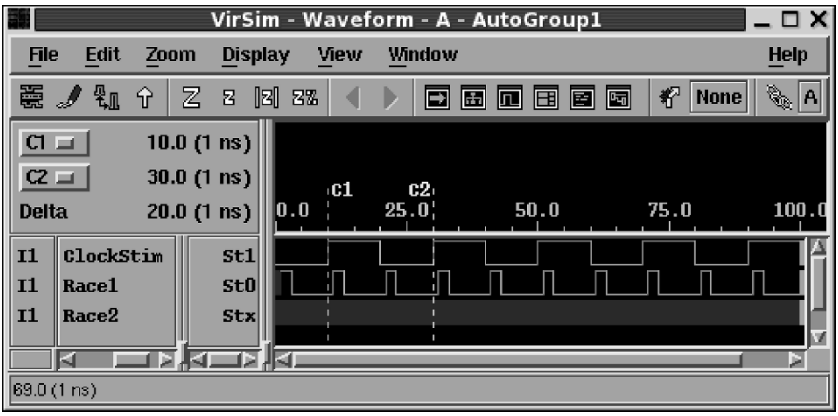


Fig. 7.4 Case 1: Inverting `always` first

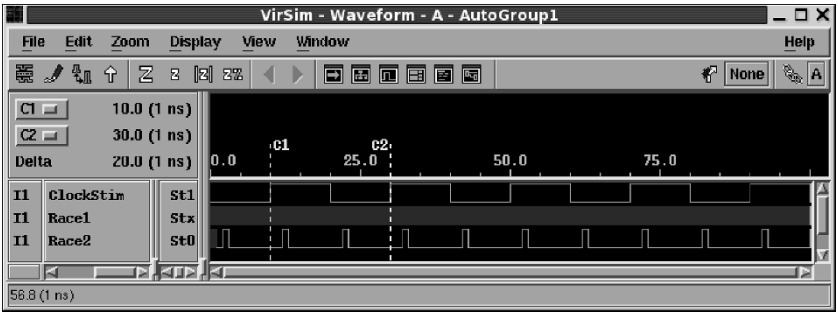


Fig. 7.5 Case 2: Inverting `always` second

Step 7. Operators and precedence. It is easy to get confused about the *and* and *or logical* vs. *bitwise* operators. For one-bit variables, they produce the same results. But for multiple-bit variables, they differ importantly.

For example, consider the three different bit-masks in these expressions (the left value "masks" the right one):

```
3'b110 & 4'b1111,
3'b010 & 5'b00111,
3'b101 & 3'b111.
```

The '`&`' is bitwise; so, the three results, in order, numerically are, `110`, `10`, and `101`, widened on the left by `0`-extension to the width of the destination vector (which is not shown).

However, "`&&`" is a logical operator, and it only returns a '`1`' (`= 1'b1`) if both operands are nonzero, or a '`0`'. So, the expression

```
3'b010 && 6'b111000
```

evaluates to '`1`', again widened by `0`-extension to the width of the destination vector. By contrast, the bitwise expression

```
3'b010 & 6'b111000
```

evaluates to '`0`', widened to the destination width.

Look at the expressions above and imagine them assigned to an 8-bit bus. What happens if one bit is '`x`' or '`z`'?

Does the logical operator return an '`x`'?

Are logical and equality operators different in the way they handle an '`x`'?

For this Step, test your insight by coding a small module `Operators` in its own file and simulating the three bit-masked expressions at the start of this Step. Use destination operands of 3 and 8 bits width. Then, replace the bitwise operators '`&`' with logical operators '`&&`' and simulate again.

Optional: If you have the time, use the simulator to evaluate the following two results, for `a` and `b` both '`1`', and `c` and `d` both '`0`':

```
NoParens = a&!d^b|a&~d^a||~a^b^c^a&&d;
Parens   = ((a&(!d)^b) | (a&(~d)^a)) ||((~a)^b^c^a) && d;
```

The point of this is that to understand someone else's mess, break the expression at the lowest-precedence operator(s); insert parentheses, and break again at the next lowest, etc. To prevent your own mess, use parentheses and spacing to clarify the meaning of complicated expressions.

7.3.1 Strength Lab postmortem

How do VCS and QuestaSim handle resolution of contention?

Is contention important when two concurrent statements assign the same logic level to a net?

7.4 Back to the PLL and the SerDes

7.4.1 Named Blocks

Several verilog constructs, which we shall study more closely in the next chapter, depend on being able to *name a block* of code. A name is assigned to any verilog block simply by following the `begin` with a colon and a valid verilog identifier. Of course, only procedural code can be in blocks, so only it can be named. A block without `begin` can't be named; so, if necessary, one simply inserts `begin` and `end` around anything to be named. No semicolon follows the identifier. Any reasonable alphanumeric string is a valid identifier if it does not begin with a decimal numeral.

For example,

```
always@(negedge clk)
   begin : MyNegedgeAlways
   ...
   end
```

Or,

```
begin : Loop_1_to_9
for (j=1; j<=9; j = j + 1)
   begin
   ...
   end // for j.
end // Named loop block.
```

Naming a block has no functional effect. However, a block name, like a module name, can be used by a tool and thus may be found in a synthesized netlist or a simulator trace list. Thus, naming blocks can help in debugging or optimizing a netlist.

Although the name itself changes nothing, it can be used to create new functionality. For example, naming the block in a looping statement such as `for`, `while`, or `forever` allows the loop to be exited by the **`disable`** statement. This is similar to the *C* language `break` statement.

7.4.2 The PLL in a SerDes

Recall the SerDes design blocked out in *Week 2 Class 2*. Our full-duplex system will require a serial transmitter (`Tx`) and receiver (`Rx`) in each direction. Each end of the serdes will be defined in a different system clock domain, these clocks being the clocks used to manipulate the parallel bus data within the two systems.

To make our design reflect the most general case, there will not be any synchronization between the two system clock domains. A serial clock will have to be generated for each `Tx`, to transmit the data over the serial line; of course, each `Rx` then will have to have a deserializing clock running at the same speed as the serializing `Tx` clock. We have decided that the system clock speed in each domain will be 1 MHz, and that all serial clocks will run at 32 MHz, these clocks being generated by four independent PLLs, one for each `Tx` and one for each `Rx`. A clock ratio of 32:1 easily is achievable by an analogue PLL currently in use in the industry.

To avoid unnecessary complexity, we shall not address error detection or correction, encryption, compression, edge detection, or packet retries, of the serial data transferred. The protocols involved would be too time-consuming for the present course.

The design on the `Tx` side of our serdes is straightforward: The parallel data are clocked in on a 1 MHz system clock, and the `Tx` PLL is used to serialize and clock them out at 32 Mb/s. The `Tx` PLL can track the system clock directly, its phase being constrained only by that of the well-defined system clock. Because we have decided to transmit packets 64 bits wide per 32 bit data word, the maximum speed of transmission on each serial line will be one word on every other system clock; thus, each receiver also will run at a maximum deserialization rate of one word per two system clocks.

The design on the `Rx` side will have to be more complicated. Because the two systems are clocked independently (see Fig. 7.6), the `Rx` PLL will have to extract the `Tx` clock from the serial data. This is equivalent to saying that the receiver will have to identify incoming packet boundaries and use their arrival rate to provide a 1 MHz clock for the `Rx` PLL. Given the extracted 1 MHz clock, the `Rx` PLL can synchronize to it and generate the 32 MHz `Rx` serial clock required to clock in the data and deserialize it, one word per two extracted 1 MHz clocks. After deserialization, the incoming data words will have to be clocked out on the receiving system clock.

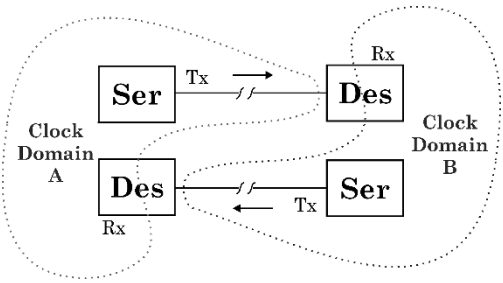


Fig. 7.6 Full duplex SerDes and the two system clock domains

To allow slack for `Tx` synchronization of the PLL serial clock and the transmitting system clock, each serializer will include a FIFO to buffer incoming data. This FIFO will reduce loss of data words by averaging out PLL-caused variations in serialization speed to match the average rate of arrival of valid parallel data from the transmitting system.

Each deserializer also will include a FIFO to take up the same kind of slack. In addition, the `Rx` FIFO will reduce data loss because of possible delays in synchronization of the two system clocks, the extracted `Tx` system clock with the `Rx` system clock. In many systems, only the receiving end would be designed with a FIFO; however, we shall add one to the transmitting end, perhaps to allow for irregularity in the sending rate.

We shall discuss more general clock-synchronizing technology later in the course.

7.4.3 The SerDes Packet Format Revisited

To construct a serial data source allowing synchronization, we shall use the data packet format previously given as,

```
64'bxxxxxxxx00011000xxxxxxxx00010000xxxxxxxx00001000xxxxxxxx00000000,
```

in which each '`x`' represents one data bit in a serialized 8-bit byte value. The format shown is sent and received MSB first. To provide a synchronizing, incoming clock for our `Rx` PLL, we shall look for a pattern in the input data stream of a pad-byte down-count from `2'b11` to `2'b00`.

To make things easy for a start, let's send just the same 4 bytes repeatedly: For this, we shall pick the ASCII codes for '`a`', '`b`', '`y`', and '`z`', in that order. These codes are, respectively, `8'h61`, `8'h62`, `8'h79`, and `8'h7a`.

At this design stage, then, our serial data source therefore repeatedly will send this binary data stream, left to right:

```
//      'a'      pad 3    'b'      pad 2     'y'       pad 1    'z'       pad 0
64'b01100001_00011000_01100010_00010000_01111001_00001000_01111010_00000000
```

The alphabetical interpretation of the 8 bytes $\times$ 8 = 64 bits is shown above the stream representation.

To test our deserializer as we write the verilog, we have to generate this data stream repeatedly at about 32 Mb/s. This is trivial: We just leave the data pattern above as a constant and pretend it is being received repeatedly over a serial line.

Next, we show how a software-oriented, behavioral (or, procedural) verilog model can be developed to synchronize a PLL to an embedded serial clock. This model will not be synthesizable; so, we shall return later to our previous frame-encoder approach to devise a synthesizable model. The next discussion is a language digression meant to accommodate software-oriented coding in verilog.

7.4.4 Behavioral PLL Synchronization (language digression)

A ***behavioral*** synchronization, or perhaps more accurately, *procedural* synchronization, is done easily in verilog, using a `for` statement to sample the stream, and another, containing, `while` statement which has no exit criterion. The construct `for(j=j; j==j; j=j)` would be preferred to a `while` for our learning purposes, but the synthesizer requires the `for` iteration variable to be initialized with a constant, which would not work in our application. So, we shall use `while(1)`.

In our procedural approach, you will notice that the model is not purely behavioral; it includes bit-level RTL procedural assignments. An outline of the model's loop would look something like this:

```
integer i, j;
reg CurrentSerialBit; // A design data object.
reg[63:0] Stream;     // For now, this is a fixed data object.
...
         // Concatenation used for documentation purposes, only:
Stream = {32'b01100001_00011000_01100010_00010000,
          32'b01111001_00001000_01111010_00000000};
begin : While_i
while(1) // Exit control of this loop will be inside of it.
  begin
  for (i = 63; i >= 0; i = i - 1) // Repeats every 64 bits, if no disable.
    begin
    #31 CurrentSerialBit = Stream[i]; // The delay value will be explained.
    ( do stuff with CurrentSerialBit ... then disable While_i )
    end // for i.
  end
end // End named block While_i.
... // Pick up execution here after disable While_i.
```

The real control of the outer `while(1)` is by the name on the block containing it, `While_i`. When "`disable While_i;`" is executed anywhere in the module containing the code above, everything scheduled between the named `begin` and `end` is terminated immediately, and execution continues below the end of the `While_i` block. As mentioned before, this works like a *C* language `break`.

Use of the disable statement: In verilog, any procedural block may be placed between `begin` and `end`, the `begin` labelled as above with a verilog identifier, and `disable` called on the block by name. Like `goto` in *C*, this should be used only when unavoidable, because it leads to complicated execution which may become difficult to debug when anything goes wrong or when it is desired to modify or improve the code.

With the example above as a start, we can decide how to synchronize the PLL to the data stream. In the `for` loop data stream above, the delay of 31 ns $\times$ 64 bits $=$ 1.984 µs period gives us about half of the speed we want for our approximately 1 MHz embedded clock. Just about right, to handle one parallel word every other clock.

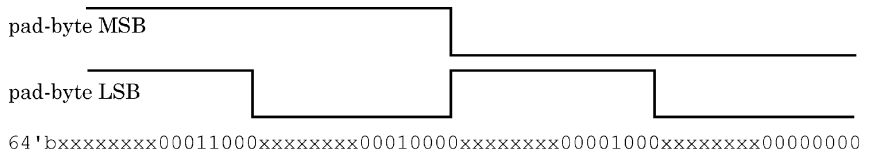


Fig. 7.7 The embedded clock (`EClock`) in the serial stream can be at twice the packet frequency. MSB and LSB refer to the 2-bit pad numbers

We know from a previous lab that, by design, our PLL clock is free-running at a nominal frequency of 32 MHz and continually monitors the frequency of its incoming 1 MHz `ClockIn` clock. In the present design, whenever the PLL receives a positive-edge `Sample` command, it uses the `ClockIn` edges it has been monitoring to make a small frequency adjustment toward that of `ClockIn`. So, we must supply a `ClockIn` nominally at 1 MHz ($= 32$ Mbs$^{-1}/32$) to the PLL, and we must issue regular `Sample` commands.

We therefore choose to issue a `Sample` command on every data packet received, and to supply the PLL with an embedded clock (`EClock`) defined by the toggle of the LSB in the 2-bit frame padding down-count. So, we shall sample on every other `EClock`; `EClock` will be connected permanently to the PLL `ClockIn` input. See Fig. 7.7.

To extract the embedded clock, `EClock`, we can use the pad pattern in the data stream; we should look first for 3 successive '`0`' bits, followed by any two more bits *nn*, which are read as a count value, followed by another 3 successive '`0`' bits. Each of these successive, padded *nn* values is separated from the previous one by 8 ignored (data) bits. The `nn` values will count down by 1 after every data byte, as the data stream is traversed from MSB toward LSB.

To set a somewhat arbitrary *synchronization criterion*, we shall do nothing until the down-count pattern has been established at least for 4 *nn* values and ends in `2'b00`. We then set the logic level of our `EClock` equal to that of the *nn* LSB; this initially will be `1'b0`. This should allow the PLL to determine the direction of a frequency correction, if any, and so we shall begin issuing `Sample` commands to the PLL.

If we *lose synchronization* as defined by any `nn` down-count miscount, we stop issuing `Sample` commands and leave `EClock` at its most recent level until synchronization is again established by the criterion above. Our PLL, of course, continues to provide an output clock at a frequency based on the most recent period of synchronization; it is up to other logic to decide what to do with this clock.

Given synchronization and desynchronization criteria, the question remains of how we should identify the pattern, `8'b000`*nn*`000`, where *nn* represents a 2-bit count.

To reach an answer to this question, the preceding outline of the model's loop may be expanded in detail and expressed as a flowchart in Fig. 7.8.

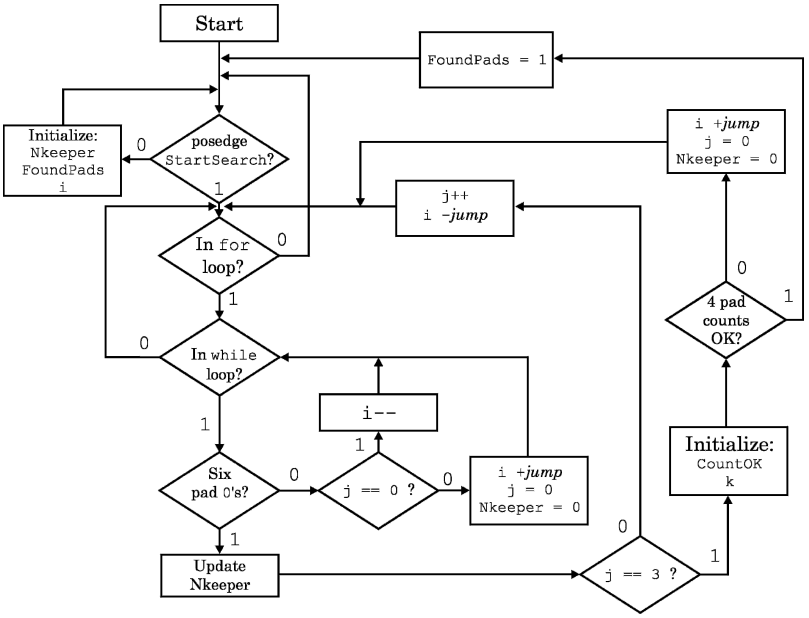


Fig. 7.8 `FindPatternBeh` flowchart used to develop the code fragments below

From the flowchart of this procedure, the data are arriving serially, so we can start the identification of a packet with a `for` statement that is triggered every `i`-th serial bit received, with `i` stored in a 6 bit (64-value) `reg`. A verilog `reg` is unsigned by default, so counting up or down in this `reg` connects 0 with 63 in either direction.

Let's look for the pad pattern assuming the *current* value of `i` has put us on the MSB + 1 byte = `SerVect[63-8] = SerVect[55]`, which is the first '`0`' in a correctly identified padded count-byte. One implementation in verilog might be by the following code fragment:

```
// Assume SerVect is a saved, serial 64-bit vector:
reg[5:0] i;         // i traverses the serial stream (64-bit vector).
reg n1Bit, n2Bit;  // Two 1-bit regs to hold the expected nn pad count.
...
FoundPads = 1'b0;
begin : While_i // Name the block to exit it.
while(1)
  begin
  if ( SerVect[i]==1'b0     && SerVect[i-1]==1'b0 && SerVect[i-2]==1'b0
       && SerVect[i-5]==1'b0 && SerVect[i-6]==1'b0 && SerVect[i-7]==1'b0
     )
     begin
     #1 FoundPads = 1'b1;     // 1 means true here, for later use.
     #1 n1Bit = SerVect[i-4]; // Save the padded nn = {n2, n1} value.
     #1 n2Bit = SerVect[i-3];
     disable While_i;              // Exit the while block, if found.
     end // if.
 ...
 i = i - 1;
 end // While_i statement.
end // While_i named block.
```

Study the code above; notice that the `j` of a previous example is not present. Be sure you understand what this code does in relation to our framing pattern if the *current* value of `i` puts it on the first '`0`' of the MSB pad byte (bit 55 in the vector above). Then, move `i`: Assume that the *current* value of `i` now is at bit 23, for example, in that pattern. If so, with the big `&&` expression in the `if`, we are looking at all the `0`'s in the byte in the part select, `SerVect[23:16]`, to see whether we can capture its *nn* count value.

The `i = i - 1` in the `while(1)` loop control means that the `&&` pattern match test will be run on the next less significant (`i-1`) position in the `SerVect` vector if a pattern match does not succeed at the current (`i`) position. This is because we assume the data stream is arriving MSB first; thus, over time, we assume that less and less significant bits will be at the center of our attention. Of course, in the special case of bit 23 of our `SerVect` above, the match will succeed, the `While_i` block will be `disable`'d, and no new `if` will be run at `i`$-1 =$ bit 22, given the code fragment shown.

The assignments shown in the verilog above all are blocking; we are not modelling hardware but are creating behavior. The delays are just to space out edges in the simulation waveforms, but we want every assignment to take effect before the next sequential line is read, just as in a software program. Later, perhaps some of this model will be seen as clearly sequential and thus perhaps will be implemented using undelayed nonblocking assignments.

Also, it might seem that the above pattern search would be speeded up by nesting six `if`'s:

```
if (SerVect[i]==1'b0)
  if (SerVect[i-1]==1'b0)
     ...
```

In terms of the verilog simulation language, this is a false impression, because the multiple && expression is evaluated left-to-right; and, on the first equality failure, the && will go false just as quickly as would a nested if in that position. However, the logic synthesizer might produce a different netlist for the nested if's, so perhaps a rewrite in the form of nested if's should be kept in mind.

Anyway, the code above will not set FoundPads = 1 unless what *probably* are the six pad '0' bits we are seeking have been found. Such a pattern match might, however, be a coincidence, and we might have matched by mistake on an i in the middle of a data byte. So, let's elaborate on our search further.

Let's declare a vector Nkeeper which is $4 \times 2 = 8$ bits wide, and use the code above to collect 4 successive, 2-bit nn values (which we think are nn values, anyway). This adds some more functionality to the code above; it can be written as follows:

```
reg[7:0] Nkeeper; // Stores 4 2-bit nn values.
reg[5:0] i;       // Indexes into a saved 64-bit SerVect vector.
reg[2:0] j        // Counts which of 4 assumed nn's we are on.
...
FoundPads = 1'b0;
i =  starting value;
for ( j=0; j<=3 ; j=j ) // The for j increment is nonfunctional.
  begin : While_i
  while(1)
    begin
    if ( six pad 0's; same as above )
      begin
      #1 Nkeeper[2*j+1] = SerVect[i-3];   // MSB of nn.
      #1 Nkeeper[2*j]   = SerVect[i-4];   // LSB.
      #1 j = j + 1;
      #1 i = i - 16; // Jump ahead to the assumed next pad byte.
      #1 disable While_i;
      end // if.
    i = i - 1;
    end // while.
  end // While_i block.
```

Minor point: The delays are located here as placeholders; they make the sequence of simulation events easier to see in a waveform display. When the model is working, these delays should be removed; ultimately, some assignments in the containing module might be changed to nonblocking ones, too. The only caution in adding such delays is that, if blocking, they add up, and the device clock must be slow enough to let them all time out under all conditions. If not, some events might be cancelled during simulation, and the model might fail to work.

The code above is the same as the preceding verilog, except that it repeats until j = 3. Setting of the pattern-match flag has been omitted for simplicity. The

width of `j` must be 3 bits or more; if it were just 2 bits, it would count past 3 to 0, and there never would be a count greater than 3; so, the outer `for` never would exit.

One problem with the above code fragment is that it might get stuck in the data and repeatedly jump past the pad bytes. Also, `Nkeeper` might get filled up with random values whether or not they came from pad bytes. For example, what if the stream contained no packet at all and was mostly '`0`' and only an occasional '`1`'?

We want to check for 4 *successive nn* pad values, separated by exactly 8 (data) bits each; so, we should modify the above. We shall decrement `i` by 1 only while searching for a pattern match and not finding it; when we find a match, we'll jump ahead by 16 bits (decrement `i` by 16) to look for another one. If we ever fail to find a subsequent match, we shall restore `j` to 0, increment `i` by 15 to get the first bit which was ignored by jumping ahead, and start the search again after advancing by 1 bit in the stream.

The result comes out something like the following:

```
reg[7:0] Nkeeper; // Stores 4 nn values.
reg[5:0] i;  // Indexes into a saved 64-bit SerVect vector.
reg[2:0] j   // Counts which of 4 assumed nn's we are on.
...
FoundPads = 1'b0;
i =  starting value;
for ( j=0; j<=3; j=j ) // No increment.
  begin : While_i
  while(1)
    begin
    if ( six pad 0's; same as above )
         begin
         #1 Nkeeper[2*j+1] = SerVect[i-3]; // MSB.
         #1 Nkeeper[2*j]   = SerVect[i-4]; // LSB.
         if (j==3) // We're done:
           begin
           #1 FoundPads = 1'b1;
           disable While_i;
           end
         // If j wasn't 3, jump ahead for another look:
         #1 j = j + 1;
         #1 i = i - 16;
         end
    else // No six-0 match this time.
         if (j==0)
              #1 i = i - 1; // First nn not found yet.
         else begin        // We found at least one nn, but now failed:
              i = i + 15;  // Drop back after jumping by mistake.
              j = 0;       // Reset nn counter; we're not in a pad byte.
              end
    end // while(1).
  end // While_i.
```

So, if the preceding block of code exits, we shall have stored 4 2-bit values in `Nkeeper` on the assumption that they will be a sequential, binary down-count in our packet format. Really, we should have dropped back by `j*16 - 1`, not just by 15; but, we'll fix that next time. But, is it true that in the above we have a way to find our pad pattern?

No, we haven't checked the counts. To be very sure of having found the pad bytes, and thus the packet boundary, we should add a check for the down-count. If we don't confirm a down-count, we have not actually found our packet framing, so, we should continue searching. We only allow the code to exit if we have found what seems to be a properly padded packet of data.

The easiest place to check for a down-count is in the block above which sets `FoundPads` to 1. We should add a branch there on whether `j` is 0 or not; if it is 0, there is only one *nn*, so, there will be no use checking it. But, if `j>0`, we can see whether the current 2 `nn` bits is exactly 1 less than the previous 2 *nn* bits.

First, let's break out the initialization of the above into a new `always` block which is read on the opposite edge of `StartSearch`. This way, even if we assign to the same variables from the two `always` blocks, there can't be a race condition unless the delay during one block extends to the other edge. This won't happen in our design, because `StartSearch` is not a clock but a search-enable signal.

```
always@(negedge StartSearch)
  begin
  #1 Nkeeper   = 'b0;   // Init count keeper every search start.
  #1 FoundPads = 1'b0;  // Move this init here.
  #1 i         = StartI;
  end
```

Then, the final searching block is as shown here:

```
always@(posedge StartSearch) // Clocked, maybe sequential logic?
begin : AlwaysSearch
for ( j=0; j<=3; j=j ) // No increment.  '<=' is relational operator!
  begin : While_i
  while(1)
    begin
    if ( six pad 0's; same as above )
         begin
         #1 Nkeeper[2*j+1] = SerVect[i-3]; // MSB of pad count.
         #1 Nkeeper[2*j]   = SerVect[i-4]; // LSB of pad count.
         // Check whether done:
         if (j==3) // We have 4 apparent nn values; do they count down?
              begin
              #1 CountOK = 2'b00;
              for (k=1; k<=3; k=k+1)
                begin
                // Use concatenation to get 2-bit nn values:
                #1 nPrev = { Nkeeper[2*k-1], Nkeeper[2*k-2] };
                #1 nNow  = { Nkeeper[2*k+1], Nkeeper[2*k] };
                if ((nNow+1)==nPrev) #1 CountOK = CountOK + 1;
                end //for k.
             if (CountOK==3) // Total of 4 were OK; so,
                    begin    // issue a pulse and stop everything:
                    #1 FoundPads = 1'b1;
                    #1 disable AlwaysSearch;
                    end
               else begin // If not a down-count, start all over:
                    #1 i = i + 16*j - 1; // See * below:
                    #1 j = 0;
                    #1 Nkeeper = 'b0;
                    end // if CountOK.
           end // if j==3
         else begin // j not 3:
              #1 j = j + 1;
              #1 i = i - 16; // Jump ahead, for another padded nn.
              #1 disable While_i;
             end // else not j==3.
         end // if 6 zero matches.
    else if (j==0) // First nn not found yet:
              #1 i = i - 1;
         else begin // * This was not first apparent nn found.
              #1 i = i + 16*j - 1; // Drop back after jump by mistake.
              #1 j = 0;            // Reset nn counter.
              #1 Nkeeper = 'b0;    // Reinit count keeper.
              end
    end // while(1) block.
  end // While_i labelled block.
end // always AlwaysSearch.
```

The preceding is a complete RTL design which correctly locates the packet pad bytes in our fake piece of serially streamed data. It would work as well, slightly adapted, to a serially varying input formatted to conform with our required framing protocol.

A copy of the RTL search code above, with a testbench, is provided in the `Lab10` directory and is in a file named `FindPatternBeh.v`.

7.4.5 Synthesis of Behavioral Code

The problem is that the model so far almost is a *C* model; it is not efficiently synthesizable. In fact, the code in `FindPatternBeh.v` probably will not synthesize at all.

One hint of synthesis problems is the presence all over of blocking assignments and delays. A test: Remove the delays: If the code still simulates correctly, it probably is synthesizable. For example,

```
if (CountOK==3)
     ...
else begin // If not a down-count, start all over:
     #1 i = i + IJump*j - 1;
     #1 j = 0;
     #1 Nkeeper = 'b0;
     end
...
```

If these `#1` delays were necessary to proper functioning of some other `always` block or module, this verilog would not synthesize to logic which was functionally correct.

Also, if one replaced all the blocking assignments in the code fragment with nonblocking ones, there would be a race condition, with the delays shown. If the assignments were changed to nonblocking, and selectively longer delays were provided, say `#2 j <= 0`, it might simulate correctly; but, again, synthesis probably would produce defective logic. For this fragment, the best solution would be to remove all delays entirely.

Delays aside, it can be difficult to rewrite a behavioral model in synthesizable form if the model is of any complexity and requires specific delays. The reader may consider spending a little time thinking about how to convert the behavioral model above, but we shall take a different tack in writing a synthesizable version.

7.4.6 Synthesizable, Pattern-Based PLL Synchronization

Let's go back and look again at the problem: We have a stream of serial data and we wish to synchronize a clock to the framing pattern. In our project, the pattern is 64 bits wide, so it makes no sense to worry about anything less than 64 bits wide.

As an alternative to the behavioral, *C*-like coding presented previously, we can do this: Assume we shall have a dynamic sampling "window" of the serial stream

which is 64 bits wide and stored in a register. We check in the window for a framing (pad) pattern; if we find it, we synchronize our clock to a boundary which is well-defined in the frame; if we don't find the pattern, we shift the window by one new bit (= serially shift in a new bit) and check again. If our window only changes one bit at a time, we cannot fail to find the frame boundaries, if they are there.

So, all we need do is check a completely static window of data. This can be done concurrently by checking every pad bit, all 32 of them, in the 64-bit sample we have. There is no need to shift or change anything, so there is no sequencing or delaying of anything at all in simulation time. All we have to do is agree not to change anything in the register while we examine the sample for our frame boundaries.

First, we define our pad boundary patterns; then, we decide where to look for them in the 64-bit window. And, that's all there is to it.

Actually, there's less to it than that: We can agree only to look for the entire 64-bit pattern centered in the window (serial packet MSB at the window MSB position). Why not? If the actual serial stream is not found to be centered on one sampling, we just shift a new LSB serial window bit in, shift the old window MSB out, and check; shift and check; and just keep doing this until we have the framed pattern centered; then, we recognize it, and off we go!

Using this approach, all we have to do in our model is choose bits to which to attach comparator logic (*xor*'s in the netlist, maybe). It might go something like this:

```
reg[63:0] SerVect = // The current 64-bit window to scan.
/*    60         50         40          30         20         10          0
 *  32109876 54321098 76543210 98765432 10987654 32109876 54321098 76543210 */
64'b01100001_00011000_01100010_00010000_01111001_00001000_01111010_00000000;
...
localparam[PadHi:0] p0 = 8'b000_00_000; // The pad patterns.
localparam[PadHi:0] p1 = 8'b000_01_000; // localparams can't be overridden.
localparam[PadHi:0] p2 = 8'b000_10_000;
localparam[PadHi:0] p3 = 8'b000_11_000;
...
  if (   SerVect[55]==p3[7] && SerVect[54]==p3[6] && SerVect[53]==p3[5]
      && SerVect[52]==p3[4] && SerVect[51]==p3[3]
      && SerVect[50]==p3[2] && SerVect[49]==p3[1] && SerVect[48]==p3[0]
      && SerVect[39]==p2[7] && ... (total of 32 compares)
     ) Found = 1'b1;
  else Found = 1'b0;
   ... (etc.)
```

This will be much easier to do after we have studied functions, which will be next time. So, for now, we leave this problem as-is and shall return to it later.

7.5 PLL Behavioral Lock-In Lab 10

Work in the `Lab10` directory.

Lab Procedure

In this lab, we shall exercise use of the `for` statement as a way of sampling a data stream. Thomas and Moorby (2002) recommends varying among `for`, `while`, and

forever, depending on the context; however, in practice, the for statement can do anything these others can; we shall use it extensively here. The others usually are simpler, so we shall exercise for to understand its complexity (and its benefits) better.

Step 1. Make a subdirectory in Lab10 named PLLsync. Copy the entire PLL design from the Lab08/Lab08_Ans/PLL directory into the new PLLsync directory. This design will have had a PLL clock which is applied to three different counter structures. The top of the design should be a module named ClockedByPLL.

You probably have a complete design of your own for this, from your Lab 8 work, but the lab instructions will work best if you use the answer files provided.

Reorganize the files. Assume there are too many files in one place in this design. Make a new subdirectory named PLL, and move the five PLL module files into the new PLLsync/PLL directory. These files already should be named after the ClockComparator, MultiCounter, and VFO submodules making up the PLL, and PLLTop.inc, a defined-constant include file. The PLLTop.v file also should be in PLL and should contain the PLL top level module which connects these submodules. The top level module should be named PLLTop; rename it, if it is not already so.

Create a simulator file list file, PLLsync.vcs, one level up (in the PLLsync directory), and invoke the simulator briefly so you are sure it can be run with the PLL design moved into the PLLsync/PLL directory. This is just to check the file locations.

Step 2. Rename and revise the design top. Change the module and file names of the three-counter design from ClockedByPLL to PLLsync. Remove all counters except the behavioral one, so that the only count output is the behavioral one. Discard the DFF.v model.

The block diagram of your new PLLsync design, which in Lab08 once was called ClockedByPLL, now should look like Fig. 7.9.

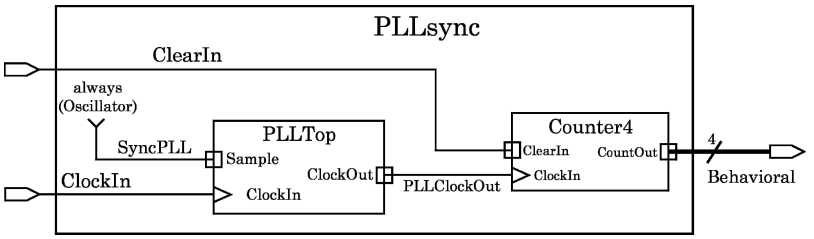


Fig. 7.9 The PLLsync design block diagram

Simulate PLLsync to verify its correct functionality. Use a 1 MHz ClockIn.

Step 3. Modify a pattern-finder to extract a clock and a PLL sample command. Make a copy of `FindPatternBeh.v` (originally placed for you in the `Lab10` directory; see left side of Fig. 7.10) named `EClockSample.v` ("E" for extract). The module in `FindPatternBeh.v` is named `FindPattern`; rename the module in the copy to `EClockSample`.

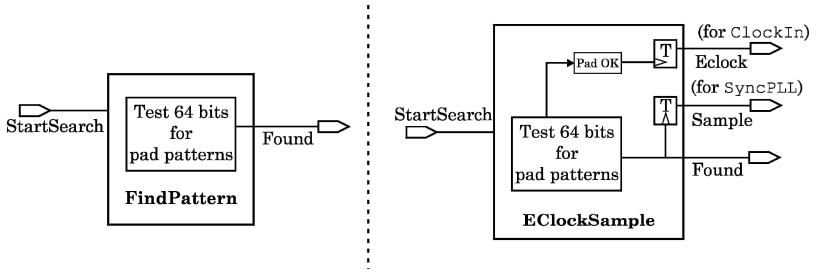


Fig. 7.10 `FindPattern` can be modified trivially to `EClockSample`, to provide inputs for `PLLsync`. `T` = toggle; `StartSearch` would be triggered on every new received serial bit

Modify the new `EClockSample` module so it outputs our required extracted clock and sample command, as we specified above. A D flip-flop with its `Q` tied back to its `D` often is called a ***toggle*** flip-flop, or, `T` flip-flop, shown in the right side of Fig. 7.10.

Do ***not*** try the pattern-based approach, which was a loopless, synthesizable design introduced at the end of today's serdes presentation. Do not worry about actual sample speed in your test bench; just extract the `EClock` and the `Sample` command as you search through the serial bit pattern with the unsynthesizable `FindPattern` looping approach.

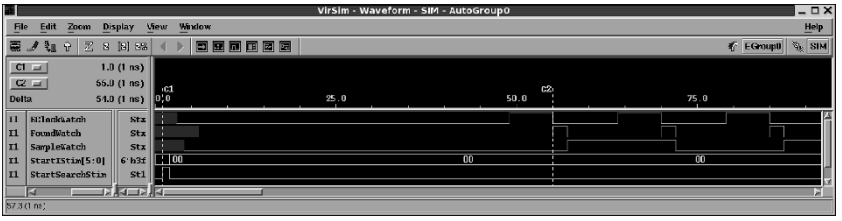


Fig. 7.11 Simulation of `EClockSample`, showing the extracted clock and the sample command

Thus, `EClockSample` simply should output the value of the pad-pattern counter LSB continuously, renamed to `EClock` (wired to `EClockWatch` in the testbench used for Fig. 7.11). While `Found` is asserted, the `EClock` is good (synchronized); while `Found` is not asserted, the `EClock` can not be depended upon.

Do not try yet to attach the `PLLsync ClockIn` to the `EclockSample EClock`. However, keep in mind that, later, we may use `EClock` to adjust the PLL frequency when `EClock` is known good; and, we can let the PLL oscillator run free when `EClock` is not known good.

7.5.1 Lock-in Lab Postmortem

How should one deal with establishing and losing synchronization?

7.5.2 Additional Study

Read Thomas and Moorby (2002) section 4.6 on **`disable`**.

Thomas and Moorby (2002) section 6.5.1 gives a truth-table for the `bufif1` primitive.

Read Thomas and Moorby (2002) sections 10.1 and 10.2 on strength and contention.

Read Thomas and Moorby (2002) appendices C.1 and C.2 on verilog operators and precedence. The operator functionality is very important to know thoroughly; precedence is less important, because it can be defined or overridden by parentheses.

Optional Readings in Palnitkar (2003)

Section 3.2.1 and Appendix A on verilog builtin net types and strengths.
Section 6.1.2 on *implicit* net declarations.
Look through sections 6.3 and 6.4 on verilog operators.

Chapter 8
Week 4 Class 2

8.1 State Machine and FIFO design

This time we shall study FIFO design and the state machines required for it. First, though, some tools for simplifying the verilog.

8.1.1 Verilog Tasks and Functions

The structure of a design is defined by its hierarchy of module instances. Any common functionality can be put into a module and then instantiated as many times as desired in a design. However, sometimes, repetitive functionality is required which is procedural and should not be visible as part of the design hierarchy. As we have seen, operating on numbers or logic states by using operators such as '+', '&', or "==" is far more convenient than wiring up adders, *and* gates, or comparators. Likewise, defining a shift-register in an `always` block with one statement is much quicker than wiring one from individual gates or even wiring one in as a module instance. In verilog, convenient handling of this kind of commonplace, repetitive, procedural functionality is made available to the user in the form of ***tasks*** and ***functions***.

Both of these constructs are declared in the module in which they are intended to be used. Although they may be called remotely by a hierarchical reference, this is bad design practice and is discouraged. If reuse in different modules is required, the declarations may be put in a file and the file then may be introduced into a module by means of a `` `include ``. Because neither tasks nor functions represent design structure, their declarations can not contain local `wire`s to wire their contents to anything; however, they may contain local `reg` variables for temporary data manipulation.

The difference between tasks and functions has to do with complexity and, more importantly, with timing. We refer here only to *user-defined* tasks and functions, not to the verilog language's builtin system tasks or system functions.

Complexity. Functions are simpler. A task can call a function or another task; a function can not call a task but may call another function. So, tasks potentially are more complicated than functions. In addition, a function just changes one thing when it is executed, its return value; in effect, a function merely defines an expression which is evaluated each time the function is called. A task can modify external `reg` objects as it executes, possibly leaving a diversity of changes after it is done.

A task may include delay expressions or nonblocking assignments, although delayed nonblocking assignments probably will not be synthesizable. A function may contain only undelayed blocking assignments.

Timing. Functions can not include delays. A task can include scheduling delays and event controls; a function can not. Because tasks can be executed over some arbitrary simulation time period, they can be used to represent the concurrency typical of hardware components.

Functions are just complicated substitutes for expressions; functions execute in zero simulation time and return a value, very much like a simple expression.

8.1.1.1 Task and Function Declarations

A ***task*** is declared with a port list much like a module header; its declaration begins with the keyword `task`, and it ends with the keyword `endtask`. Here is an example of a task declaration and call:

```
task SwizzleIt (output[3:0] SwizOut, input[3:0] SwizIn, input Ena);
  begin
  if (Ena==1'b1)
       #7 SwizOut = { SwizIn[2], SwizIn[3], SwizIn[0], SwizIn[1] };
  else #5 SwizOut = SwizIn;
  end
endtask
...
always@(posedge GetData)
 SwizzleIt( Bus2, Bus1, SwizzleCmd );
```

A ***function*** is effectively the name of a temporary `reg` type that expresses the value returned by the function. For this reason, a function is declared as an object with a specific width, and with inputs only. Even though a function is called with inputs only, and can not have `output` or `inout` identifiers declared for it, its input(s) must be declared with the keyword `input`. A function may be called in a continuous assignment statement, although this is done rarely in practice. A function may call timing-independent system tasks or functions, such as `$display`.

When called, a task or function must be passed real params (arguments) by position, only; port-mapping by name is not allowed.

A function is declared between the keywords `function` and `endfunction`. For example, here is a function declaration and call:

```
reg[7:0] CheckSum;
function[7:0] doCheckSum ( input[63:0] DataArray );
  reg[15:0] temp1, temp2; // Just to illustrate local declarations.
  begin
  temp1 = DataArray[15:0] ^ DataArray[31:16];
  temp2 = DataArray[63:48] ^ DataArray[47:32];
  doCheckSum = temp1[7:0] + temp2[7:0] ^ temp1[15:8] + temp2[15:8];
  end
endfunction
...
#2 CheckSum = doCheckSum(Dbus);
```

8.1.1.2 Task Data Sharing

Because tasks affect declared data objects as they run, a possible problem arises when two calls are made to the same task during the same simulation time period: Both task calls operate on the same data (the task is declared only once; both executions operate on the same, unique objects declared in the ***one*** task declaration), so concurrent execution may lead to unpredictable behavior which may be fatal to the hardware being simulated.

To prevent sharing of declared internal data, and to allow recursion, tasks or functions may be declared as **`automatic`**, which means that their data are copied and pushed on the (simulator CPU) stack as the sequence of recursive executions proceeds. The `automatic` declaration prevents sharing of local data among different task or function calls; the rationale becomes the same as that of the handling of local variables when making recursive function calls in *C*.

You may wish to look at the on-disc example in the `Lab11` directory, `Find3Mod.v`, to see how the `automatic` keyword is used. Implementation of `automatic` is not consistent across all EDA tools, so its use should be avoided if portability is important.

8.1.1.3 Tasks and Functions are Named Blocks

Finally, tasks and functions are considered named blocks, and a task can be disabled by name from within itself or from anywhere that it can be called. It also may contain named blocks. A function can't be disabled except from within itself, because it executes in zero time; however, named blocks in a function can be disabled by statements within the function.

8.1.2 A Function for Synthesizable PLL Synchronization

We outlined a synthesizable rewrite of the `FindPattern` module in the previous chapter; the idea was to do one giant, 32-expression `if` to check format in a 64-bit data stream. However, an equally valid way would be to run four function calls, each checking just 8 bits at a time.

For example, below is a function, `checkPad()`, which uses a `for` loop to do such a check. A function executes procedurally in zero simulation time, so it can be written in software style, with no worry about simulator-synthesis scheduling inconsistencies. No delay is allowed, of course, but we already assumed that our synthesizable version of `FindPattern` would require no simulation-object update.

```
function   // 64-bit vector      8-bit pad pattern   offset in the stream
checkPad ( input[VecHi:0] Stream, input[PadHi:0] pad, input[AdrHi:0] iX );
 reg OK;       // Flags pattern match.
 integer i     // i is the data stream offset.
       , j;    // j is the pad-byte pattern offset.
 begin
 i = iX;       // Init to stream MSB to be searched.
 OK = 0;       // Init to failed state.
 begin : For   // Capitalized, this is not a verilog keyword.
 for (j=PadHi; j>=0; j = j-1)
   begin
   if (Stream[i]==pad[j])
        OK = 1;
   else begin
        OK = 0;
        disable For; // Break the for loop.
        end
   i = i - 1;
   end // for loop.
end // For.
checkPad = OK;
end
endfunction
```

Notice the "`disable For;`" statement, which stops all activity inside the named block, requiring that the final line, "`checkPad = OK;`" be executed next. This last ends execution, because a verilog function returns as soon as it is assigned a value.

Two comments:

(*a*) the index variables are declared `integer`, because the `for` loop exit variable must be used in a signed comparison: To exit the `for` loop, a value less than `0` must be expressed. The synthesizer's optimization routines will remove unused bits from any 32-bit `integer`, leaving only a register big enough to do the job.

(*b*) The `for` loop is put inside a block named, perhaps humorously, "`For`"; this is legal, because all verilog keywords are lower-case, and verilog is a case-sensitive language. The block is named to allow `disable` to trigger early exit (during simulation) on a mismatch.

If we assume pad-byte patterns declared as local (permanent) parameters,

```
...
localparam[PadHi:0] pad_00 = 8'b000_00_000;
localparam[PadHi:0] pad_01 = 8'b000_01_000;
localparam[PadHi:0] pad_10 = 8'b000_10_000;
localparam[PadHi:0] pad_11 = 8'b000_11_000;
...
```

then, the function `checkPad()` may be called this way:

```
...
if (     checkPad(Stream, pad_11, OffSet-(1*PadWid))
      && checkPad(Stream, pad_10, OffSet-(3*PadWid))
      && checkPad(Stream, pad_01, OffSet-(5*PadWid))
      && checkPad(Stream, pad_00, OffSet-(7*PadWid))
     )
      FoundPads = 1;
else  FoundPads = 0;
```

This is much more readable (and reusable) than 32 equality comparisons all globbed into one `if` expression.

See `FindPattern.v` in the `Lab11` directory for a full implementation of the function calls above.

8.1.3 Concurrency by `fork-join`

We have looked at isolated, concurrent, independent execution by different `always` or `initial` blocks, and we have looked at procedural execution, line by line, within sequential blocks. We have seen how procedural evaluation of delayed blocking assignments differs from concurrent evaluation of delayed nonblocking assignments. Verilog also provides a construct which allows controlled concurrency between statements. This is called the *parallel block*, or `fork-join`, construct. It is modelled after the *fork* system call in the unix operating system.

A `fork-join` currently is *not* synthesizable, but it is used in simulation. Because IP blocks are not normally synthesized, the `fork-join` may be used represent hardware in a simulation model of an IP block to be included in an otherwise synthesizable netlist.

A `fork-join` block is allowed wherever a procedural block is allowed, and it may contain any number of statements which are executed concurrently by the simulator, with the usual unknown, effectively random, order of initialization of each such statement relative to the others. What is special about the `fork-join` block is that it provides for a waiting point at which all its concurrent statements are allowed to catch up with one another and resynchronize.

The statements following the `fork` are effectively independent threads of execution; they are gathered together to a single time point, indicated by `join`, when the last of them finishes its execution. The gathering applies to the next sequential statement following `join`.

For example, in the following code, `DataBus` will see several glitches during simulation. If the delay times shown can't be modified to fix this, the glitches will have to be tolerated:

```
always@(posedge Clk)
  begin
  #1 DataBus[0] <= 1'b0;
  #2 DataBus[1] <= 1'b1;
  #4 DataBus[3]  = 1'b0;
  #3 DataBus[2] <= 1'b1;
  OutBus = DataBus;   // Some OutBus bits change before others.
  end
```

In verilog, the delayed nonblocking assignments should be concurrent, but they may not simulate correctly in some simulators. Regardless, by collecting the result inside a `fork-join` block, the glitches can be prevented:

```
always@(posedge Clk)
  begin
  fork
  #1 DataBus[0] <= 1'b0; // These could be blocking assignments; they
  #2 DataBus[1] <= 1'b1; // still would be scheduled concurrently.
  #4 DataBus[3]  = 1'b0;
  #3 DataBus[2] <= 1'b1;
  join
  OutBus = DataBus;      // All OutBus bits change together.
  end
```

When you want several assignments to occur simultaneously during simulation, and you have to use procedural delays, consider using a `fork-join` block to accomplish this. Otherwise, avoid this construct.

8.1.4 Verilog State Machines

There are two main kinds of state machine, Mealy and Moore. In a Moore machine, the outputs depend solely on the current state; in a Mealy machine, the current state has some control over outputs, but additional control comes directly from inputs and may be independent of the state. For example, a state machine might cycle through various conditions, but a separate device might not always be ready to accept the state machine's outputs; if so, the other device may put some or all of the state machine's outputs into a high impedance state or may switch the outputs using a multiplexer; this machine then would be a Mealy machine. We shall not consider

this distinction further here, because the essence of the state machine is in how it changes state, not in how its output logic is controlled.

A state machine consists minimally of a state register which changes value over time in a deterministic way. A simple state machine is a toggle flip-flop, such as the one we used in our ripple counter lab exercise. If the current state is *set* (`'1'`), then on the next clock the state changes to *clear* (`'0'`); if the current state is *clear*, then on the next clock the state changes to *set*. Any digital counter is a simple state machine; but, usually, state machines are much more complicated. To describe complicated functionality, a bubble diagram or a flow chart is used. Such diagrams may be found in abundance in the data book of a modern microprocessor.

Good verilog design of a simple state machine is to separate the state register control from most of, or all, the combinational logic in the machine. This separation is not required by the language, but it simplifies understanding and maintenance of the state machine, as well as (usually) reducing the amount of coding required. See Fig. 8.1.

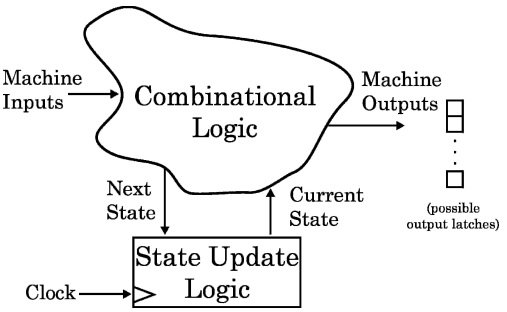


Fig. 8.1 Abstraction of a verilog state machine. Separation of the functionality allows blocking assignments to be used throughout the combinational code

The state update logic includes the state register and is not accessible to external devices. The combinational logic uses inputs and the current state to (*a*) assign outputs and (*b*) determine the next value of the state register.

8.1.5 FIFO Functionality

FIFO is an acronym for *F*irst-*I*n, *F*irst *O*ut: The first value written into a FIFO is the first one to be read out, somewhat like a pipeline in which the fluid sent in first is the first to be delivered.

A FIFO is a kind of register or memory stack. A stack structure is typified by storage which is addressed by numbers calculated from recent numbers, as opposed to being calculated independently as some arbitrary pointer or address value. It is then very reasonable that the complement of a FIFO should be a LIFO, *L*ast-*I*n, *F*irst *O*ut. A LIFO is like a standard microprocessor stack: The last value pushed onto the stack is the first one popped off of it.

A LIFO can be controlled by a single register, a stack pointer, because the location used to push data into a LIFO is the same as the location from which data would be popped. However, a FIFO is more complicated, because, like a pipeline, it has two ends to be controlled. Whereas a LIFO can have just one stack pointer, a FIFO has to have two different pointers, one to write data in, and the other to read data out.

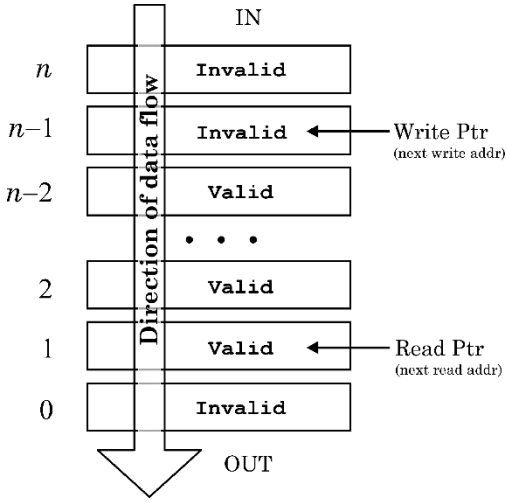


Fig. 8.2 First-in, first-out functionality gives a FIFO register file a specific directionality

As shown in Fig. 8.2, the FIFO storage consists of some number n of registers, indicated by horizontal rectangles, and a predefined direction of data flow. Data are written in at some rate; they are read out at some perhaps different rate. One application of a FIFO is as a buffer between two different clock domains on a chip; bursts of reading or writing are cushioned by the FIFO so that devices in both domains can move data at some useful rate without waiting on every clock tick for the other domain to become ready. Other FIFO applications are in RS-422 or ethernet serial links, both of which usually communicate between independent clock domains.

There are two main conventions in representing FIFO read and write pointers: (*a*) These pointers can be viewed as pointing to the next valid address (register) in the FIFO; or, (*b*) they can be viewed as pointing to the most recently used one. In this course, we shall adopt the former convention: A pointer points to the ***next*** address.

Our read pointer, as shown in Fig. 8.2, then must point to valid data in the FIFO, the next datum to be read; the write pointer must point to the unused register into which the next new datum will be written. Thus, the write pointer points to currently

invalid data, generally data which have been copied (read) out, so that that register's contents are of no further value.

If the data flow is as shown in Fig. 8.2, from top to bottom, it must be that the two pointers move upward after each use. Now, what happens when the write pointer shown reaches register *n*, at the "top" of the FIFO? Simple: It wraps around and positions itself at the bottom register shown, if that register's contents are invalid. We have seen the same thing when one of our unsigned `reg` counters wraps around from its maximum value to 0 again; and, in fact, unsigned counters can be used to control FIFO read and write pointers.

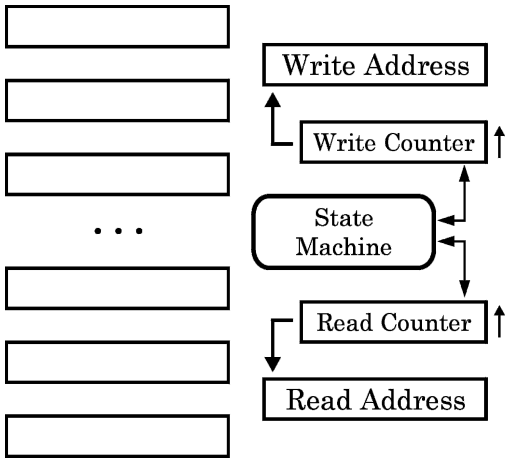


Fig. 8.3 FIFO component parts

The component parts of a FIFO are as shown broken down in Fig. 8.3. There is a set of data storage registers; a read-address and write-address pointer, and an unsigned counter to control each pointer value. The state machine enables or disables counting and monitors the current count values.

Address translation for the FIFO register file could be used to map the counts to and from Gray code addresses. Thus, the register addresses can be encoded values rather than simple, sequential binary counts. The small vertical arrow on the right of each counter indicates that it is allowed to count just one way, here depicted as "up". If the counters could count both the way shown and the opposite way, which is to say both "up" and "down", there would not be any well-defined direction of data flow in the FIFO, and it would not be a FIFO. By encapsulating count-to-address translation in verilog ***task***s, the translation rationale can be changed arbitrarily without need to alter anything else in the FIFO implementation.

8.1.6 FIFO Operational Details

Before attempting to write verilog describing a FIFO, let us look more closely at how it must operate. First, suppose the FIFO was "empty", which is to say, that all data had been read out of it and no new data had been written since. How to describe this depends on where the write pointer was when the read pointer was used to read out the last valid datum.

Suppose, for example, the write pointer was at the second register from the top in Fig. 8.3. Then, the FIFO would look something like that in Fig. 8.4, labelled $Empty_1$.

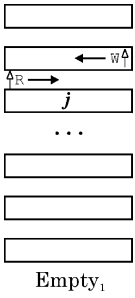


Fig. 8.4 Register addressing for an empty FIFO

The horizontal arrow sits on the register at which the pointer is pointing; the small vertical arrow indicates the only allowed direction of change. The write pointer, labelled with a "W", is free to move upward; the read pointer ("R") has just read from register j and is not allowed to move upward to $j+1$, because those data are invalid, as explicitly indicated by the position of W. Thus, we must depict R as itself being invalid, pointing nowhere useful.

In the situation just described, suppose, now, that one new data value was written. The FIFO no longer would be empty. This datum would be written to register $j+1$, and W would move to point to the next invalid register, $j+2$. We are now allowed to read from $j+1$, so R should be pointing there. This is shown in Fig. 8.5, labelled $Empty_1 + W$.

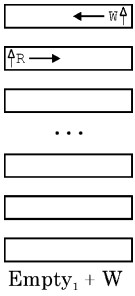


Fig. 8.5 Register addressing for an almost-empty FIFO

Let's look at a different, but equally "empty" condition. Suppose the last valid datum had been read while the write pointer was pointing to the bottom register ($n = 0$, in Fig. 8.2). Then, the read pointer must be invalid and again must point nowhere. We may indicate this by putting R right below W, because when W does a write, R will be ready to read from that exact, same register. Therefore, this second empty FIFO may be depicted as shown in Fig. 8.6 again to the left, labelled $Empty_2$.

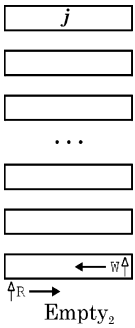


Fig. 8.6 Register addressing for an empty FIFO

In this case, when the next write occurs, making the FIFO no longer empty, W will move from 0 to point to register 1, and R will become valid, pointing to register 0. This is shown in Fig. 8.7, labelled $Empty_2 + W$.

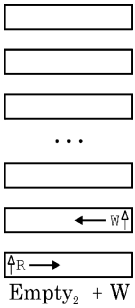


Fig. 8.7 Register addressing for an almost-empty FIFO

Very good. We may be able to guess from this that R must be less than W at all times, because R never can move upward to point to the same register as W; therefore, R never can cross over W and get above it.

However, now let us look at the opposite condition of the FIFO, when it is "full". This means that not enough has been read recently, and that all possible registers have been written, so that no more writes are allowed, and therefore the W pointer must be invalid.

Suppose this condition occurred just after W had written its last datum into some register j, as shown in Fig. 8.8, labelled $Full_1$.

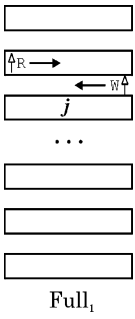


Fig. 8.8 Register addressing for a full FIFO

Oops! We guessed wrong. As easily seen, it is R which is now just ahead of the next possible position of W. W is not allowed to move up past R, because writing to the valid register $j+1$, pointed to by R, is forbidden.

If a read now takes place, the FIFO no longer will be full, register $j+1$ will become invalid, and the condition will change to the one shown in Fig. 8.9, labelled $Full_1$-R.

Fig. 8.9 Register addressing for an almost-full FIFO

After the read, R points to $j+2$, and W is allowed to point to $j+1$, where a write now can occur.

Finally, let's look at another full condition, where R is at the bottom of the FIFO, at register 0. If so, it must be that W is invalid but will point to register 0 as soon as a read has allowed the FIFO to contain an invalid register. The full condition is shown on the left, in Fig. 8.10, labelled $Full_2$.

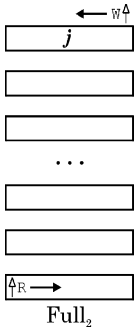
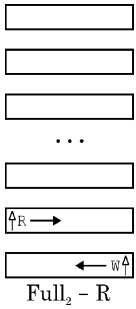


Fig. 8.10 Register addressing for a full FIFO

The position of W in $Full_2$ actually is nowhere, because W is invalid; but, because we know its first possible valid location, it is shown ready to wrap around from the other end of the register set to register 0; all that this wrap requires is one read.

After that one read, the FIFO no longer is full and the situation is shown in Fig. 8.11, labelled $Full_2$-*R*.

Fig. 8.11 Register addressing for an almost-full FIFO



We see that R now points to register 1, and that W has become valid and points to register 0, at the bottom of the FIFO as shown.

The conditions of "full" and "empty" are of great concern to the other devices depending on the FIFO for data transfer. A full FIFO means that the supplier of data must stop trying to send data; an empty FIFO means that the consumer of data must stop trying to receive data. Thus, a FIFO in general must provide output flags that indicate when it is full or empty.

8.1.7 A Verilog FIFO

Our FIFO will be composed of a memory (register file) and a control module. The control module will be designed as a state machine. As explained above, we shall separate the state machine sequential transition logic from the combinational logic of the rest of the state machine. A schematic of the particular design we shall use is given in Fig. 8.12.

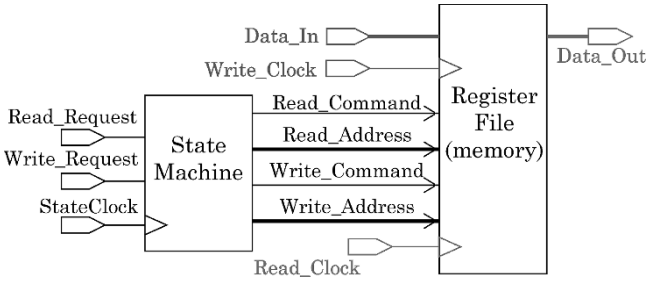


Fig. 8.12 Block diagram of FIFO

In the following, we shall refer to "read" and "write" *commands* by the state machine. These terms refer to commands issued by the state machine to the RAM, at the read or write address being output by the state machine.

Keep in mind that there will be an external device which is using the FIFO and which will be sending read and write *requests* to the FIFO; the state machine will honor these requests by issuing commands to the RAM. However, when the FIFO is full, a write request will have to be ignored; when the FIFO is empty, a read request will have to be ignored.

The problem in the detailed description of the FIFO above is in the read and write pointers when they are not usable; one of them is unusable in the "empty" condition, the other in the "full" condition. At face value, we can't write verilog to handle a pointer which is pointing nowhere. On the other hand, study of the preceding details reveals that the `R` and `W` pointers, as well as their next transitions, always are well defined numerically if we stay just one step away from the FIFO states of empty or full.

So, we shall design around "almost-empty" and "almost-full" and shall treat "full" and "empty" as special cases which include invalid pointers.

For brevity, let `R` and `W` indicate the numerical positions *n* of the FIFO registers to which they point. Then, when `R == W-1`, the FIFO is one read away from empty; and, when `W == R-1`, the FIFO is one write away from full. These are easily described arithmetic relations. The condition `R==W` is, of course, operationally forbidden for ***two*** valid pointers; one must be invalid. All other values of `R` and `W` allow simple arithmetic describing correct operation with no special concern: The pointer simply is incremented by `1` each time, after it is used.

Given this, we may describe our FIFO as a state machine with a possible state transition only on a read or write and with the following five simple states:

`normal`	Normal. Read may transit to `almost_empty`; write may transit to `almost_full`. No other transition allowed. A 4-register FIFO will move directly between `a_empty` and `a_full`, with no `normal` state.
`a_empty`	Almost empty. Read transits to `empty`; write transits to `normal`.
`a_full`	Almost full. Write transits to `full`; read transits to `normal`.
`empty`	Empty. Read is forbidden; write transits to `a_empty`.
`full`	Full. Write is forbidden; read transits to `a_full`.

A state transition diagram ("bubble diagram") for this machine would be as shown in Fig. 8.13.

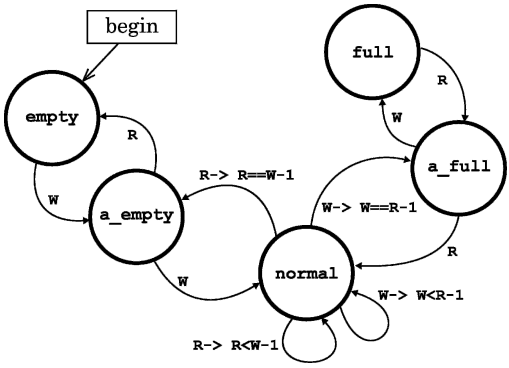


Fig. 8.13 FIFO state machine transition diagram. "`begin`" is the power-up or reset transition. Assumes a FIFO with five or more storage registers and `R` and `W` the address of next action

We'll start by defining our state register and its states. For 5 states, the major alternatives would be a 3-bit binary register or a 5-bit one-hot register. Let's use a binary encoding.

The machine must retain state, so we require a block of clocked sequential logic for state transitions. Applying our previous rules for coding, we want nonblocking assignments separated from blocking ones, so we shall isolate state transitions in a small, clocked block with nonblocking assignments only. Combinational logic will determine the next state from the current one, so we need only pass the sequential block a variable with the next state encoded in it. The transition logic can be blocking assignments. We shall just code the state assignments in the state machine module; the FIFO registers themselves will be in a separate module.

We need two other sequential elements, the read and write counters. They have to be clocked so that state changes take place only while the FIFO pointers are in a known, consistent state. We'll therefore only allow the combinational logic to be read on the opposite clock edge from the one which updates the address counters and the state register. If we update the state register on the positive edge of the clock, this means that we should update the address counters on the same edge but read the combinational block, which programs all updates, only while the clock is low. There are several ways of implementing the counter updates, but a task called in the combinational block perhaps is the simplest:

```
task incrRead; // Called while clock is low.
  begin
  @(posedge Clk)
    ReadCount = ReadCount + 1;
  end
endtask
```

When this task is called, it stops at the event control and waits on a positive clock edge; when that edge occurs, it increments the read counter address and then exits. A similar task may be declared for the write counter.

Using tasks to increment the counters, and perhaps also to compare values, would make modifying the counter easier and less error-prone during early development than having to go around and edit isolated stretches of combinational code, if we decided to change to a one-hot or gray-code address counter instead of a binary counter.

Turning to the state encoding and ignoring state transition logic, so far we have this:

```
//
// Don't allow any other module to affect the state encoding,
// so, use localparams:
//
localparam  empty = 3'b000, // all 0 = empty.
          a_empty = 3'b010, // LSB 0 = close to empty.
           normal = 3'b011, // a_empty < normal < a_full.
           a_full = 3'b101, // MSB 1 = close to full.
             full = 3'b111; // all 1 = full.
//
// The sequential block controlling state transitions:
//
always@(posedge Clk, posedge Reset)
  if (Reset==1'b1)
       CurState <= empty;
  else CurState <= NextState;
// End sequential state transition block.
```

We cannot reset `NextState` in the clocked block, because the synthesizer would object to the contention of the clocked assignment to `NextState` in the sequential logic combined with the inevitable other unclocked assignments in the combinational logic; so, we shall have to plan for a reset of `NextState` in the combinational logic. This synthesizer behavior is a good thing in general; it helps prevent race conditions.

We have to process requests to read and write, so our combinational block for controlling state transitions should be sensitive both to state changes and to these requests.

A first cut at the combinational block might be as follows, with variables named "*xxx*`Reg`" being the `reg` used for procedural assignment to output wire *xxx*:

```
always@(ReadReq, WriteReq, CurState, Clk) // NOTE: Request, not Register.
if (Clk==1'b0) // Only read after a negedge of clock.
  begin
  case (CurState)
    empty: // Combines reset unique conditions
           // with a simple empty state during operation:
           begin
           if (Reset==1'b1)
             begin
             // Reset conditions:
             FullFIFOReg = 1'b0; // Clear full flag.
             WriteCount  = 'b0;
             WriteCmdReg = 1'b0;
             ReadCmdReg  = 1'b0;
             NextState   = empty;
          end
          //
          // Generic empty conditions:
          EmptyFIFOReg = 1'b1; // Set empty flag.
          ReadCmdReg   = 1'b0; // Disable RAM read.
          // One transition rule:
          if (WriteReq==1'b1 && ReadReq==1'b0)
            begin
            ReadCount = WriteCount; // Could also init to Adr 0.
            incrWrite; // Call task, which blocks on posedge Clk.
            WriteCmdReg = 1'b1; // Issue a RAM write.
            EmptyFIFOReg = 1'b0; // Clear empty flag.
            NextState = a_empty;
            end
          else ReadCount = 'bz; // Nowhere.
          end // empty state.
 a_empty: begin
          ...
          end // a_empty state.
  normal: begin
          ...
          end // normal state.
  a_full: begin
          ...
          end // a_full state.
    full: begin
          ...
          end // full state.
 default: NextState = empty; // Always handle the unexpected!
 endcase
 end // always.
```

A problem which might be overlooked here is that `WriteCount` is being read in this block, but that it is not on the sensitivity list, risking creation of an unwanted latch during logic synthesis. To avoid this, in our next approximation, we shall change `always@(ReadReq, WriteReq, CurState, Clk)` to `always@(*)` to avoid any possible future unpleasant surprise.

Let's consider one more state and set up transition rules for it. The `normal` state seems a good candidate. We proceed as follows:

Transitions in our state machine only occur on a read or write. When a clock occurs, but the FIFO performs neither read nor write, it is possible to consider the

machine in an "idle" state. For some state machine problems, implementation of an explicit idle state is useful; but, for ours, it is unnecessary, because we don't consider occurrence of a clock to be an event relevant to the machine state. We don't care about anything but read or write. The two clock edges just provide orderly execution as well as adequate time for logic to settle.

In the `normal` state, on any read or write, we only have to check to see whether the next state should be `a_empty` or `a_full`. There is no way in our design that we could go directly from `normal` to `empty` or `full`, barring a machine reset. We also know that

$$(\textit{write counter value}) > (\textit{read counter value}) + 1$$

and

$$(\textit{read counter value}) > (\textit{write counter value}) + 1$$

both are required to be true to remain in the `normal` state.

So, the simplest way to look for a transition out of `normal` is to add another `1` to every new counter value and compare with the current value in the other counter, checking for an equality instead of a greater-than. If an equality has occurred, then the machine must exit the `normal` state.

Taking all this into consideration, the combinational block `case` alternative for `normal` should be something like the following:

```
normal: begin
        // On a write:
        if ( {WriteReq,ReadReq}==2'b10 ) // Concatenation.
          begin
          ReadCmdReg  = 1'b0;   // Disable RAM read.
          WriteCmdReg = 1'b1;   // Issue a RAM write command.
          incrWrite; // Call task, which blocks on posedge Clk.
          //
          // Transition rule: Check for a_full:
          if ( ReadCount == WriteCount+1 )
               NextState = a_full;
          else NextState = normal;
          end
        // On a read:
        if ( {WriteReq,ReadReq}==2'b01 )
          begin
          WriteCmdReg = 1'b0;  // Disable RAM write.
          ReadCmdReg  = 1'b1;  // Issue a RAM read.
          incrRead; // Call task, which blocks on posedge Clk.
          //
          // Transition rule: Check for a_empty:
          if ( ReadCount+1 == WriteCount )
               NextState = a_empty;
          else NextState = normal;
          end
        end // normal state.
```

To reduce the complexity of comparing 2 bits, we concatenate them into a single, 2-bit expression. The read `if` statement might have been put into an `else`, making

it the alternative to write; for now, the `else` has been omitted because it seemed of limited value and would have added lines of code for the designer to read and understand.

Keep in mind that in a context in which the request bits might be reversed by an external device while the write operation was being processed, the independent `if`'s (write and then read) in the `normal` state code above would be seen as a possible race-condition hazard. If so, an `if-else` or a nested `case` definitely would be a good idea.

Notice that the state transition is assigned last; this is to ensure that all operations to be completed in the current state are scheduled before the state can be changed.

Finally, notice that in the code above we are comparing *some*`Counter Value`+1 with *someother*`CounterValue`. The "+1" might be a problem, because the width of the expression becomes ambiguous: Is it the expression of an integer ('`1`') or of a rather small reg ("*some*`Counter`")? When a verilog variable has been assigned, the result takes on the width and type of the destination variable; but, this expression will not be assigned to anything. Integers are *signed* types, and the last thing we would want is for a negative count to be expressed. To avoid potential problems caused by different simulator implementations, in our next approximation of this model, we shall assign the sum to a `reg` of the same width as *some*`Counter` before making the equality comparison.

Delay times are omitted in the code above because blocking assignments guarantee the evaluation sequence, and preceding each statement with, say, "`#1`", would make no obvious difference. Furthermore, synthesis will introduce a variety of unpredictable delay differences in each branch of the logic, and backannotated delays from floorplanning and layout would supercede programmed delays in this model, anyway.

One reason to add delays in the code might be to make the simulation protocol behaviorally accurate with regard to other devices to be connected to the state machine controller; however, such delays far better would be added in the final continuous assignments, as shown in `FIFOStateM_header.v`:

```
module FIFOStateM
       #(parameter AdrHi=4) // 2** (AdrHi+1) registers. Default=32.
        ( output[AdrHi:0] ReadAddr
        , ...
        , Clk, Reset
        );
  //
  reg[AdrHi:0] ReadAddrReg, WriteAddrReg;
  ...
  reg          EmptyFIFOReg, FullFIFOReg
             , ReadCmdReg, WriteCmdReg;
  //
  assign #1 ReadAddr  = ReadAddrReg;
  assign #1 WriteAddr = WriteAddrReg;
  assign #1 EmptyFIFO = EmptyFIFOReg;
  assign #1 FullFIFO  = FullFIFOReg;
  assign #1 ReadCmd   = ReadCmdReg;
  assign #1 WriteCmd  = WriteCmdReg;
...
```

A possible reason to add delays in the individual statements would be to spread the simulator waveform display to displace edges and perhaps make the sequence of events easier to see; however, such modifications then would have to be marked somehow in the simulator so that they could not be forgotten and become part of the design. Such delays might be justified occasionally; but, generally, the only delays in a design should be those near the top of the design file, in continuous assignment statements to the module outputs.

8.2 FIFO Lab 11

Do all work for this lab in the `Lab11` directory.

Lab Procedure

Step 1. Use of a task in the Lab 10 design. Copy the file, `FindPatternBeh.v`, from the `Lab10` directory to `Lab11`, renaming it `FindPatternTask.v`. As usual, rename the module so it corresponds to the file name. In this verilog, there is a repeated stretch of code:

```
#1 i = i + IJump*j - 1;
#1 j = 0;
#1 Nkeeper = 'b0;
```

Clearly, the two identical occurrences are supposed to do exactly the same thing; and, if it became necessary to change one, the other should be changed the same way. This is an ideal situation in which a task or function should be used. Time delays are involved, so it has to be a task.

Enclose the code above in a `task`, and call the task at the two places where the code above appears. Briefly run a simulation to verify correctness.

Step 2. An assertion task.

Use the `ErrHandle` code below as an assertion somewhere in each of the Steps in this lab, including the previous one, to warn the user of some error or dubious condition. Test it for each of the possible actions (see Fig. 8.14). There is an example file containing this task in your `Lab11` directory:

The `Sts` type is 4 bits, which allows for 16 different status conditions. Because a `reg` is unsigned, this also means that `4'b1000` or "above" will be used to represent negative status values passed to the task. There are four possible `Action` values:

- `0:` The `Sts` is `0` or the `Msg` is null. This condition is normal, and the task silently returns, with no message.
- `1:` The `Sts` is `-1 (4'hf)`. The condition is a fatal error and the simulation is finished (`$finish`).
- `2:` The `Sts` is more negative than `-1 (4'hf>Sts>=4'h8)`. The condition is an error and the simulation is stopped (`$stop`) but may be continued.
- `3:` The `Sts` is positive (`4'h1--4'h7`). The condition is a warning or an informative one, the `Msg` is printed to the simulator console, but no other action is taken.

```
task ErrHandle(input[3:0] Sts, input[255:0] Msg);
reg[1:0] Action;
  begin
  if (Sts==4'h0 || Msg=='b0)
                      Action = 2'b00; // == 0.
  else if (Sts==4'hf) Action = 2'b01; // Sts == -1, 2's complement.
  else if (Sts>=4'h8) Action = 2'b10; // Sts <  -1, 2's complement.
  else                Action = 2'b11; // Sts > 0.
  //
  case (Action)
    2'b00: Sts = 0; // Do nothing.
    2'b01: begin
           $display("time=%4d: FATAL ERROR. %s"
                   , $time,                 Msg);
           $finish;
           end
    2'b10: begin
           $display("time=%4d: ERROR Sts=%02d. %s"
                   , $time,           Sts,     Msg);
           $display("\nYou may continue the simulation now.");
           $stop;
           end
  default: $display("time=%4d: Sts=%02d. NOTE: %s"
                   , $time,     Sts,           Msg);
  endcase
  end
 endtask
```

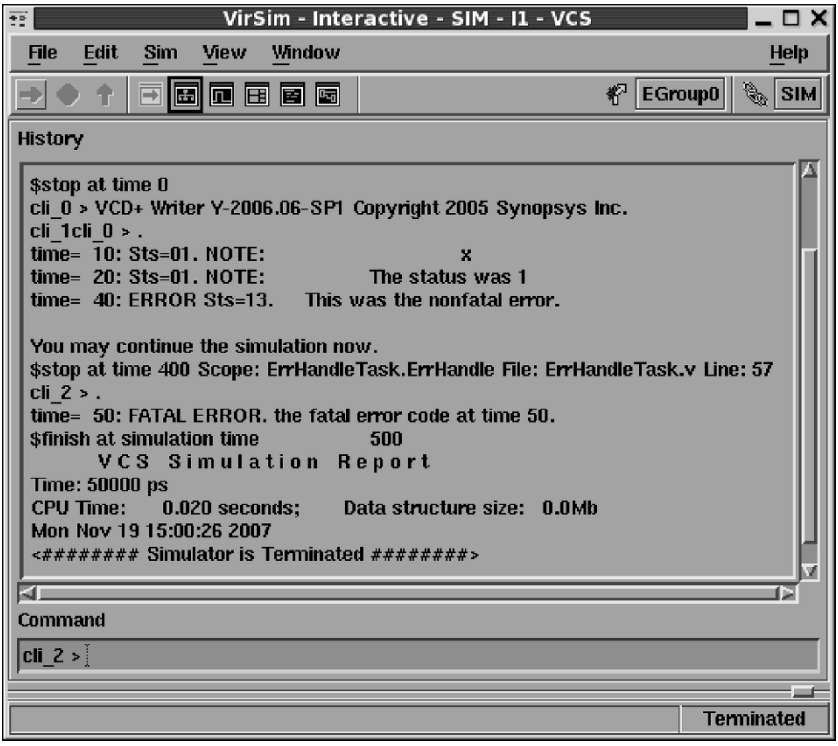


Fig. 8.14 Test simulation of `ErrHandleTask`

Of course, there is no effect on simulation waveforms; however, the messages asserted will appear in the simulator console window, as shown in Fig. 8.14.

Step 3. FIFO state machine design. A module header and partial design has been provided in the file, `FIFOStateM_header.v`. A sketch of a testbench also is included. Copy this file to a new one named `FIFOStateM.v`, and complete the design as follows:

A. Change the combinational `always` block's sensitivity list so it will be sensitive during simulation to any change in a variable which is read within that block.
B. Complete the assignments and transitions for the remaining states. Simulate the machine to check correct address generation. Use `for` loops in your testbench to write the FIFO full, then read it empty, to verify correct address, state register, and flag operation.
C. Synthesize the design, optimizing first for area and then for speed.

Step 4. Attach a register file to the FIFO state machine, making a complete, functional FIFO. We shall improve the memory and controller functionality in a later lab. For now, proceed as follows:

Copy your `Mem1kx32.v` static RAM design from the `Lab07` directory into the `Lab11` directory. Create a new verilog module named `FIFO_Top` in its own new file in the `Lab11` directory. In `FIFO_Top.v`, instantiate your `FIFOStateM` and `Mem1kx32` and connect them together. Supply top-level input and output data busses to read and write to the FIFO. You will have to combine the state machine's separate read and write address busses to provide a single address for the memory; use a top-level continuous assignment statement and a conditional operator to do this.

After simulating enough to satisfy yourself that your FIFO is working (see Fig. 8.15 and 8.16), synthesize the FIFO in random logic and optimize first for area and then for speed.

Step 5. If time permits, and if you have not already done so, double check your state machine combinational block for possible race conditions as discussed in the lecture notes above. Race conditions can be avoided by using a single statement to check at one point in simulation time, in each state, to decide what the requested operation (or state transition) shall be.

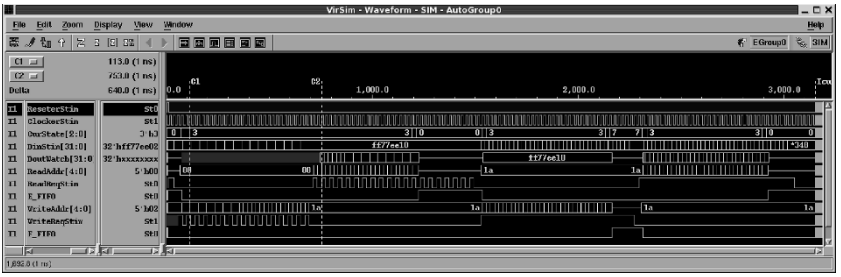


Fig. 8.15 First-cut FIFO simulation – limping, but with obvious race conditions avoided

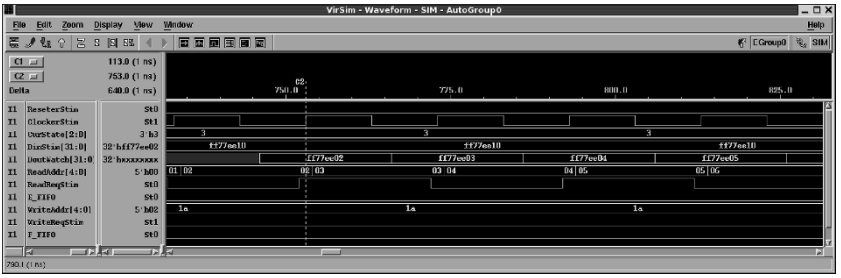


Fig. 8.16 Close-up of the FIFO simulation, showing the first read-out of the register file

8.2.1 Lab Postmortem

What is the relationship of FIFO design across clock domains and the use of Gray code counters?

8.2.2 Additional Study

Read Thomas and Moorby (2002) section 3.5 on functions and tasks.
Read Thomas and Moorby (2002) section 4.9 on `fork-join`.
Read and understand the simple memory model in the Thomas and Moorby (2002) section 6.3, exercise 6.9.

(Optional) Thomas and Moorby (2002) contains several different perspectives on state-machine modelling. If you would like to know more about this, try (re)reading sections 1.3.1, 2.6–2.7, chapter 7, and appendices A.14–A.17. See the optional readings below, too.

Optional Readings in Palnitkar (2003)

Read Chapter 8 on tasks and functions.

The code in Appendix F.1 represents a synthesizable FIFO. The use of counters to track the FIFO state is important; some of our References discuss gray-code counters and other FIFO subtleties. Notice that Palnitkar's appendix F FIFO has only four storage registers.

There are state machine models in sections 7.9.3 and 14.7. Look through these to see how the control is implemented.

Chapter 9
Week 5 Class 1

9.1 Rise-Fall Delays and Event Scheduling

This time we shall look into delays, timing, and the way they are used in verilog statements.

9.1.1 Types of Delay Expression

We'll deal with path delays, component internal delays, and switch-level delays later in the course. For now, we shall concentrate only on gate-level (gate-to-gate) and procedural delays.

Regular vs. Scheduled Delay. Two kinds of delay expression are *regular* and *scheduled*. The scheduled kind is called "intra-assignment" in the Thomas and Moorby (2002), section 4.7. Although the Thomas and Moorby (2002) authors seem to find scheduled delays useful, for example in section 8.3.1, in practice they rarely are so. A regular delay appears to the left of the target of a statement; a scheduled delay appears to the right of the assignment operator ('=' or '<='). The possibilities are,

```
#(delay) variable1  =  value;              // 1. regular blocking.
#(delay) variable2 <=  value;              // 2. regular nonblocking.
variable3  = #(delay)  value;              // 3. scheduled blocking.
variable4 <= #(delay)  value;              // 4. scheduled nonblocking.
#(delay) variable5  = #(delay)  value;     // 5. both, blocking.
#(delay) variable6 <= #(delay)  value;     // 6. both, nonblocking.
```

The *regular delay* statement delays the statement until the specified delay value of time has lapsed; then, the RHS expression is evaluated, and the statement is executed.

All nonblocking delay statements are flagged as errors or with serious warnings by the synthesizer. Blocking delays are synthesized with the delay values ignored. As we have mentioned before, the synthesizer can't synthesize delay values,

especially those scheduling concurrent, future events, and delayed nonblocking assignments might be meant to be executed in any arbitrary order, from the synthesizer's perspective.

The *scheduled delay* statement causes immediate evaluation of the RHS of the statement and delays assignment of the result for when the specified value; then, the statement is executed. This is equivalent to a VHDL simulator's *transport* delay scheduling mode.

The "both" delay statement above combines regular and scheduled delays. We shall examine it a little more in lab.

Terminology Difference:
Thomas and Moorby (2002) introduces the term *regular event* in section 8.4.3; this is meant to refer to something different from our *regular delay*. The Thomas and Moorby (2002) *regular event* is what this book and the IEEE Std 1364 would call an *active event*, as explained below.

It is strongly discouraged to use the scheduled delay for anything. In effect, this is an analogue, active-element RC delay-line delay which almost always will be unrealistic in the simulation of on-chip digital logic. Furthermore, this construct in effect creates a new concurrent thread of execution within a procedural block, negating the value of sequence to control functionality. For concurrency, instead use independent `always` blocks or continuous assignment statements. If you are not planning to synthesize the logic, you may use a `fork-join`. However, this should be a last resort. Keep in mind that a `fork-join` can not allow delays to time out independently; rather, the `join` restores sequence only after all `fork`ed statement delays have lapsed.

It might seem reasonable to use scheduled delays for appearances: To separate simultaneous edges in a simulator waveform display for better visibility. However, no simulator known to the author will display scheduled-delay displacements differently from regular-delay displacements; thus, the verification tool may become risky for verification, by confusing modelled delays (setup or hold timing slacks, for example) with scheduled delay displacements which are not part of the design.

For example, suppose a D flip-flop was clocking in data, but the setup delay was not evident (see Fig. 9.1 *A*). A scheduled delay would make this clearly visible, but only if everything went well, and no design problem developed:

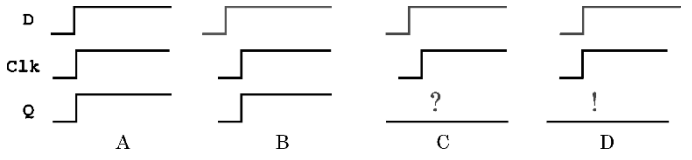


Fig. 9.1 Scheduled delays can cause confusion. *A*, original display; *B*, appearance fixed by scheduled delay on `Clk` and `Q`; *C*, external events cause loss of setup, but ***scheduled*** delay conceals the reason; *D*, same external events as *C*, and omission of scheduled delays exposes the setup loss

As pointed out in Thomas and Moorby (2002) (section 8.4.1), a scheduled delay also may be viewed as though it created a temporary `reg` variable which is invisible and thus protected against other assignment statements: For a delay value ***d***, "`x <= #d y;`" is the same as, "`temp = y; #d x = temp;`" in which `temp` is protected against other assignment during the intervening `#d` delay. This can be wasteful of simulator memory, and it may introduce design bugs where otherwise none would be. The hidden value can't be cancelled by input changes, making realistic inertial scheduling impossible.

Multivalue Wire-Delay Expressions. Even in CMOS technology, in which gates are relatively symmetrical, rise vs. fall, there is some difference in performance between the delay to a '`0`' and the delay to a '`1`'. For example, capacitive interaction with ground or power planes can change timing differently. Also, NMOS transistors can be very slow to pull up to a '`1`', and PMOS to a '`0`'. To model this kind of difference, verilog allows primitive-gate or wire-connecting timing expressions to include two, and sometimes three, values, thus:

```
assign #(3, 5) OutWire_001 = newvalue;
bufif1 #(3,5,7) GateInst_001(OutWire_001, InWire_001, Control_001);
```

The two values are in (*rise*, *fall*) order, and the three values are in (*rise*, *fall*, *to*`z`) order and represent the *rise delay* from '`0`' to '`1`' and the *fall delay* from '`1`' to '`0`'. For assignments, or for components capable of it, the third value represents *high-impedance delay*. We shall study these differences in detail later in the course. They are ***not*** allowed in procedural assignments; a procedural statement only is allowed one delay value.

In summary, a connecting delay expression may have one, two, or three delay values:

`#(`***every_delay***`)`	Simulator uses this for all scheduling.
`#(`***rise_delay***`,`***fall_delay***`)`	Simulator uses *rise_delay* to schedule changes to '`1`'. Simulator uses *fall_delay* to schedule changes to '`0`'.
`#(`***rise_delay***`,`***fall_delay***`,`***z_delay***`)`	Simulator uses *rise_delay* and *fall_delay* as for 2 values. Simulator uses *z_delay* to schedule changes to '`z`'.

Delays *to* '`x`', or two-valued delays *to* '`z`', use the shortest delay in the expression; this is one facet of the principle of "delay pessimism".

Multivalue delay expressions may be used in:

- Continuous assignment statements.
- Primitive component instantiations.

Scheduling Surprises. Here are a few perhaps surprising points to ponder, before we look into the verilog event queue. In the following, keep in mind that many simulators designed for VLSI netlists will not simulate inertial delay properly, when the delays are hand-entered in continuous assignments or procedural blocks:

First, when a list of `#0` nonblocking statements is encountered in a procedural block, the simulator schedules them all at once in the current time, *after* undelayed events, and they can not be depended upon to be run in any well-defined order (order is implementation-dependent):

```
  x <= 1'b1;
  y <= 1'b0;    // z gets the new 1'b1 value of x.
#0 z <= x;
```

Of course, if the designer only writes synthesizable code, this problem never arises.

Second, when a list of *un*delayed statements is encountered, they are guaranteed to be run in the order listed, in an interval of zero time duration. However, nonblocking statements assign the right-hand value read, whereas blocking statements block evaluations, update the left-hand sides in order, and use only the updated values:

```
x <= 1'b1;
y <= 1'b0;    // z gets the old value of x;
z <= x;       // x gets the 1'b1 originally scheduled.
```

Third, only one evaluation can be held scheduled for a future time. Thus, when conflicting assignment statements are read, the last one read determines the delay and value to be scheduled:

```
#2 x <= 1'b1;
#2 x <= 1'b0;    // x will be scheduled for 1'bz
#2 x <= 1'bz;    // at 2 time units from current_time.
```

This is pathological code, and many tools will refuse to simulate it correctly. Again, writing only synthesizable code avoids this whole issue; but, it is good to be aware of it – for example, when writing a testbench.

Now, let us proceed to an understanding of why we should not have been surprised.

9.1.2 Verilog Simulation Event Queue

Specifications for simulation are built into the verilog language, including specification of how events shall be queued for execution and scheduled for future simulation times. From here on, the word *time* will refer only to *simulation time*, unless explicitly stated otherwise.

As described in IEEE Std 1364, the verilog language depends on a ***stratified event queue***. A conforming simulator initially starts with a list of undelayed events,

executes them, sets time to 0, and then proceeds forward in time in a completely deterministic way. However, when two or more events are scheduled for the same time, the language may not specify an order of execution; in other words, all may be found in the same ***stratum***. Thus, when events are simultaneous, different simulator implementations sometimes may produce logically different, but equally correct, results. It is up to the designer to write verilog avoiding such differences, when they matter to the validity of the design. Thus, it is important to understand the stratified event queue.

The verilog stratified event queue is illustrated in Fig. 9.2. Before 2005, the verilog IEEE Std 1364 defined additional, PLI-related strata which now are obsolete.

Keep in mind that an ***event*** is a change in value, caused by an output driver (structurally) or by execution of an assignment statement. Mere evaluation of an input, or of the expression on the right-hand-side of a statement, is not an event.

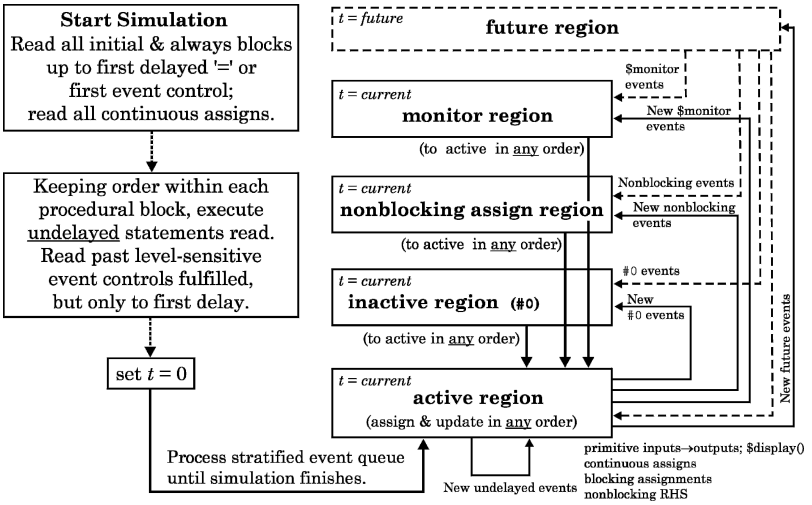


Fig. 9.2 Verilog Event Queue.. The five-region queue proper is on the right, separated from simulator initialization actions. Solid lines represent current-time flow or processing order; dotted lines represent input obtained on advancing the current time to the next (future) time

The solid lines in Fig. 9.2 are meant to represent current-time accesses to data, or the flow of control in the current time. For example, one of the solid lines ("To active in any order") represents access to the *inactive* region of the stratified queue after all active events have been executed (after all new values are assigned to their variables). That solid line means that all *inactive* events found are moved to the *active* region. Any ordering of statements in the verilog source code, or any ordering which occurred while moving events into the *inactive* region, is lost during the transfer from the *inactive* to the *active* region.

It is interesting that, even though `#0` means zero time delay, events scheduled at the current time with `#0` are placed initially in the *inactive* region, until all *active* events have been processed. The *active* events can include undelayed blocking

assignments, changes in verilog primitive inputs resulting in undelayed output changes, or other events which had been scheduled for the current time because of being delayed from some previous time.

Other solid lines in Fig. 9.2 show that the *active* region may trigger new *inactive* events ("New `#0` events"), new nonblocking assignments, and so forth. When all *active* events have been exhausted, and no new event can be created in the current time (by transfer from the *inactive*, *nonblocking assignment*, or *monitor* region, in that order), the time is advanced to that of the next event, in the future, which was read and scheduled. The dotted lines in Fig. 9.2 then represent the flow of new data from the new current time into the stratified queue.

The often-useful `$strobe` system task is executed from the *monitor* stratum, which is after the *nonblocking assign* stratum; this is why `$strobe` can report the *result* of a nonblocking assignment.

9.1.3 Simple Stratified Queue Example

There is an example named `InactiveStratum.v` in your `Lab12` directory. You may wish to simulate it to understand the reasoning here. The verilog, with some of the comments and extra spacing omitted, is as follows:

```
‘timescale 1ns/100ps
module InactiveStratum;
reg Clk, A, Z, Zin;
always@(posedge Clk)
  begin
     A = 1’b1;
  #0 A = 1’b0;
  end
‘ifdef Case1
// Case 1: Z inactive:
 always@(A) #0 Z = Zin; always@(A) Zin = A;
‘else
// Case 2: Zin inactive:
 always@(A) Z = Zin; always@(A) #0 Zin = A;
‘endif
//
initial
  begin
  #50 Clk = 1’bz;
  #50 Clk = 1’b0;
  #50 Clk = 1’b1;
  #50 $finish;
  end
endmodule
```

For either condition of compilation, there are two `always` blocks sensitive to change on `A`. There is only one active `Clk` edge in the testbench, and that triggers two successive changes in `A`: The first change is from '`x`' to '`1`', and the second, without advancing simulation time, is a change of `A` from '`1`' to '`0`'.

Here is how the stratified event queue determines this simple simulation:

1. Nothing happens (except to `Clk`) until time 150, when `Clk` goes to '`1`', which is a positive edge in the *active* region during time 150.
2. At time 150, the top `always` block is desensitized, and the simulator puts the one new *active* event, the top assignment statement, into the time = 150 *active* region.
3. Evaluating and finishing the one *active* assignment statement event, the simulator assigns the value '`1`' to `A`.
4. This assignment allows the blocked, second statement in the top `always` block to be read and placed in the time = 150 *inactive* region.
5. The `always` blocks sensitive to the change in `A` to '`1`' now must be read.

The remaining events depend on whether we are in Case 1 or Case 2.

Case 1

5. Both other `always` blocks are desensitized, and the undelayed assignment of `A` to `Zin` becomes a new event in the time= 150 *active* region. The `#0` delayed assignment to `Z` in the other `always` block becomes a new event in the time = 150 *inactive* region.
6. There is only one *active* event, so `A` is evaluated to '`1`', and `Zin` is assigned '`1`'.
7. There are no more *active* events, so the two events in the time = 150 *inactive* region are moved in random order into the *active* region: These events are:

```
A to 1'b0; Z to Zin;
```

8. We know `Zin` definitely is `1'b1`, so `Z` goes to `1'b1`. Whether or not executed first, `A` goes to `1'b0`.
9. The change in `A` again triggers both of the Case 1 `always` blocks, and this puts the assignment to `Zin` into the *active* region and the assignment to `Z` in the *inactive* region.
10. The value of `Zin` is changed to `1'b0`, and this empties the active region. The one event in the inactive region then is moved into the active region and executed, changing `Z` also to `1'b0`.
11. All statements in all `always` blocks have been read; so, there are no further *active* events. All `always` blocks are resensitized. The simulator moves all *nonblocking* assignments, without reevaluation, to the *active* region of time = 150 (there are none). After that, all events from the time = 150 *monitor* region are moved into the *active* region (there are none).

12. The simulator locates the first event in future time, the `$finish`, puts it in the time= 200 *active* region and executes it, terminating the session.

The final values in Case 1 then should be: `A=0`, `Z=0`, `Zin=0`. Notice that delaying the assignment to `Z` has guaranteed that it will be determined by the value of `Zin`, whenever `A` changes.

Case 2

5. Both other `always` blocks are desensitized, and the undelayed assignment of `Zin` to `Z` becomes a new event in the time= 150 *active* region. The `#0` delayed assignment to `Zin` in the other `always` block becomes a new event in the time = 150 *inactive* region.
6. There is only one *active* event, so `Zin`, never yet assigned, is evaluated to 'x', and this value also is transferred to `Z`.
7. There are no more *active* events, so the two events in the time = 150 *inactive* region are moved in random order into the *active* region: These events are:

```
A to 1'b0; Zin to A;
```

8. We know `A` definitely will go to `1'b0`, but we can not tell how the race to assign `Zin` will be resolved. However, we know that either the old or the new value of `A` will be used, so `Zin` definitely will become either `1'b1` or `1'b0`, not `1'bx`.
9. The change in `A` again triggers both of the Case 1 `always` blocks, and this puts the assignment to `Z` into the *active* region and the assignment to `Zin` into the *inactive* region.
10. The value of `Z` is changed either to `1'b0` or `1'b1`, and this empties the active region. The one event in the inactive region then is moved into the active region and executed, changing `Zin` to `1'b0`.
11. All statements in all `always` blocks have been read; so, there are no further *active* events. All `always` blocks are resensitized. The simulator moves all *nonblocking* assignments to the *active* region of time = 150 (there are none). After that, all events from the time = 150 *monitor* region are moved into the *active* region (there are none).
12. The simulator locates the first event in future time, the `$finish`, puts it in the time = 200 *active* region and executes it, terminating the session.

The final values in Case 2 then should be: `A=0`, `Z=(0 or 1)`, `Zin=0`. Notice that delaying the assignment to `Zin` by `#0` has created a race condition on `Z`.

This example was unusually complex to decipher and understand manually; it would be an error to write such code in a real design, because maintenance or modification would be difficult, time-consuming, and error-prone, especially if there was more to the module than three simple `always` blocks. Also, as commented in the on-disc `Lab12` file, two very widely used and well-written simulators produce different simulations!

Application of even one of our rules of thumb would mitigate the complications of this instructional example. The example purposely ignores three of our coding rules of thumb:

- Use nonblocking assignments in a clocked block.
- Never put a `#0` delay in a design.
- Don't allow strange latches (the two `always@(A)` blocks).

9.1.4 Event Controls

There are just two kinds of event control, other than (re)location of a statement in a specific procedural block: The *@ statement* and the *wait statement*. All other statements merely are executed or not.

The @ statement is allowed only in (or at the start of) a procedural block, and it causes a wait in further reading of the block until the event expression changes. This change makes `@` effectively edge-sensitive. We already have used this statement extensively, so examples should not be necessary.

The `@` event expression may be the name of a data object or the choice of an object edge (`posedge` or `negedge`). When the expression changes from some other logic *level* either to a logic '`1`' or '`0`', the expression comes true, the `@` statement is executed, and subsequent procedural statements may be read and possibly executed. When the *edge*-chosen expression changes to the specified level, up to '`1`' (`posedge`) or down to '`0`' (`negedge`), the `@` statement likewise is executed. The `@` statement itself changes nothing in the simulation data.

Introducing `@` with the concurrent `always` ("`always @`") allows an `@` statement concurrently to initiate reading of a procedural block of other statements.

The *declared event*, although rarely used in practice, is mentioned here for completeness. A declared event is introduced by the verilog keyword **`event`**, and it allows for the assignment of names to nondesign objects called *event*s, so that event controls (`@` statements) might be triggered remotely from anywhere in the same module. The syntax is, "`event MyEventName;`" followed somewhere else by an event statement using the name. For example, suppose a task included the event-control line,

`@(MyEventName)` *do_something*;

in which *do_something* was some statement. The event control would be triggered by a statement consisting of an arrow (*hyphen* plus *greater-than*) and the declared name; for example,

`if (`*expr*`) -> MyEventName;`

Declared events effectively are *goto* constructs, and designers have avoided using them; they depend on an unusual syntax and thus provide complexity with no redeeming special functionality. Better to declare a function or task and simply call it under event control.

The `wait` statement is a level-sensitive construct, and it depends on a logical expression, not a design object name. The syntax is

`wait (`*expr*`)` *statement*`;`

When *expr* comes true, `wait` executes its statement. For example, `wait (x>5) x = 0;` Although a construct of the same name is used frequently in VHDL, `wait` in verilog rarely is seen; this probably is because of availability of the more versatile `@` event control.

9.1.5 Event Queue Summary

All the preceding considered, it's a good idea not to use #*n* delays in procedural code or anywhere else in a design module. If necessary, put #*n* delays only on module output drivers.

If this advice is taken, then the event queue complexities can be ignored in synthesizable coding; and, the following simple considerations are the only ones which will remain:

1. Nonblocking assignments in a `begin-end` block always use old values and are executed *after* all other statements at a given simulation time.
2. Blocking assignments in a `begin-end` block always use updated values, as in *C*, and are completed *before* the first nonblocking statement at a given simulation time.
3. Therefore, under ordinary circumstances, use only nonblocking assignments in clocked (edge-sensitive) blocks, and use only blocking assignments in other blocks. This simulates normal setup and sequential activity properly and tells the synthesizer where setup is required.

Because understanding of the stratified event queue is necessary to an understanding of verilog, we shall be using procedural delays frequently for instructional reasons. But, in general, one should not use procedural delays in ones design.

9.2 Scheduling Lab 12

Do this work in the `Lab12` directory.

Lab Procedure

Step 1. Two scheduling and delay examples.

```
module SchedDelayA;
reg a, b;
initial
  begin
  #1 a <= b;
  #1 a = 1'b1;
  #2 a = 1'b0;
  #2 a = 1'b1;
  #1 a = 1'b0;
  #0 b = 1'b1;
  #0 b = 1'b0;
  #0 b <= 1'b1;
  #5 $finish;
  end
//
always@(b) a = b;
//
always@(a) b $<$= a;
//
endmodule // SchedDelayA.
```

```
module SchedDelayB;
reg a, b;
initial
  begin
  #0 b = 1'b1;
  #0 b = 1'b0;
  #0 b <= 1'b1;
  #1 a <= b;
  #1 a = 1'b1;
  #2 a = 1'b0;
  #2 a = 1'b1;
  #1 a = 1'b0;
  #5 $finish;
  end
//
always@(a) b <= a;
//
always@(b) a = b;
//
endmodule // SchedDelayB.
```

Here are a few questions to ponder, based on the two examples above. You may simulate to answer them, but try first to guess without simulation.

In `SchedDelayA`, at what time does `a` first rise? `b`?

In `SchedDelayA`, at what time does `a` last change? `b`?

Answer the preceding questions for `SchedDelayB`.

Step 2. Scheduling and delay with rise-fall timing. Here is `SchedDelayC`:

```
module SchedDelayC;
wire a;
reg b, c, d;
initial
   begin
   #0  b = 1'b0;
       c = 1'b1;
       d = 1'b1;
   #5  d = 1'b0;
   #10 c = 1'b1
   #20 $finish;
   end
//
assign #(1,3,5) a = c;
assign #(2,3,4) a = d;
//
endmodule // SchedDelayC.
```

A. When does `a` first get a well-defined logic level (‘`1`’ or ‘`0`’)?
B. Is the order of first assignment of `b`, `c`, and `d` predictable? If not, why?
C. What is the final value of `a`?

Step 3. Scheduling with mixed-up assignments. First, using the file, `Scheduler.v` in your Lab 12 directory, simulate to see the difference in stratified order of evaluation between events from blocking assignments, zero-delay (`#0`) assignments, nonblocking assignments, and monitor events. See Fig. 9.3. The comments in the verilog file explain the simulation result.

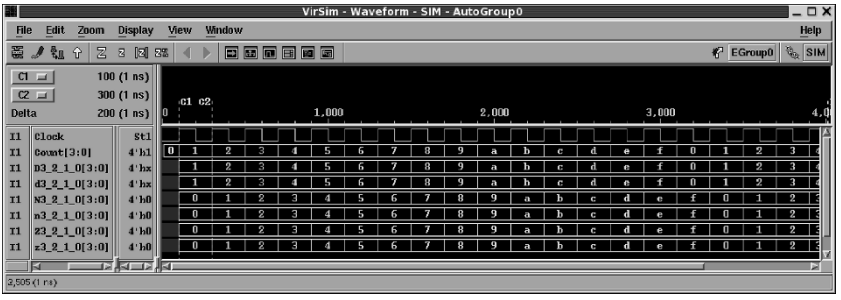


Fig. 9.3 `Scheduler.v` simulation

Optional: For the rest of this Step, use Silos, if you wish to see exactly correct verilog, because simulators designed for serious coding do not handle mixed blocking and nonblocking assignments, or nonblocking concurrency, properly. If not, just read through the verilog files provided. This exercise makes a point about using regular delays, only. In the `Lab12` directory, open the file `BothDelay.v` and read through it. It is based on the code example immediately following the topic above in this section entitled, "Regular vs. Scheduled Delay".

Try to predict when each change will take place. If you simulate this module in Silos, you will find that the `initial` block ends with what may be a surprise: Can you explain it?

Finally, suppose these assignments in a new module:

```
initial
begin
   Y <= 1'b0;
#0 Y <= 1'b1;
#0 Z <= 1'b1;
   Z <= 1'b0;
 $display("display: %04d: Y=%1b Z=%1b", $time, Y, Z);
 $strobe( "strobe:  %04d: Y=%1b Z=%1b", $time, Y, Z);
 $finish;
end
```

What will be the final values of `Y` and `Z` whenever this block is run? Which system function will report them correctly? Should the simulator compiler place the delayed assignments initially in the inactive region or in the nonblocking delay region? What on Earth would be the reason for writing such code!? By now, it should be clear why the synthesizer rejects, or warns about, delayed procedural statements, especially nonblocking ones.

Step 4. Event control sensitivity. Below are two different modules. Instantiate both of them in a third, containing module named `EventCtl`, of course in a file named `EventCtl.v`. Simulate to check functionality (see Fig. 9.4); then, synthesize.

```
module EventCtlPart(output xPart, yPart, input a, b, c);
reg xReg, yReg;
assign xPart = xReg;
assign yPart = yReg;
always@(a,b)
  begin: PartList
  xReg <= a & b & c;
  yReg <= (b | c) ^ a;
  end
endmodule // EventCtlPart.
//
module EventCtlLatch(output xLatch, yLatch, input a, b, c);
reg xReg, yReg;
assign xLatch = xReg;
assign yLatch = yReg;
always@(a)
  begin: aLatcher
  if (a==1'b1)
    xReg <= b & c;
  end
always@(b)
  begin: bLatcher
  if (b==1'b1)
    yReg <= (b | c)^a;
  end
endmodule // EventCtlLatch.
```

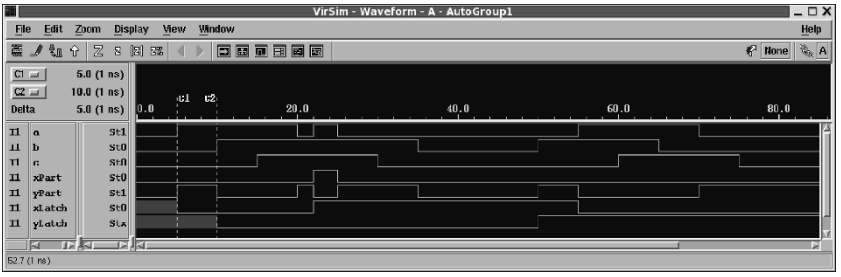


Fig. 9.4 Simulation of `EventCtl`. The `*Part` and `*Latch` wires respectively are from its two instances

The omission of one variable from the sensitivity list should cause synthesis of a latch. The module names are to help keep track of what is what in the synthesized netlist.

The synthesizer is biased not to infer latches; and, if you want latches, in addition to an incomplete sensitivity list, for guaranteed success, you should code for latch modules with 1-bit output ports.

Examine the synthesized verilog netlist to see whether the expected latches have been created.

Step 5. Rise-fall delays. Rise-fall delays mainly are used with structures in a netlist, such as gates or IP ("Intellectual Property": large, predesigned blocks); however, they work with RTL code, too, with some adaptation.

Use parameters `tR`, `tF`, and `tZ`, with ***regular*** delays, in a module named `RiseFall` to schedule the following assignments on each of the `*Reg` variables shown in the code block below:

1. a rise (`tR`) after 4 time units;
2. a fall (`tF`) after 3; and
3. a high-impedance (`tZ`) after 5 time units.

All rise-fall delays specify delay characteristics of a wire or port, not a statement, so it won't work to insert them in procedural code, for example in an `always` block.

Therefore, the values of the `reg`s declared below will have to be assigned to `wire`s in continuous assignment statements to see the effect of specifying different delays for rise and fall. Declare these wires (use the names given, without "`Reg`") and assign them:

```
reg[3:0] OutBusReg;
reg[7:0] DataBusReg;
reg Out2valReg, Out3valReg;
...
always@(negedge Clk)
  begin
  OutBusReg  <= 4'bzz01;
  DataBusReg <= 8'b1111_0zzz;
  Out2valReg <= 1'b1;
  Out3valReg <= 1'bz;
  end
always@(posedge Clk)
  begin
  OutBusReg  <= 4'b0101;
  DataBusReg <= 8'b1zzz_0000;
  Out2valReg <= 1'b0;
  Out3valReg <= 1'b0;
  end
```

Simulate a module which includes the code above, and the specified delays. You should obtain waveforms as in Fig. 9.5.

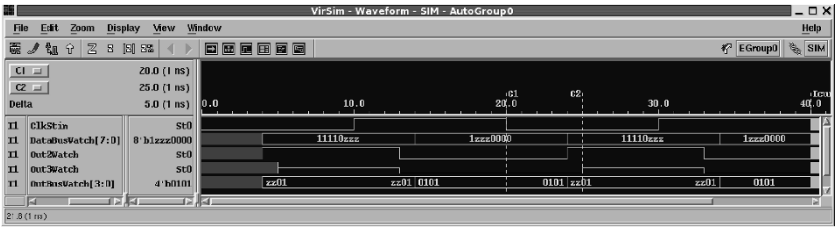


Fig. 9.5 Simulation of `RiseFall`, showing the delay differences

9.2.1 Lab Postmortem

Should the verilog event queue have fewer strata? More?
Mull over and explain the `BothDelay.v` result.
What is the value of named blocks in synthesis?
What is the use of `@(*)`?
Why not use nonblocking assignments in combinational blocks?
What if a '`1`' goes to '`z`'? Is that a rise or a fall? How about '`0`' to '`z`'?

9.2.2 Additional Study

Read Thomas and Moorby (2002) section 6.5 on rise, fall, and three-state net-connection delay specification.

Read Thomas and Moorby (2002) chapter 8 on scheduling of procedural and behavioral events.

(*Optional*) Thomas and Moorby (2002) explains scheduled ("intra-assignment") delay usage in section 4.7. This construct usually should be avoided, but it is worth understanding what it is supposed to do. The authors point out that even where apparently useful, a scheduled delay can not do any better than a regular nonblocking delay in fixing an otherwise malfunctioning design.

(*Optional*) If you are interested, you can find further information on the verilog simulator event queue in IEEE Std 1364, section 11.

Optional Readings in Palnitkar (2003)

Read chapter 7, with special attention in sections 7.2.2 and 7.3 to the use of a # delay expression on the RHS of an assignment statement ("intra-assignment delay"). We shall avoid this usage, but it is important to understand what it is intended to do.

Read section 5.2.1 on rise-fall delays, which primarily are used when describing the timing in netlists.

Chapter 10
Week 5 Class 2

10.1 Built-in Gates and Net Types

In this chapter, we shall study verilog netlists in some detail. Netlists are purely structural implementations made up of elementary gates and hierarchical (sometimes large) other components.

10.1.1 Verilog Built-in Gates

Thomas and Moorby (2002) section 6.2.1 and appendix D discusses all the verilog built-in primitive gates; they are specified in IEEE 1364, section 7. We list them below for convenience, omitting for now the switch-level primitives. Notice that none of them is sequential logic, although sometimes a three-state output can be considered as saving briefly its most recent non-`z` logic state.

Multiple Inputs	Multiple Outputs	One Input & Output	One Output
`and`	`buf`	`bufif1`	`pullup`
`nand`	`not`	`bufif0`	`pulldown`
`or`		`notif1`	
`nor`		`notif0`	
`xor`			
`xnor`			

We already have used `bufif1` in a lab; `notif`*x* is just an inverting `bufif`*x*.

In their port connection lists, these primitives always have their output(s) as the first, leftmost pin(s); and, they are allowed instantiation without an instance name. Verilog primitives are the only gates which can be instantiated without an instance name. As shown already in lab, any of the gates above may be instantiated with a strength specification.

The `pullup` and `pulldown` components should be avoided, because they are not synthesizable.

10.1.2 Implied Wire Names

It isn't always necessary explicitly to declare a net if it is of `wire` type. When one end of a connection is a module port connection, merely providing the name of the port connected is enough to declare implicitly the name of a net. For example, this is a complete module declaration:

```
module NoNets (output Xout, input Ain, Bin);
and And01(Xout, Ain, Bin);
endmodule
```

The following verilog unidirectional buffer or pass-through module also is complete and legal, although it probably should be removed during logic optimization:

```
module NoThing (output Xout, input Ain);
assign Xout = Ain;
endmodule
```

10.1.3 Net Types and their Default

There is a refinement of the idea of implied net names: It is legal to change the default type of implied connection nets, as explained in IEEE Std 1364, section 19. This is done by a compiler directive, `` `default_nettype ``, which affects all implied nets following it during compilation. The type named may be any net type; the default type is `wire` under the usual circumstance of no directive. A `` `default_nettype `` directive changes this default. We have used the `wor` type in a lab, and the others with logic function work the same way. The types which may be `default`ed are as follows:

Type	Net Functionality
`wire`	Connection only.
`tri`	Connection; identical to `wire` except in name.
`tri0`	Pull down to logic '`0`' level, with resistive (`pull`) strength.
`tri1`	Pull up to logic '`1`' level, with resistive (`pull`) strength.
`wand`	Logical *and* of driver logic levels.
`triand`	Logical *and*; identical to `wand` except in name.
`wor`	Logical *or* of driver logic levels.
`trior`	Logical *or*; identical to `wor` except in name.
`trireg`	Unique. Storage of a (capacitive) charge at a given strength level when its driver(s) all are in high-impedance ('`z`') logic state.

The default net type **none** is discussed below. The two remaining verilog net types, `supply0` and `supply1`, are power-supply elements and may not be used as default implied net types. Note that all these net types are gate, RTL, and behavioral modelling constructs and are independent of verilog strength expressions and of the verilog switch-level constructs we shall study later in the course.

Use of the `` `default_nettype `` directive may incur a big risk for very little advantage: What if a module simulates correctly because of implied nets in a default state of, say `wor`, and then is reused or moved elsewhere, so that it is compiled *before* `` `default_nettype wor `` appears? The simulation probably then will fail, possibly with mysterious symptoms.

Most importantly, the default type may be set to `none`. After `` `default_nettype none `` is encountered, no implied net connection of any type is allowed; all nets must be declared explicitly.

The safest way of using `` `default_nettype `` is to avoid it. If this directive has to be used, it is recommended to use it only to set the default type to `none`.

10.1.4 Structural Use of Wire vs. Reg

Thomas and Moorby (2002) (section 5.1) explains the verilog port connection rules. The simulator will enforce them when they are overlooked. It is easy to overlook these rules, especially when writing a testbench, where design considerations are not a priority.

Here's another perspective on the connection rules:

- First, drivers may be a `reg` or any net type. This is shown in Fig. 10.1, in which declarations in the containing module are assumed to include "`reg InReg;`" and "`wire InWire;`".

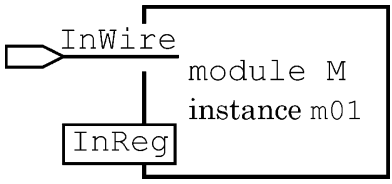


Fig. 10.1 `M m01( ..., .In1(InWire), .In2(InReg) );`

- Second, anything driven by a module output and external to that module must be a net. The reasoning here is very simple: A `reg` may hold a value or be a driver (first rule); if a module output could be connected directly to a `reg` type, there would be contention between the `reg`, as a driver holding a value, and the module output port. Therefore, module outputs must be connected to net types only. This is shown in Fig. 10.2; the `reg` connection to an output port is illegal – and, the dual-meaning '**x**' emphasizes the reason why.

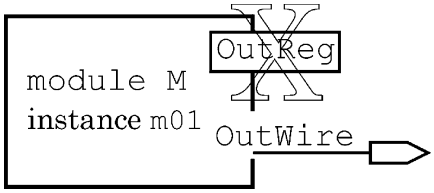


Fig. 10.2 `M m01(.Out1 (OutWire), .Out2 (OutReg), ...);`

- Third, anything driving internally from a module input must drive a net; in other words, internal connections to input ports must be nets. There is no way to avoid this, because driving anything with a module input, including a (internal) `reg`, uses the implied net associated with every input port.
- Fourth, these considerations also mean that a bidirectional port (an `inout`) must be connected on both sides to a net. This is because an `inout` port is subject both to the input and the output restrictions just described; these leave only a net type as a way to connect on either side of an `inout` port. Thomas and Moorby (2002) says that an `inout` port must be connected to a three-state gate; however, a plain `wire` may be used, assuming that internal-external contention can be resolved in a meaningful way because of strength differences.

10.1.5 Port and Parameter Syntax Note

We have done many port and parameter connections already in lab. We point out that both ports and parameters are declared similarly in ANSI format; and, in instantiation, the connection format also is very similar. Parameter declarations, like delay values, are introduced by the '#' token. They are easily distinguished using ANSI declarations, because the keyword, `parameter`, always appears in a parameter declaration. For example, there is no delay involved in this declaration (the parameter named `Delay` might be used for anything):

```
module DeviceM
     #(parameter BusWidth=8, Delay=1) // No resemblance to a delay time.
      (output[BusWidth-1:0] OutBus,
      , input[BusWidth:1] InA, InB, input Clk, Rst);
...
```

In another module instantiating the one above, parameter overrides by name also obviously are not delays:

```
DeviceM #( .BusWidth(16), .Delay(5) ) // Default overrides, not delays.
        DevM_01 ( .OutBus(Dbus), .InA(ArgA), .InB(ArgB)
                , .Clk(ClockIn), .Rst(Reset)
                );
```

A confusion can occur when assigning a parameter value by position, rather than by name. Suppose in the preceding example that we overrode parameter values by position and not by (ANSI) name. The following might look like a delay assignment:

```
DeviceM #(16, 5) // Default overrides may resemble delays.
        DevM_01 ( .OutBus(Dbus), .InA(ArgA), .InB(ArgB)
                , .Clk(ClockIn), .Rst(Reset)
                );
```

For comparison, here is a delay associated with the output of an instantiation of a verilog primitive:

```
xnor #(16, 5) Adder_U12 ( Z, A, B, C );
```

Parameter overrides, like delays, immediately precede the instance name; to avoid misunderstanding, not only between delays and parameters but also among different parameters, it is strongly recommended to override parameters only by name.

10.1.6 A D Flip-flop from SR Latches

In our next lab, a gate-level, simple D flip-flop will be constructed from *nand* gates assembled as three SR latches.

To understand the logic, start with an output and choose output states which imply fully defined inputs. A *nand* gate outputs '1' if any input is '0'; however, when the output is '0', all inputs are fully determined to be '1'. So, look first at the case of a *nand* output of '0'.

Why not begin by considering an SR latch with a small modification in which both SR inputs are tied together?

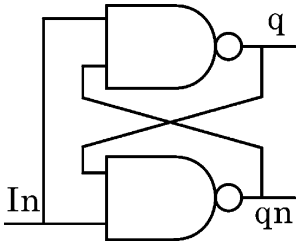


Fig. 10.3 An SR latch with one input

In Fig. 10.3, arbitrarily labelling the outputs `q` and `qn`, suppose `q = 0`. Then, `qn` must be `1` and `In` must be `1`. We could have had `qn = 0`, also, if `In` was `1`. Therefore, the latched state is with `In = 1`. If `In = 0`, then both `q` and `qn` must be forced to `1`.

This design doesn't permit any specified input value to be latched, but it does exhibit sequential behavior.

Now let us add an input which we hope might supply the value to be latched (Fig. 10.4):

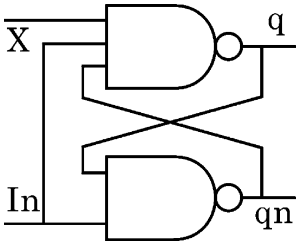


Fig. 10.4 A one-input SR latch with a second input

To latch data, we now must have both `In = 1` and `X = 1`. With `X = 1`, if `In` goes to `0`, `q` and `qn` go to `0`; if `In` then returns to `1`, there is a latched value, but it is indeterminate.

From the latched state, if `X` goes to `0`, `q` is forced to `1` and `qn` to `0`. If `X` returns to `1`, the latched state will remain `q = 1` and `qn = 0`.

In this second design, `X` does act somewhat as a data input, with `In` a clock or maybe an asynchronous clear, if we look only at `qn` as the stored bit. Suppose we hold `X` at `0` and toggle `In` from `1` to `0` and back to `1`: We get `qn = 0`. If we could invert `X` and store its `qn` of `0` in another SR latch which always would be interpreted as inverted data, maybe we could get closer to a real D flip-flop?

For starters, we need something more loosely tied to the `In`. Let's go back and look at a plain SR latch again, as in Fig. 10.5:

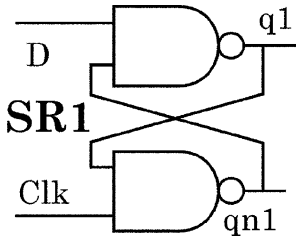


Fig. 10.5 An SR latch which acts like a flip-flop, but only when `D` is '`0`'

Truth-table for the device in Fig. 10.5

time	D	Clk	q1	qn1			
0	0	0	1	1			
1	0	1	1	0	qn1	follows	D
2	1	1	1	0	qn1	latches	D
3	1	0	0	1	q1	follows	Clk
4	1	1	0	0	q1	latches	Clk
5	0	1	1	0	qn1	follows	D

Just as above, we see that it acts like a positive-edge flip-flop, but only when the data input is at '0'.

To make this work for data input '1', we can invert the data as in Fig. 10.6:

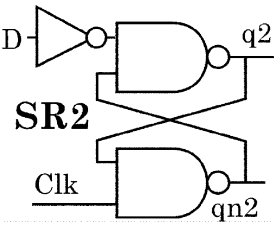


Fig. 10.6 An SR latch which acts like a flip-flop when D is '1'

Truth-table for the device in Fig. 10.6

time	D	!D	Clk	q2	qn2			
0	1	0	0	1	1			
1	1	0	1	1	0	qn2	follows	!D
2	0	1	1	1	0	qn2	latches	!D
3	0	1	0	0	1	q2	follows	Clk
4	0	1	1	0	0	q2	latches	Clk
5	1	0	1	1	0	qn2	follows	!D

So, from the two preceding tables, all we need do now is to build a flip-flop which assigns qn1 to its Q when D was latched as a '0' and assigns qn2 to its Q when D was latched as a '1'. We can't use a mux for this, because a mux is combinational and can not retain the past state of D while selecting the current assignment to Q.

However, we can use a third SR latch in a latched state whenever clock is low; we do this simply by driving the inputs of the third SR latch with the outputs of the *nand*s which receive the clock.

The result is our D flip-flop, shown in Fig. 10.7:

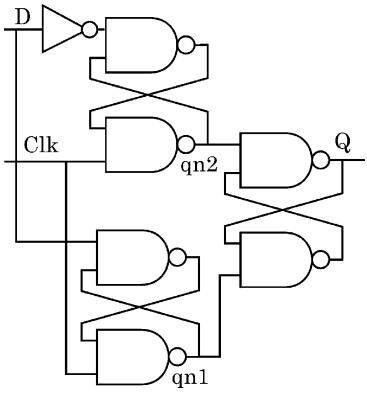


Fig. 10.7 Two SR latches on D, one for '1' and the other for '0', with a third SR to latch the result

Truth-table for the device in Fig. 10.7

time	D	!D	Clk	qn1	qn2	Q
0	1	0	0	1	1	? ℒ
1	1	0	1	1 ℒ	0	1
2	0	1	1	0	0 ℒ	1
3	0	1	0	1	1	1 ℒ
4	0	1	1	0	1 ℒ	0
5	0	1	1	0	1 ℒ	0
6	0	1	0	1	1	0 ℒ
7	1	1	0	1	1	0 ℒ

ℒ = latched

10.2 Netlist Lab 13

Work in the `Lab13` directory.

Lab Procedure

Step 1. A gate-level D flip-flop. Figure 10.8 gives a gate-level schematic of a D flip-flop, with a reset:

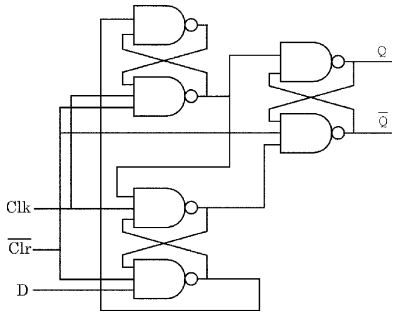


Fig. 10.8 A D flip-flop implemented structurally with *nand* gates

Use this schematic to create your own D flip-flop module named `DFFGates`; use verilog primitive `nand` gates for the gates shown. In your design, make the D flip-flop positive-edge triggered and with asserted-high clear. As usual, put `DFFGates` into a file named the same as the module.

Suggestion: Start by assigning names to the *nand* gates in Fig. 10.8, and use those names as instance names in your netlist. Also, consider the possibility of breaking down the D flip-flop into S-R latch elements and connecting latches instead of *nand* gates. The on-disc answer for this version of the exercise includes a PDF schematic of the S-R latch approach.

Simulate your structural D flip-flop to verify its functionality.

Step 2. A gate-level synchronous counter. Go back to `Lab08` and find your `Synch4DFF` design, a synchronous counter assembled from behavioral D flip-flops. If you did not complete this `Lab08` step, use the answer module provided.

Copy both your behavioral `DFFC.v` and `Synch4DFF.v` into the `Lab13` directory. Duplicate `Synch4DFF.v` as `Synch4DFFGates.v`, renaming the module inside correspondingly.

Replace the `DFFC` instances in `Synch4DFFGates.v` with `DFFGates` instances. Simulate to verify your completely gate-level design (see Fig. 10.9).

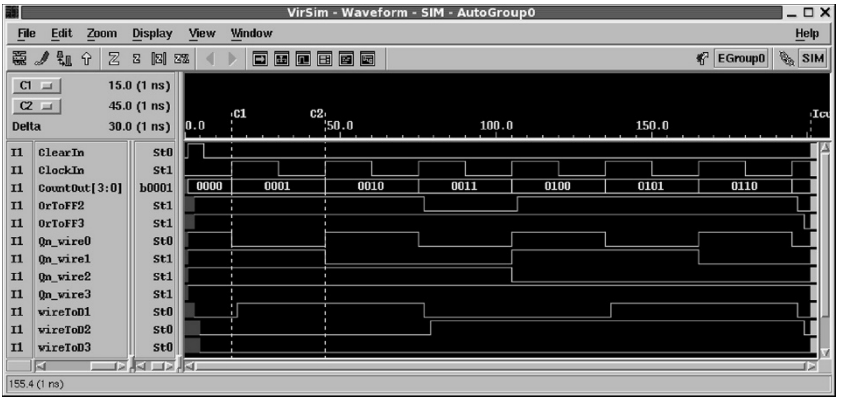


Fig. 10.9 Simulation of the synchronous counter of DFF's defined structurally by *nand* gates

Step 3. Synthesis at gate level. Synthesize both the `Synch4DFF` and `Synch4DFFGates` designs. Keep your synthesis design rules the same for all conditions, but vary the constraints as follows:

Optimize both for area and then for speed. When doing the area optimization, use no constraint except one for area; set that to 0.

When doing the speed optimization, impose no constraint except one for maximum output delay and one for clock period. Adjust the result so that both of these constraints are *un*fulfilled but are no more than 1 ns too small for the actual netlist result reported by the synthesizer.

Compare the speed and area netlist sizes: Did the design make any difference?

Optional: Simulate the speed-optimized netlist (see Figs. 10.10 and 10.11).

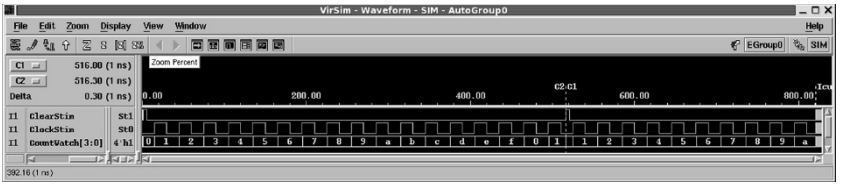


Fig. 10.10 Simulation of the speed-optimized `Synch4DFFGates` synthesized netlist

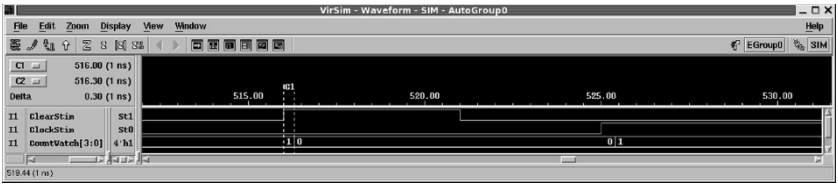


Fig. 10.11 Closeup of the `Synch4DFFGates` synthesized netlist simulation near a counter reset

10.2.1 Lab Postmortem

Do you understand the relationship of built-in gates and strengths?

What is the relationship of structural design to logic optimization and technology mapping?

In synthesizing from a behavioral vs. gate-level design, as in the lab Step 3, should the technology mapper do better when the D flip-flops are synthesized from behavior or from gates? Why?

10.2.2 Additional Study

Read Thomas and Moorby (2002) sections 6.2.1–6.2.3 on net types and primitives.

Read Thomas and Moorby (2002) chapter 10 on strength and switch-level modelling.

Read Thomas and Moorby (2002) section 5.1 on port connection rules.

Optional Readings in Palnitkar (2003)

Section 4.2.3 shows the verilog port connection rules.

Read chapter 10.1 on assignment of delays to gates in a netlist. Look through the rest of chapter 10 if you are interested; we'll study component internal delays later in the course.

Do the exercises in section 5.4.

Work through the verilog of the examples in section 5.1.4; the code is available on the Palnitkar CD, so these examples can be simulated without any keyboard entry. The simulator will display waveforms more realistically if you add a `` `timescale ``.

There are procedural models of the devices of section 5.1.4 in section 6.5, so it may be interesting to contrast the differences. Palnitkar gives a gate-level schematic for a D flip-flop in his figure 6.4 (section 6.5.3).

Chapter 11
Week 6 Class 1

11.1 Procedural Control and Concurrency

In this chapter, we'll review and extend our understanding of verilog procedural control statements. We'll also look further into concurrent control with tasks and other named blocks.

11.1.1 Verilog Procedural Control Statements

The `for` is the most versatile looping control statement in verilog; and, synthesizers typically can make better use of a `for` statement than the others. For completeness, now, we shall describe the others. Except in this lab, you are encouraged to use `for` as much as possible in your work.

We already have used `if` extensively, and we shall discuss the `case` variants below. The remaining control statements are `forever`, `repeat`, and `while`.

`forever`. The syntax simply is `forever`, followed by a statement or possibly a block of them.

The `forever` is a looping statement which is not intended to be terminated once it is executed; however, other verilog constructs may be used to terminate it. It is used almost always in an `initial` block, for obvious reasons. Usually, a `forever` loop will continue until `$stop` or `$finish` is executed by the simulator.

```
initial
  begin : Clock_gen
  Clock1 = 1'b0;
  forever #10 Clock1 = ~Clock1;
  $finish;
  end
...
initial
  begin : Test_vectors
  Abus = 32'h0101_1010;
  Dbus = 'b0;
  #5 Reset = 1'b0;
  ...
  #220 $finish; // Terminate the simulation after about 10 clocks.
  end
```

A concurrent clock generator such as, "`always@(Clock1) #10 Clock1 <= ~Clock1;`", is not equivalent to the `forever` above. The main difference is that the above `forever` clock has been implemented with a blocking assignment, preventing reliable setup of clocked variables unless special delays are added to related combinational logic.

Another difference is that the `forever` schedules an assignment to `Clock1` every 10 units regardless of anything; the `always` can not schedule a new assignment until a change in the current value has been simulated. Thus, the `always` clock can not oscillate if the assignment is a blocking one, while the `forever` clock can.

Also, `always` is itself a concurrent construct, not a procedural one. A looping `always` can not be included in a procedural block.

The `forever` example is not synthesizable; `initial` blocks in general have no corresponding hardware. Of course, neither is the `always` synthesizable.

Finally, notice that the `$finish` in the `Clock_gen` initial block above never will execute. Not until after *forever*, anyway.

Two different termination techniques for `forever`:

```
// Technique 1:
   ...
   forever
     begin
     #1 Count = Count + 1; // Include a delay or @!
     if (Count>=1000) $finish;
     end
```

```
// Technique 2:
begin: Mod16_Counter
  forever
    begin
    Count = Count + 1;       // Count is an integer or wide reg.
    #5 Dbus[4:0] = Count%16; // This delay avoids hanging the simulator.
    end
  end // Mod16_Counter.
// Somewhere else, or in the loop, put disable Mod16_Counter.
```

A restartable clock generator is easy to do:

```
always@(posedge Run)
 begin : RunClock1
 Clock1 = 1'b0;
 forever #10 Clock1 <= !Clock1;
 end
always@(negedge Run) disable RunClock1;
```

Again, one should consider the setup and hold implications of the use of a blocking vs. a nonblocking assignment in any clock generator.

`repeat`. The syntax simply is `repeat` (*number_of_times*), followed by a statement or a block of statements. The `repeat` is strictly a loop-counting construct. When the `repeat` is read, the value of *number_of_times* is stored, and the statement is executed that number of times. Unlike the `for` iteration value, the value of *number* can not be reread or changed during loop execution. The `repeat` statement(s) may be terminated early the same way as shown above for `forever`.

Example of a `repeat`:

```
...
i = 0;
repeat(32)
 begin
 #2 Abus[i] = LocalAbus[i+32] + AdrOffset;
 i = i + 1;
 end
```

You'll notice that the coding overhead for useful invocation of `repeat` is similar to that of `for` in a loop-counting construct; but, the code in a `disable`'d `repeat` would be spread out and thus may be more prone to error. In a block of several hundred lines, the control might be difficult to find. The author has worked with state machine designs of over a thousand lines in a module. Usually, `for` will be preferred to `repeat`, unless no loop control is to be exercised.

`while`. The syntax is `while` (*expression*), followed by a statement or block of statements. The *expression* is evaluated on first reading, and it is reevaluated each time after that, as soon as the controlled statement ends. On any evaluation, if the *expression* is not equal to *false* (a logic-level or numerical '`0`'), the `while` executes

its statement(s). Otherwise, the `while` terminates, and execution picks up after the statement(s) controlled by the `while`.

The `while` arguably is a reasonably efficient alternative to a `for` in many contexts. The `while`'s *expression* may be a relational one involving a count, so `while` is more versatile than `repeat`. Also, a `while` may be almost interchangeable with `for` in a loop-counting context involving a delay. Examples follow.

```
// Example 1:
...
SleepState = 1'b1;
while(SleepState==1'b1)
  begin
  slowRefresh; // A task.
  #SleepDelay Status = CheckUserInput;   // A function call.
  if (Status!=Asleep) SleepState = 1'b0; // Exit the while.
  end
// Example 2:
Count = 10;
while(Count > 0)
  begin
  ... (do stuff)
  #3 BusA[Count] = BusB[Count];
  Count = Count - 1;
  end
```

The following `for` statements are equivalent to the `while` examples above:

```
// Example 1:
...
for (SleepState = 1'b1; SleepState==1'b1;
     SleepState = (Status!=Asleep)? 1'b0: 1'b1
    )
  begin
  slowRefresh;
  #SleepDelay Status = CheckUserInput;
  end
// Example 2:
...
for(Count=10; Count>0; Count=Count-1)
  begin
  ... (do stuff)
  #3 BusA[Count] = BusB[Count];
  end
```

Assignment of an updated control variable has to be located away from the `while` header, in any `while` statement block. This brings out an advantage of the `while`: It works the same way whether or not the control variable is changed

with a delayed assignment. A `for` control header is not allowed to include a delay; so, iterator updates sometimes have to be moved out of the `for` control header.

In the `for`, usually all control is up-front, in the header; this makes it easier to understand the control or to debug errors. For this one reason, it is recommended to use `for` in preference to any other loop control, whenever it is reasonably possible.

11.1.2 Verilog `case` Variants

The main advantage of `if` is that relational expressions can be used, so whole ranges of values may be covered in one expression. By comparison, in a verilog `case` statement, only specific values are allowed in the alternatives. These values may be variables and may be combined with comma separators, but they still have to be enumerated explicitly. However, the table-like arrangement of `case` alternatives often makes a `case` more readable than a chain of `if ... else`'s. In addition, `if` and `case` differ importantly in the way they handle expressions containing '`x`' or '`z`' logic levels.

An `if` attempts equality matches on all logic states at once, including '`x`' and '`z`'. If a bit pattern includes an '`x`', then `if` interprets the '`x`' as a combined '`1`' and '`0`' and returns '`x`' ; thus, it can not express a match, even if that '`x`' is included explicitly in the expression. If a variable *X* contains any bit at '`x`' or '`z`' level, then even "`if (X==X)`" will fail to match, and the `else` (if any) will execute!

A `case` attempts a match on the specific pattern of bits, whether or not any of them is '`x`' or '`z`'. If a vector in the `case` expression contains some '`x`' or '`z`' levels, and one of the alternatives contains the same pattern, `case` evaluates to a match and will selectively execute the statement for the matching alternative.

For example,

```
X = 4'b101x;
Y = 4'b101z;
//
if (X==4'b101x) ...;  // Evaluates as 'x'; won't match the value above.
if (X==Y) ...;        // Won't match; and if (X!=Y) won't, either.
if (X==X) ...;        // This won't match, either!
//
case (X)
  4'b1010: ...;
  4'b1011: ...;
  4'b101z: ...;
  4'bxxxx,
  4'bzzzz: ...; // Comma separation is legal.
  default: ...; // This will execute because nothing else did.
endcase
case (X)
  4'b1010: ...;
  4'b101x: ...; // This matches and will execute.
  4'b101z: ...;
  4'bxxxx,
  4'bzzzz: ...;
  default: ...;
endcase
```

One other thing to keep in mind: Although the conditional operator ("`?:`") can be used as a substitute for `if` under some circumstances, it is an operator and not a statement; it returns an expression. The logic is the same as that of `if` for 1-bit alternatives. For multibit vector alternatives, when an '`x`' or '`z`' appears in the condition, this causes the return value of the expression to go bitwise to '`x`', unless corresponding bits in the '`?`' and '`:`' vector values agree.

A special operator is provided to allow `if` to match unknowns, the ***case equality operator***, "**`===`**" (negation is "**`!==`**"), which we briefly have mentioned before. This operator can not cause an evaluation to be '`x`'. Using this operator, '`x`' and '`z`' match themselves, just as do '`1`', or '`0`', wherever they appear, just as in a `case` statement:

```
X <= 4'b101x;
Y <= 4'b101z;
//
if (X===4'b101x) ... else ...; // expr = 1; the if executes.
if (X!==Y) ... else ...;       // expr = 1; the if executes.
if (X!=Y) ... else ...;        // expr = x; the else executes.
//
// The conditional operator is not an if but accepts case equality:
Z = (X==Y )? 1'b1: 1'b0; // Assigns 1'bx.
Z = (X===Y)? 1'b1: 1'b0; // Assigns 1'b0.
Z = (X!==Y)? 1'b1: 1'b0; // Assigns 1'b1.
```

To take advantage of the table-like `case` syntax, two variants of that statement have been defined in verilog, ***`casex`*** and ***`casez`***.

casex. A `casex` expression matches an alternative as though '`x`' and '`z`' were wildcards: Any bit matches an '`x`' or a '`z`', including an '`x`' or a '`z`'. The statement ends with `endcase`, the same as for the `case` statement we have been using. It can be very confused to allow a specific logic level such as '`z`' become an "anything" character; so, to clarify intent, a '`z`' wildcard in an alternative may be written as '`?`', rather than arbitrarily writing either '`x`' or '`z`', or even mixing them. Examples follow.

```
X <= 4'b101x; Y <= 4'b101z; // X and Y are reg[3:0].
// Example 1:
casex (X)
  4'b100z: ...; // No match.
  4'b10xx: ...; // Executes, because it matches and comes first.
  4'b11xz: ...; // Can't execute on this value of X.
  4'bxxxx: ...; // Would execute if 4'b10xx didn't.
  default: ...; // Would execute if nothing else did.
endcase
// Example 2: Some sort of bit-mask or decoder:
casex (Y)
  4'b???1: ...; // Executes, because of casex LSB wildcard 'z' in Y above!
  4'b??1?: ...; // Would execute if the first one didn't.
  4'b?1??: ...; // Can't execute on this value of Y.
  4'b1???: ...; // Would execute if nothing above did.
  default: ...; // Would execute if no '1' or wildcard in Y.
endcase
```

There might seem to be an advantage in using `casex` when decoding or searching for patterns in data, especially in sparse matrices of data. In Example 2 above, either a chained `if` would have had to be used, or a `case` would have had to be written which individually rejected 12 of the 16 patterns of '`1`' and '`0`' possible in a 4-bit object.

Unfortunately, `casex` is extremely dangerous and error-prone. In the code above, the designer's intent presumably was a prioritized search for `1`'s in the four bits of `Y`. However, any failure in an assignment to `Y` during simulation, or a high-impedance value as actually shown, might cause the `casex` to switch to a `Y` without a '`1`'. Thus, if the intent was to look for a '`1`' in well-defined but arbitrary data, chained `if`'s would be a better way:

```
Y <= 4'b101z; // Y is reg[3:0].
//
if      (Y[0]==1'b1) ...; // 'z' means no match (but === 1'bz would match).
else if (Y[1]==1'b1) ...; // This one executes.
else if (Y[2]==1'b1) ...; // No, not '1' and below the first match.
else if (Y[3]==1'b1) ...; // No, below the one that first matches.
else     ...;             // Executes if no '1' in Y.
```

Notice that indenting the chain this way makes it fairly readable, although the expression is changing on every line, risking a typo. Also, the `if`'s are a little risky, because the evaluations would be wrong if `Y` had been declared `reg[0:3]` instead of `reg[3:0]`, and this is harder to see with the `if`'s instead of a `case`.

The consequences of using `casex` are even worse when one considers synthesis. The "don't care" entries in the `casex` alternatives might seem advantageous in synthesis, because usually a synthesizer uses '`x`' or other don't-cares to its advantage: The designer doesn't care, so the synthesizer is free to do its best. What the synthesizer does, then, is ignore wildcarded alternatives. In the first `casex` example above, then, the second and third alternatives are folded into one, the earlier statements are synthesized but not the later (depending on the synthesizer implementation), and the resulting netlist probably will not simulate the way the original verilog does, assuming that `Y` can vary during design operation. In addition, in that example, the `default` alternative probably will be folded into the `4'bxxxx` one above it during synthesis, which may or may not have been foreseen by the designer.

Because of unexpected consequences of too much wildcarding, it is not recommended ever to use `casex` for anything. When data are sparse, or when wildcarding is unavoidable for other reasons, use `casez` in place of `case` or `if`. At least with `casez`, an unexpected simulator '`x`' will not be wildcarded, and the synthesizer will jump on the don't-cares more lightly.

casez. A `casez` expression matches an alternative with any '`z`' in the expression or in an alternative treated as a wildcard. No '`x`' is a wildcard. Also, as for `casex`, a '`?`' may be used as a wildcard character in an alternative instead of '`z`'.

```
X <= 4'b1x00; // X is reg[3:0].
// Example 1: No match for any of the alternatives:
casez (X)
 4'b100z: ...;
 4'b10xx: ...;
 4'b10xz: ...;
 4'bxxxx: ...;
 4'b0zzz: ...;
 default: ...; // Executes.
endcase
// Example 2: '?' is same as 'z'
casez (X)
 4'b???1: ...; // No match.
 4'b??1?: ...; // No match.
 4'b?1??: ...; // No match.
 4'b1???: ...; // Executes; X[3] matches 1==1, and others are wild.
 default: ...; // Can execute on X == 4'b0000, or on anything
 endcase       // with no '1' or 'z' anywhere, etc.
```

The `casez`, like `casex`, should be avoided; when one is tempted to use it, see whether a simple `case` or chain of `if` can be used instead. If there is no other way, `casez` may be used. The `casex` ***never*** should be used, because it has no wildcarding advantage over `casez`, and it is far more prone to create results in which simulation of the synthesized netlist can not be made to match simulation of the original design.

11.1.3 Procedural Concurrency

We shall use the word *thread* somewhat generically and intuitively here; there is no intentional reference to the similar concept in software parallelism, in which a "thread" is differentiated from a "process".

We already have studied parallel blocks (`fork-join` blocks) and have worked with them a little in lab. It is possible to take advantage of parallelism not only in individual statements, as we have done, but in entire threads of execution of the simulator. This most easily is done by *parallelizing tasks*.

Assigning two or more parallel threads each to its own *task* has many advantages: (*a*) the thread is completely well-defined, because it consists just of the statements in each task. (*b*) Tasks parallelized in a given procedural block easily can be identified for design debugging purposes; one can know where they start (`fork`) and where they finish (`join`). This can't usually be said of a collection of concurrently executing `always` blocks. (*c*) The block names (`task` names) generally will be preserved in the synthesized netlist; otherwise, one must dedicate special attention to naming `always` blocks or whatever else one has chosen instead of tasks. (*d*) Modification of the statements in a task can be relatively free of unintended side effects, because statements in a task are localized and thus can be changed independent of any of the design constructs which cause the task to execute. By contrast, parallelizing a

collection of unnamed statements requires that any change be made in a collection of code containing all the statements parallelized, all at once.

Let's look at a typical example of concurrency implemented with the concurrent construct we have been using most throughout this course, the `always` block. Here is the essential code of a device which detects toggling bits on a 4-bit bus and reports the bit number on a 2-bit output bus. The reason for the concurrency is to simulate nonprioritized evaluation of the bus bits; checking them in a procedural block would imply a priority of some bits, which would be checked before others:

```
...
task CheckToggle(input[1:0] BitNo);
  begin
  @(InBus[BitNo]) #1 ToggledReg = BitNo;
  end
endtask
//
always@(posedge CheckInBus) CheckToggle(0);
always@(posedge CheckInBus) CheckToggle(1);
always@(posedge CheckInBus) CheckToggle(2);
always@(posedge CheckInBus) CheckToggle(3);
...
```

Very similar functionality can be achieved by replacing the multiple `always` blocks with a single `always` block containing a `fork-join`:

```
...
always@(posedge CheckInBus)
  begin
  fork
  CheckToggle(0);
  CheckToggle(1);
  CheckToggle(2);
  CheckToggle(3);
  join
  end
...
```

However, the `fork-join` is not entirely equivalent to the multiple `always` blocks, which is the reason we prefer to call it a *fork-join* rather than a *parallel* block: The `fork-join` block does not exit until all four input bits have toggled, meaning that all four concurrent tasks have been run to completion. If one input never toggles, then each change of an `InBus` bit will spawn a new `fork-join` block instance, which will join the others in the simulator event queue, waiting for all four bits to toggle so it can exit. In effect, this creates a simulator software memory leak, and it might crash the simulator or cause it to become very slow at evaluating new events.

The point here is that, when using a `fork-join` block, one must be certain that the `join` eventually will be reached.

The effect of `fork-join` may be seen easily by its effect on mixed blocking and nonblocking assignments. The following example is for illustration only (Fig. 11.1); it is a bad idea to mix blocking and nonblocking assignments in a real design:

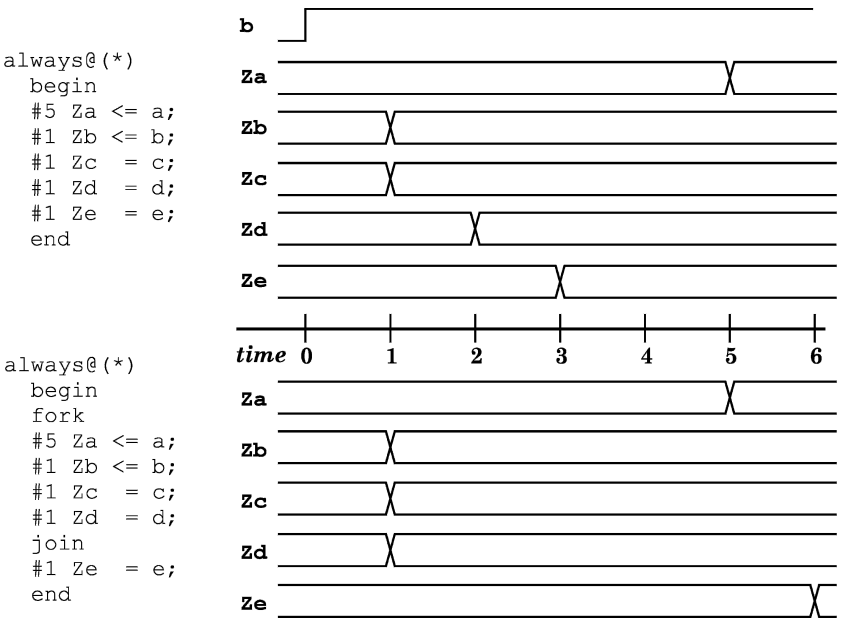


Fig. 11.1 A `fork-join` radically can change scheduling of delayed events

There is another very obvious way of achieving concurrency in verilog: Put the concurrent functionalities into separate `modules` or different module instances. Events in different `module` instances are scheduled concurrently, just as are the `always` and `initial` blocks within them, and just as are the different hardware chips on a board.

11.1.4 Verilog Name Space

One final thing: Up until now, all the examples have built upon assumed experience with *C* language, or other programming languages, in which the name declarations generally precede the programming. Verilog does not allow "global" names, the way *C* does, declared outside a module; so, we have assumed this sort of layout in the verilog source file:

```
module module_name ( I/O's );
reg name declarations; ...
wire name declarations; ...
(programming stuff, using previously declared (or implied) names in statements)
endmodule
```

In many other languages, it is bad style, or illegal, to add declarations anywhere but before the programming. Declare everything first (to avoid name conflicts); then do the programming. However, verilog only requires that a variable name be declared before it is used. Furthermore, functions or tasks may be declared *anywhere* in a module.

We have seen that verilog variable names may be declared locally in `modules`, or in `functions`, `tasks`, and, for temporary use, even inside `always` blocks. These local names do not conflict with one another because the name is visible to the compiler only in the local block of code involved.

Verilog allows declarations in a *named region*. So, it is possible to declare new `reg` or net names anywhere in a module, which is a named region, even between `always` or `initial` blocks, and have those names usable anywhere in the file following the declaration.

This feature should be used with caution; but, in our next lab, it may be useful to declare a few variables close to the `always` blocks in which they are used, rather than far away, at the beginning of the source file. This is a feature which makes verilog more object-oriented than *C*: Objects which do distinct things can be more self-contained than otherwise. For example:

```
module ...
... (500 lines of verilog) ...
reg[7:0] ClockCount;
always@(negedge ClockIn)
  begin : Ticker
  if (StartCount==1'b1)
       ClockCount = 'b0;
  else ClockCount = ClockCount + 8'h1;
  if (ClockCount >= 8'h3a) (do something);
  end
```

It should be mentioned that the synthesizer may not synthesize this, because the `always` block contains more than just one top-level `if` statement.

Notice in the next example that the count would not be preserved by some simulators if the counter was declared local to the `always`, because on every `negedge` of `ClockIn`, the `reg` would be redeclared and would contain `4'bzzzz` by default, going into the `if`:

```
always@(negedge ClockIn)
  begin : Ticker
  reg[7:0] ClockCount; // ERROR! Redeclared every time always is read!
  if (StartCount==1'b1)
       ClockCount = 'b0;
  else ClockCount = ClockCount + 8'h1;
  ...
end
```

Variables declared in a `task`, however, are static and hold their most recent values no matter how many times the `task` is called. Also, concurrently running instances of the same `task` share its declared variables. This holds except for tasks declared **`automatic`**: `task automatic Mytask` gets private copies of its local variables for each instance. Because of this, an `automatic task` may be called recursively.

In the `CheckToggle` examples above, the examples actually will not work as expected, because the input `BitNo` will be declared implicitly as a `reg` just once, and this one variable will be shared among all executing instances of `CheckToggle`.

Thus, all executions of `CheckToggle` above will be checking the same bit of `InBus!` The effect is exactly the same as though `reg[1:0] BitNo` had been declared in the module but external to the task declaration, and the task declaration included no I/O. Task local variables and passed variables (declared task I/O's) are ***static*** `reg` variables shared among all calls. Sometimes this sharing may be desired, because it allows running instances to communicate with one another.

In the `CheckToggle` examples above, and in general when a task may be called more than once in the same simulation time interval, one would want independent variables, not shared ones. Thus, the example task declaration above should have included an `automatic` keyword:

```
...
task automatic CheckToggle(input[1:0] BitNo);
  begin
  @(InBus[BitNo]) #1 ToggledReg = BitNo;
  end
endtask
...
```

A verilog `function` also may be declared `automatic` for purposes of recursion. A simple example of useful function recursion is calculation of a factorial:

```
...
function automatic[31:0] Factorial(input[3:0] N);
  begin
  if ( N>1 )
       Factorial = N * Factorial(N-1);
  else Factorial = 1;
  end
endfunction
...
```

Notice the location of the function width index expression, and the use of the declared input as an implied `reg`. The externally-called `Factorial` function can not return until `Factorial` internally has been called with `N==1`, resulting in evaluation of `N*(N-1)*(N-2) ... 2*1 == N!`. Recursive functions or tasks may not be synthesizable; so, they should be avoided when not essential to the design.

11.2 Concurrency Lab 14

Do this lab in the `Lab14` directory.

Lab Procedure

Step 1. A `forever` block. Write a small simulation model with a testbench clock implemented by means of a `forever` block. Consider a clock generator an exception to the general rule that one should not have more than one `initial` block in a module. Simulate the clock to verify functionality.

Step 2. A `repeat` block. To the Step 1 design module, add a clocked `always` which contains a `repeat` block to initialize a 32-bit bus with a fixed pattern of alternating '`1`' and '`0`' ("...`010101`..."), one bit at a time. Simulate it.

Step 3. A `while` block. To the Step 2 module, add a `while` in its own `always` block to check a 32-bit input bus, one bit at a time, for a specific pattern of '`1`' and '`0`' (anything you want). If the pattern should be found, cause the `while` to set an output flag bit. Do this *without* using a `disable` statement. Simulate the combined module.

Step 4. A `case` application. Write an encoder which receives as input a one-hot bit pattern in an 8-bit register and which outputs the 3-bit binary value giving the numerical position of the '`1`' in the register (starting at 0). The encoder also should have a second output equal to the ASCII code for the bit position ('`0`' `= 8'h30`, '`1`'`= 8'h31`, ...). Use a `case` statement to do the encoding. Simulate the model to verify it.

Step 5. Concurrency exercise. This is what may be a difficult exercise in understanding simulation; don't worry about making the design synthesizable.

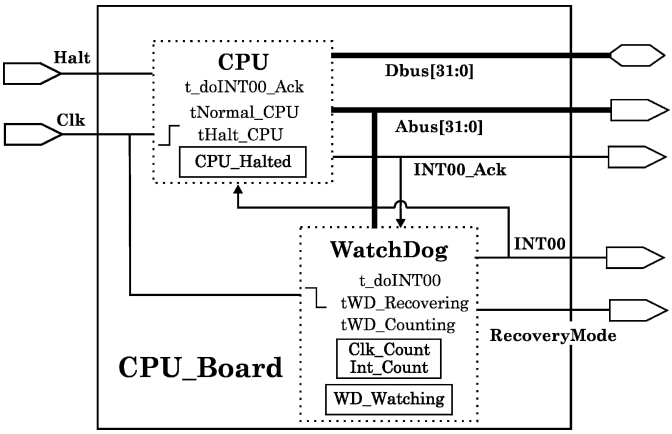


Fig. 11.2 The `CPU` and `WatchDog` design. Dotted blocks represent logic, not hierarchy

Set up a top-level module named `CPU_Board` with two named `always` blocks, `CPU` and `WatchDog`. For this exercise, implement the two `always` blocks as described below; normally, a good design would put the `CPU` and `WatchDog` in separate modules (Fig. 11.2).

The module should have a 32-bit `Abus` output, a 32-bit `Dbus` bidirectional bus, and `Halt` and `Clk` inputs. It also should have three other 1-bit outputs, `RecoveryMode`, `INT00`, and `INT00_Ack`. Supply the same clock to both `always` blocks, but different edges may be used.

We don't care much about `CPU` functionality in this exercise, so we'll use the verilog `$random` system function to supply bit patterns on the `CPU` busses, something like this:

```
...
always@(posedge Clk)
  begin : CPU
  ...
  #1 Dbus = $random($time); // $time repeats only between simulations;
  #1 Abus = $random($time); // so should the (32-bit) random patterns.
  ...
  end
```

A watch-dog device is a simple and essentially nonfunctional component of a system. The watch-dog device monitors activity of a more complex device and helps the system recover if that activity should indicate a malfunction.

Typically, the watch-dog starts a timer whenever activity of some kind ceases; when the timer lapses, the watch-dog device interrupts or reboots the system. In this way, a high-reliability system can be protected from hardware or software defects which cause a communications deadlock. One of the shortcomings of distributed or parallelized systems is that they can deadlock if two or more components persist each in waiting for the other(s) to release shared resources.

A. **The `WatchDog` always block**. It should do these four things: (a) It should count clocks after each change on the `CPU` address bus, resetting the count to `0` after each such change; (b) when the count exceeds 10 cycles, it should interrupt the `CPU` with an `INT00` pulse to attempt to restore it to activity; and, it should repeat the `INT00` pulse periodically until the `CPU` acknowledges; (c) it also should assert a `RecoveryMode` output on the `CPU_Board` module, to signal external devices of its action; (d) on assertion of `INT00_Ack` by the `CPU`, the `WatchDog` should deassert the `RecoveryMode` output, stop issuing `INT00`, and resume counting clocks.

B. **The `CPU` always block**. It should do these four things: (a) It normally should output a variety of `Dbus` and `Abus` patterns, as described above; (b) it should halt on command, by an external (testbench) `Halt` input pulse; (c) it should service an `INT00` interrupt by resuming activity (assuming `Halt` not asserted); and (d) it should assert `INT00_Ack` briefly upon resuming activity, or while continuing activity if the interrupt should occur while the `CPU` was not halted.

Implement the `CPU_Board` to fulfill these requirements, using two named `always` blocks and at least one verilog task.

It is suggested to implement one `always` block completely before worrying about the other one. The `CPU` should be easier than `WatchDog`, so maybe try it first. Simulate the result to verify it (see Fig. 11.3).

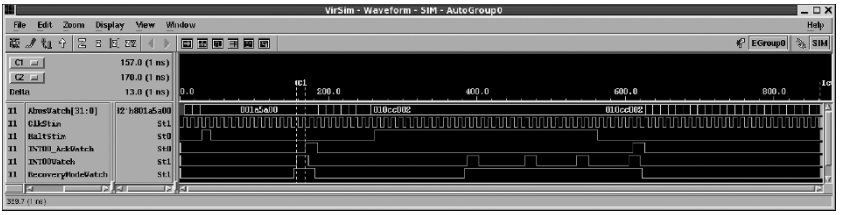


Fig. 11.3 Simulation of an implementation of the `CPU_Board` design

The clock counts in this design are for lab exercise; a real embedded CPU probably would require many more clocks than 5 or 10 to service an interrupt.

11.2.1 Lab Postmortem

What's wrong with `casex`?

How should the `CPU_Board` design be described by state-machine bubbles? One or two machines?

11.2.2 Additional Study

Read Thomas and Moorby (2002) sections 3.1–3.4.4 on procedural control constructs. Note that the `repeat` statement *is* synthesizable with current software.

Also, keep in mind that `casex` never should be used in design, and that `casez` should be avoided unless absolutely necessary.

(Re)read Thomas and Moorby (2002) section 4.9 on `fork-join`.

(Optional) Read the paper by Mike Turpin on the dangers of `casex` and `casez` in synthesis: "The Dangers of Living with an X (bugs hidden in your Verilog)", downloadable at: `http://www.arm.com/pdfs/Verilog_X_Bugs.pdf`.

Optional Readings in Palnitkar (2003)

Read sections 7.5–7.7 on the topics presented this time.

Chapter 12
Week 6 Class 2

12.1 Hierarchical Names and `generate` Blocks

Our purpose will be to introduce in this chapter several different constructs useful to create regular patterns of hardware structure.

12.1.1 Hierarchical Name Access

We introduced this briefly in an earlier lab. It is possible in verilog to access any element in a design tree from any other location in that tree. The rationale is very much like that of access to files in a modern filing system by means of path names: The file is referenced by a path either beginning at the root directory (top of the design) or beginning in the current, working directory (current module instance). In a unix or linux filing system, the name separator is a slash '`/`'; in a verilog design, the separator is a dot '`.`'.

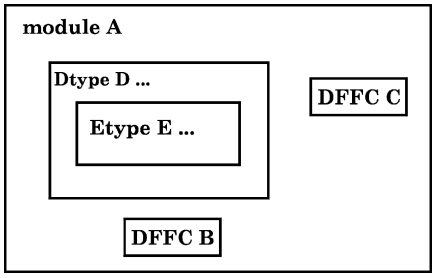


Fig. 12.1 Module `A` and its design hierarchy as a "floorplan"

For example, consider an arrangement of module instances in which module `A` instantiates modules `B`, `C`, and `D`; and module `D` in turn instantiates module `E`; `A` is a module name; the others are instance names, the instances being of various different types (declared modules).

J. Williams, *Digital VLSI Design with Verilog*,

This kind of system is called a hierarchy, because each module instance is contained in only one other one. Representation of this containment resembles a floorplan of a building, as illustrated in Fig. 12.1.

Another way of representing this is as a sort of tree – a family tree, or any other one which grows with its root spreading outward and downward. In a design tree, as in a tree's root system, each element has only one precedent (parent) element. This representation is shown in Fig. 12.2.

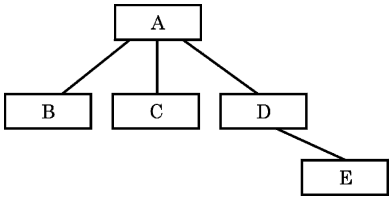


Fig. 12.2 Module `A` and its design hierarchy "tree"

Verilog hierarchical names are implied when any object is named; it does not apply only to `module` instances. So, naming a `task` or a `fork-join` block, or declaring a variable, establishes a name which may be used hierarchically. Any procedural `begin` also may be named.

In verilog, hierarchical name access is allowed anywhere in the design by use of the full path in the hierarchy, through module names or instance names or both. This isn't to say that such access is advisable as a general design practice.

For example, suppose in Fig. 12.1 and 12.2, module `DFFC` was a flip-flop, making `C` a flip-flop instance. Then, inside module `A`, the `Q` output port of module `DFFC`, instance `C`, could be described by the hierarchical name, `C.Q`. For example, `C.Q` could be connected to the `B.D` input port by

```
assign B.D = C.Q;
```

Notice that the preceding statement is written with paths to the ports involved (and their implied nets) implying that the statement must be located in module `A`. The instance names of the two `DFFC`'s can be known from `A` ; therefore, instance access is allowed without specifying the full path in the design.

When a lower-level statement is made, hierarchical references to containing (parent) objects only may be made through the containing `module` names; instance names are not allowed.

If located in `D` (actually, if written in module `D_Type`), the preceding assignment statement would have to be written as,

```
assign A.B.D = A.C.Q;
```

In instance `D` of module `D_Type`, writing `assign B.D = C.Q` would not be legal, because `B` and `C` are instance names of objects not instantiated in `D`.

As another example, suppose module E_Type had an output port wired by OutBus, and suppose module DFFC had a net OutWire; then, to connect a wire OutWire to that port, using a statement in module DFFC, one might write,

```
assign A.D.E.OutBus = OutWire;
```

The same connection made by a statement in module A might be written,

```
assign D.E.OutBus = C.OutWire;
```

Obviously, hierarchical names are a complex and dangerous feature, and they are outlawed in most design projects except only under very restricted circumstances – usually, they are permitted in the design only for reference to objects within the current module. Hierarchical name references are similar to the infamous ***goto*** in software engineering; a rarely useful feature which generally creates bugs.

This use of the hierarchy is not recommended as a general practice; proper design would dictate that wiring be routed through the ports of modules, and not across the verilog name space. However, it is allowed by the language; and, it can be useful in a testbench or assertion, in which reference across the design does not imply a breach of the design structure.

Hierarchical reference can be useful, and consistent with good design, when it is used to access automatically generated substructure within a module. We shall see soon how this kind of *downward* hierarchical access can help us use generate'd structures.

12.1.2 Verilog Arrayed Instances

An interesting feature of the language is that it permits instances to be arrayed, in a way very much like the memory arrays we have studied before. The range specification again is an enumeration range, not a bit width. For example,

```
and #(3,5) InputGater[2:10](InBus, Dbus1, Dbus2, Dbus3);
```

inserts an and gate so that it drives the corresponding InBus bit with the *and* of three data bus bits, each corresponding in bit number. The preceding statement is equivalent to nine instantiations:

```
and #(3,5) InputGater[2] (InBus[2], Dbus1[2], Dbus2[2], Dbus3[2]);
...
and #(3,5) InputGater[10](InBus[10], Dbus1[10], Dbus2[10], Dbus3[10]);
```

Notice that the array number remains as an index value in brackets []; also, the port connections remain indexed.

Instance arrays may be applied to user-defined primitives or modules. The range may be parameterized. Also, the enumeration ranges may be replaced by a concatenation to change the connection bit order. For example,

```
MyBuffer Xbuff[1:3](.OutPin(OutBus), .InPin({InBus[5], InBus[3], InBus[7]}));
```

is equivalent to,

```
MyBuffer Xbuff[1] (.OutPin(OutBus[1]), .InPin(InBus[5]));
MyBuffer Xbuff[2] (.OutPin(OutBus[2]), .InPin(InBus[3]));
MyBuffer Xbuff[3] (.OutPin(OutBus[3]), .InPin(InBus[7]));
```

Note: The Silos simulator (demo version) may not be able to compile arrayed instances of user-defined modules; apparently, it can compile arrays of verilog primitive gates with ascending-order array indices, only.

Next, we shall study the more flexible `generate` statement to create bussed design objects. However, simple bussing can be accomplished easily and transparently by the arrayed instance construct.

12.1.3 `generate` Statements

A `generate` statement is a concurrent statement invoking one of the verilog procedural control statements. A `generate` can be conditional and be based on an `if` or `case`; but, more frequently, `generate` is used with `for` to create arrays of design objects. A `generate` statement can contain design structure such as gate or `module` instances, `regs`, nets, continuous assignment statements, or `always` or `initial` blocks. A `generate` statement is not allowed to contain another `generate`.

There are two somewhat different kinds of `generate` statement, ***conditional*** and ***looping***. We shall discuss the conditional kind first, introducing it in the context of conditional compilation:

12.1.4 Conditional Macroes and Conditional `generate`s

In *C* or *C++*, there is a collection of directives, also called macroes, for the compiler preprocessor; these are used for conditional compilation. For example, in the following, each line beginning with '`#`' introduces a preprocessor directive:

```
#define  MacroName
...
#ifdef MacroName2
...  (compile something)
#else
... (compile something different)
#endif
```

Verilog has similar directives, but they are introduced with an accent grave ‘`’, also called a “backquote”, and not a ‘#’, the latter being reserved in verilog for quantification by delay or parameter. We have seen `timescale in almost every verilog source file, and we used `include in the first lab exercise of the course. We discussed `default_nettype at some length.

Verilog’s compilation directives include `define, `ifdef, and so forth (see IEEE Std 1364 section 19, or our *Week 12 Class 1* chapter, for a complete list), but their usage is not especially subtle or informative, and we shall introduce them only as they become useful to us. However, it is important to distinguish their meaning from that of a conditional `generate`. Compiler directives can create alternative simulations by acting outside the simulation language to rearrange it; conditional `generate`s create language-based alternative structures.

A `generate` can be parameterized just as can be any other verilog statement; so, the `generate` statement can facilitate parameterized modelling of large IP blocks.

For example, here is a conditional `generate` fragment which instantiates a multiplexer named `Mux01` either with latched or unlatched outputs:

```
parameter Latch = 1;
...
generate
if (Latch==1)
     Mux32BitL Mux01(OutBus, ArgA, ArgB, Sel, Ena);
else Mux32Bit Mux01(OutBus, ArgA, ArgB, Sel, Ena/*unused internally*/);
endgenerate
```

A similar result might be achieved by conditional compilation directives:

```
`define Latch 1 // Macro name, but no macro; used as flag.
...
`ifdef Latch // NOT `ifdef `Latch; that substitutes a `1'!
  Mux32BitL Mux01(OutBus, ArgA, ArgB, Sel, Ena);
`else
  Mux32Bit Mux01(OutBus, ArgA, ArgB, Sel, Ena/*unused internally*/);
`endif
```

However, conditional `generate` should be preferred. The reason is that `define’d token states persist during compilation from the point at which they

occur, over all modules subsequently accessed by the verilog compiler. If the compilation order of modules using the `'define`'d `Latch` above should change, (*a*) it might not be defined yet (or even might have been `'undef`'ed) when the `'ifdef` above was encountered; or, (*b*) `Latch` might be `'define`'d for unrelated reasons before the `'ifdef` above was encountered. Either alternative could cause unexpected results *not* accompanied by any warning.

A similar problem was described with `'default_nettype` previously. Avoid compiler directives when possible, except `'timescale`. If they are necessary, try to use them only within a single module file, and then `'undef` every one possible at the end of that file.

12.1.5 Looping Generate Statements

A looping `generate` may be used for hardware alternatives, but its primary benefit is its applicability in building parameterized, repetitive hardware structures which would be time-consuming and error-prone if done manually (even by cut-and-paste). A looping `generate` statement itself can be generated conditionally, but we shall not concern ourselves here with that level of complexity.

The Generate *genvar*

There is a special, nonnegative integer type required for use in a looping generate and which is guaranteed not to be visible in synthesis or simulation; this type is ***genvar***. A `genvar` is used for indexing and instance name generation by the compiler as it converts the `generate` statement to the structure it represents. The conversion may be viewed as an unrolling process: The looping statement is unrolled, like a carpet, to a netlist, which then may be simulated.

One or more `genvars` may be declared inside the `generate` statement, provided each is declared before the looping construct using it. Different `genvars` must be used at different levels in nested loops. The looping construct almost always is a `for` statement.

12.1.6 `generate` Blocks and Instance Names

The block containing the looped statements must be named; this name is propagated, indexed by one or more `genvar` values, as the hierarchical root path to the instances generated.

In effect, the generation block is a structural element, a submodule, in which are located the instances. This makes it possible to use object names within each block without indexing, or modifying, them to make them unique.

A `generate` block, looping or not, may not contain another `generate` block; the verilog language forbids it. It may not contain module I/O declarations, a

specify block, or parameter declarations. However, multiple levels of looping (for loops) are allowed.

As our first example, suppose we want to generate an 8-bit bus of wires, each driving the D input of a flip-flop the output of which should be gated by an inverting three-state buffer.

Also, suppose these other requirements: The library type of the flip-flop is DFFa; the instance names all should be FF-*something*.

In addition, the data input bus should be named DBus; the buffered output should be DBusBuf. There should be common Clk, and Rst; and there should be individual QEna signals; all with the obvious functionality, as shown in Fig. 12.3.

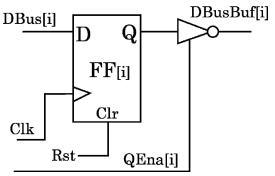


Fig. 12.3 Generic schematic of one of the required array of flip-flops

The code for the generated array is below.

```
// module I/O's include 8-bit DBusBuf output, and DBus & QEna inputs.
...
generate
  genvar i;
  for(i=0; i<=7; i=i+1)
   begin : BuffedBus // Here is the block name.
   wire QWire, QWireNot;
   DFFa FF (.Q(QWire), .D(DBus[i]), .Clr(Rst), .Clk(Clk) );
   not Inv(QWireNot, QWire);
   bufif1 Buf(DBusBuf[i], QWireNot, QEna[i]); // notif1 would be OK here.
   end
endgenerate
```

We used a verilog primitive inverter (not) in this example to illustrate local wiring; a notif1 would have been fine instead of the not and the bufif1.

The result of the above generate loop, unrolled, is an array of 8 named blocks, BuffedBus[0] ... BuffedBus[7]. Each unrolled block contains its own nets named Qwire and QWireNot, and its own three gate instances, one DFFa, one bufif1, and one not, each with the instance name exactly as given in the loop statement; thus, logic levels are propagated properly among the unrolled, i-indexed array of named blocks.

To clarify the process, consider the code example immediately above. The first generate, with index number 0, would unroll as a set of instances with the following identifiers:

```
BuffedBus[0].QWire
BuffedBus[0].QWireNot
BuffedBus[0].FF(.Q(BuffedBus[0].QWire), .D(DBus[0]), .Clr(Rst), .Clk(Clk))
BuffedBus[0].Inv(BuffedBus[0].QWireNot, BuffedBus[0].QWire)
BuffedBus[0].Buf(DBusBuf[0], BuffedBus[0].QWireNot, QEna[0])
```

The index numbers `0` for `DBus[0]`, `QEna[0]`, and `DBusBuf[0]` are from externally indexed vectors, not from the genvar `i`.

We don't get the desired flip-flops named "`FF[i]`", but we do get something just as good, "`BuffedBus[i].FF`".

The unrolled loop is shown in Fig. 12.4.

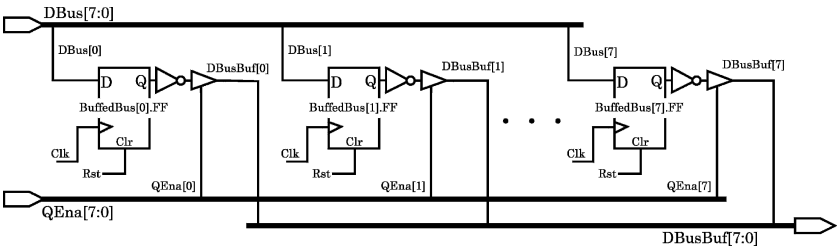


Fig. 12.4 Generated array of flip-flops with individual output enables

Notice the use of the `genvar` in the verilog coding above: It `generate`s the instance names of the unrolled blocks, and it is used to connect `generate`'d instances to outside *indexed* objects. There is no special reason to index components or nets within a `generate` loop statement, because the differently named, unrolled blocks create their own, unique new local instances and nets. For example, the `not` instance in the 3rd block unrolled would be addressed in the simulator or in a synthesized netlist as, `BuffedBus[2].Inv`. If the block name was changed in the loop statement to `DB`, then that `not` would be named, `DB[2].Inv`.

So, if an external object is vectored or arrayed, that object must be referenced in the `generate` loop statement using a `genvar` index value. However, *something* in the loop must be indexed in order to create a mapping to more than one indexed block name. External objects (nets) which are not vectored or arrayed, are not replicated within the unrolled loop and instead are fanned out within it. We can see this with the external nets, `Clk` and `Rst`, in the example above.

Declarations allowed inside a `generate` loop include net types, `reg`, and `integer`. A `task` or `function` declaration, being allowed in a module, also may be located inside a `generate` block, but these declarations are *not* allowed inside the `generate` loop statement.

It is easy to understand how the unrolled naming works, if the above `generate` loop statement is to be used with externally generated vectors of local wiring. If so, then the `genvar` index has to be referenced within the loop block for connectivity. For example,

```
...
wire[7:0] QWire;
//
generate
  genvar i;
  for(i=0; i<=7; i=i+1)
    begin : IxedBus // Here is the block name.
    DFFa FF (.Q(QWire[i]), .D(DBus[i]), .Clr(Rst), .Clk(Clk) );
    notif1 Nuf(DBusBuf[i], QWire[i], QEna[i]);
    end
endgenerate
```

In this example, the first, `i=0` (inverting), buffer would be named, `IxedBus[0].Nuf`. It would be driven by `IxedBus[0].FF.Q` through external `Qwire[0]` and would drive external bit `DBusBuf[0]`. It would be enabled externally by `QEna[0]`.

The following example shows an equivalent `generate` fragment which does not use an external wire to drive `Nuf`:

```
for(i=0; i<=7; i=i+1)
  begin : IxedBus // Here is the block name.
  wire Qwire;
  DFFa FF (.Q(QWire), .D(DBus[i]), .Clr(Rst), .Clk(Clk) );
  notif1 Nuf(DBusBuf[i], QWire, QEna[i]);
  end
```

An alternative approach to this whole buffered FF example would be to put the DFF and its buffer in a new module, perhaps named `FF`, and then to `generate` (or instance-array) on the module.

Finally, here is an example of a combinational decoder implemented by `generate`:

```
parameter NumAddr = 1024;
...
generate
  genvar i;
  for (i=0; i<NumAddr; i=i+1)
    begin : Decode
    assign #1 AdrEna[i] = (i==Address)? 1'b1: 1'b0;
    end
endgenerate
```

This would create a structure with an input bus of the same width as the variable `Address`, which may be assumed to be $log_2(1024) = 10$ bits wide, minimum, and an `AdrEna` bus with 1024 one-bit outputs. Whenever the value of `Address` equalled that of some number n in the range 0–1023, the n-th bit in `AdrEna` would

go high after 1 unit of delay, and all other bits would go (or stay) low. This could be used to enable output in reading a word from a RAM.

An alternative procedural (RTL) way of implementing this decoder, which is very simple and includes no `generate` or component instance, would be something like this:

```
parameter NumAddr = 1024;
integer i;
...
reg[NumAddr-1:0] AdrEna;
always@(Address)
  begin : Decoder
  for (i=0; i<NumAddr; i=i+1)
    #1 AdrEna[i] = (i==Address)? 1'b1: 1'b0;
  end
```

12.1.7 A Decoding Tree with Generate

Let's look further at the design of a big decoder. If the designer had a preferred library component such as a 4-to-16 decoder available, the desired fanout tree might be shown abstractly as in Fig. 12.5.

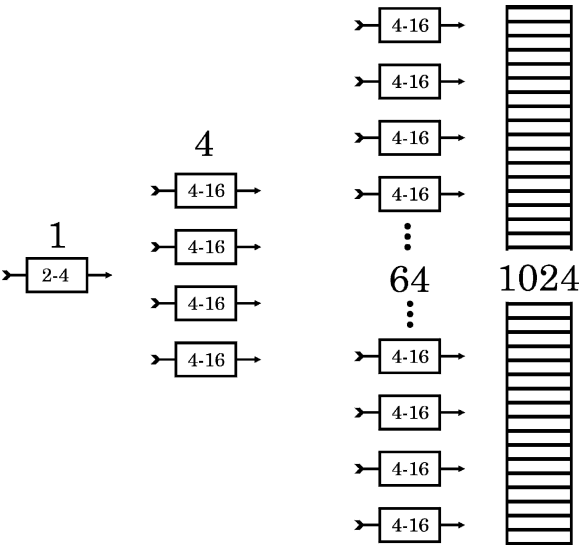


Fig. 12.5 Fanout representation of a 10-to-1024 decoder composed of 4–16's

The leftmost, level 1 decoder, selects one of the four level 2 decoders. Each of them selects one of 16 others, totalling 64. The 64 level 3 decoders each selects one of 16 addresses, for a total decode of 1024 addresses (10 bits on the address bus).

For this arrangement to work, decoders with an *enable* input must be available; a disabled decoder should put '0' on every one of its output pins, thus decoding nothing. A simplified schematic showing this idea is given in Fig. 12.6.

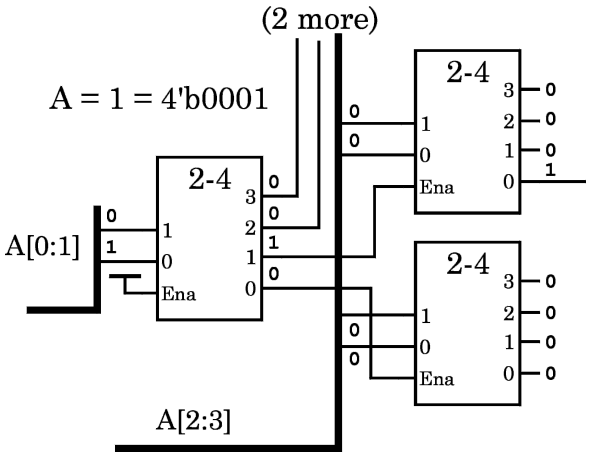


Fig. 12.6 Detail of the select logic for `A = 1`, showing only 2–4 decoders for simplicity

This arrangement easily is mapped to verilog concepts: In the full-sized tree, the level 1 selection can be represented by the state of the two LSB's on a 10-bit bus.

Each subsequent level then adds 4 bits of information, because the number of outputs is just 4 times the number of (binary) inputs. Therefore, the level 2 states map to the next 4 bits, bits 2–5; and, the level 3 to the remaining bits, 6–9, which include the MSB.

To fix the idea, Fig. 12.7 shows the path of enabled decoders in the tree when address `0` is selected:

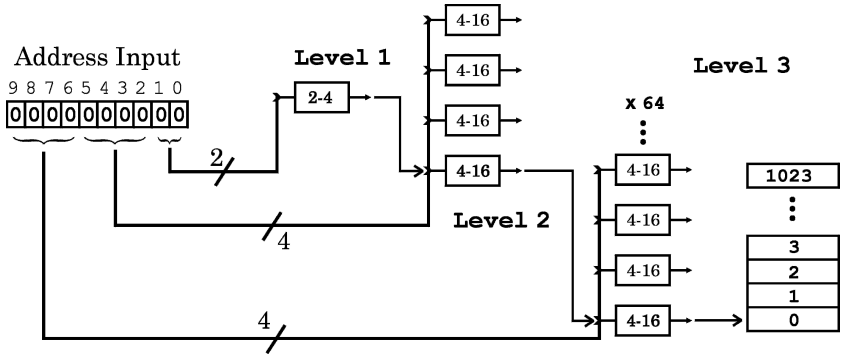


Fig. 12.7 Address `10'h0` decoded. Large arrows indicate the enabled branch

The trick in doing this kind of thing easily is to ignore whether the numerical value of the address which is decoded happens to equal the bit position on the decoded address bus; in almost all applications, it won't matter, because generating an address to *write* data always will result in the same decoded value when a *read* is intended at the same address. What is important is that each decoded bit position map uniquely (1-to-1) to the encoded input value. For example, looking at Fig. 12.7, which shows the decode of address `10'h0`, we can see that address `10'h1` would enable Level 2 decoder 1; with all other address bits 0, this would select output location 256. But, the fact that address 1 is *schematically* decoded to location 256 would be completely irrelevant in the majority of applications.

Given that the level 1 decode above requires less than one 4-to-16 decoder, there isn't any need for a `generate` to implement it. Let's assume an optimized 4-to-16 decoder component is available in the synthesis library named, `Dec4_16`. We'll keep track of the process with the help of mnemonic identifiers; each input wire will be named `in` *something*, and each output wire will be named `Decoded` *something*.

```
// Level 1 decode is trivial:
wire[3:0] DecodedL1;
wire[1:0] inL1;
wire[15:4] DecodedUnused; // Stub to suppress warnings.
//
// Get Address LSB's:
assign inL1 = Address[1:0];
// The level 1 decode, using verilog concatenation:
//             output[15:0]       ,  input[3:0]
Dec4_16 U1({DecodedUnused, DecodedL1}, {2'b0, inL1}); // Done!
...
```

Next, we'll see that `generate` comes in handy for the level 2 decodes, but that they still could be cut-and-pasted conveniently.

What is valuable about using `generate` at level 2, is that the level 2 `generate` provides a template for implementation of the level 3 decode, which, with 64 decoder instances, would be a difficult thing to do correctly any way but by generation. This is how the level 2 decode might be done:

```
// The Level 2 decode, which requires 4 decoders, fully utilized:
wire[16*4-1:0] DecodedL2;
wire[3:0] inL2;
assign inL2 = Address[5:2];
//
generate
  genvar i;
  // Generate the 4 4-16's:
  for(i=0; i<=3; i=i+1)
    begin : DL2
    wire[15:0] tempL2;
    Dec4_16 U2(tempL2, inL2); // Each gets bits 2 - 5.
    //
    // Compose the decoded address from L1 and L2, and assign the bit:
    assign DecodedL2[(16*(i+1)-1):16*i] =
               (DecodedL1[i]==1'b1)? DL2[i].tempL2: 'b0;
    end
endgenerate
```

The level 3 decode, requiring 64 decoders, is hardly any more complicated when done with `generate`; it assigns everything to the 1024-bit address-enable bus, which might represent logic in a memory chip used to select a word for read or write:

```
// The Level 3 decode requires 64 x 4-16's:
reg[(16*4)*16-1:0] AdrEna;
wire[3:0] inL3;
assign inL3 = Address[9:6];
//
generate
  genvar j;
  // Generate the 64 4-16's:
  for(j=0; j<=4*16-1; j=j+1)
    begin : DL3
    wire[15:0] tempL3;
    Dec4_16 U3(tempL3, inL3); // Each gets bits 6 - 9.
    //
    // Compose the decoded address from L2 and L3, and assign the bit:
    assign AdrEna[(16*(j+1)-1):16*j] =
                           (DecodedL2[j]==1'b1)? DL3[j].tempL3: 'b0;
    end
endgenerate
```

The most important rule to keep in mind when writing a generated structure, is that the result will be structural; nothing in the generated structure will be allowed to change during simulation time. Everything is a data object or a connection.

12.1.8 Scope of the `generate` Loop

What difference do these make?

```
generate
for (i=0; i<Max; i = i+1)
  begin : Stuff
  reg temp;
  and A(Abus[i], InBus[i], InBus[i+1]);
  always@(posedge Clk) #1 temp <= &InBus;
  assign #1 OutBit1 = temp;
  end
endgenerate
```

or,

```
reg temp;
always@(Abus) #1 temp <= &ABus;
assign #1 OutBit1 = temp;
generate
for (i=0; i<Max; i = i+1)
  begin : InAnd
  and A(Abus[i], InBus[i], InBus[i+1]);
  end
endgenerate
```

or,

```
generate
reg temp;
for (i=0; i<Max; i = i+1)
  begin : InAnd
  and A(Abus[i], InBus[i], InBus[i+1]);
  end
always@(Abus) #1 temp = &ABus;
assign #1 OutBit1 = temp;
endgenerate
```

12.2 Generate Lab 15

Work in the `Lab15` directory, and be sure to save the results of Step 1 and Step 2 separately, so they can be compared in Step 3.

Lab Procedure

First, we'll do some `generate` . Then, we'll reimplement our `Mem1kx32` memory using generated structure; we then can use this memory in our `FIFO`.

Step 1. Arrayed instances. Implement the second `generate` example in the discussion above, the one with `notifs` ("`IxedBus`"), using arrayed instances and without using `generate`. Simulate your result to verify it. Synthesize a netlist and examine the resulting schematic. Keep the netlist for the next Step.

Step 2. Generated instances. Put the earlier ("`BuffedBus`") `generate` example in the discussion above (the one with `bufifs` and `nots`) into a modified `module` with parameters defining all indexed widths. Use a parameter value in the `generate` loop to ensure consistency of bus widths with the `genvar` variable. Simulate the result to verify it. Synthesize a netlist with your same constraints as in Step 1, and compare the netlist schematic with the one from Step 1. Do the results differ?

Step 3. Area vs. speed in a small design netlist. Pick a design from Steps 1 or 2, synthesize it to compare areas when optimized for area (no speed constraint) instead of speed (no area constraint).

Step 4. RAM `generate` exercise. In our Memory Lab 7 (*Week 3, Class 1*), we designed a verilog RAM behaviorally; it was called `Mem1kx32`. Below the instructions here in this lab is a slightly modified copy of the RAM requirements.

Implement this RAM again, this time in a module named `Mem1kx32gen`. Use a generated vector of 33 D flip-flops (33rd for parity) for storage at each address. Simulate to verify your work.

Consider using the following procedure for this exercise:

A. Delete the old memory array declaration. Don't copy anything but module I/O's and their associated variables from `Mem1kx32`; the old design is not very compatible with a structural `generate`, so attempting extensive reuse may waste considerable time. However, your testbench should work with this lab's design, so keep a copy of the Lab 7 testbench handy.

B. Implement a simple behavioral D FF with `D` and `Clk` inputs, and a `Q` output. No clear or inverted `Q`. Put it in a module named `Bit`.

C. Use a loop `generate` statement with `genvar j` to generate a single vector of 33 FFs.

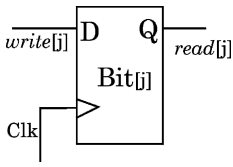


Fig. 12.8 The *j*-th element of the vector `generate` loop

Every `Bit` instance should be connected to a common clock, as shown in Fig. 12.8. One bit (`j`) of a 32-bit data-in wire vector should be connected to each

D; likewise connect a data-out wire vector to each Q. The unrolled word structure should be as in Fig. 12.9.

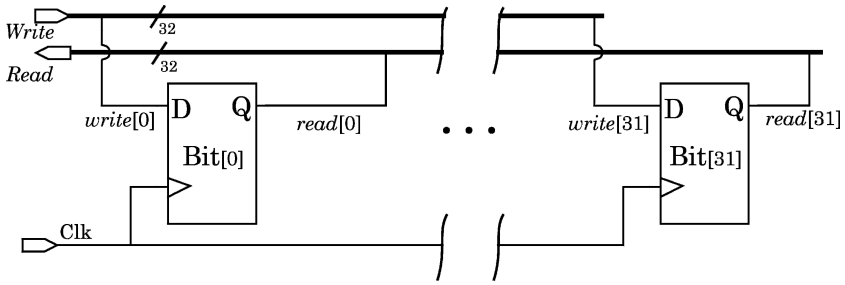


Fig. 12.9 The *j*-generate'd vector structure

After getting the 32 bit vector of storage to simulate, add a bit and implement the parity check. At this point, you should have a generate which creates a single 33-bit vector.

D. Use another loop in your generate statement with genvar i to generate just four RAM storage locations (words), each one consisting of the unrolled vector loop generate statement from Step 4 **C**. To select one word rather than another, the simplest way is to add a buffer to each read bit, so that words not addressed will have their read output (flip flop Q pin) disabled. The disable can be by a decode of the word address.

This will be a prototype of the final 32-word memory. Your testbench can exercise all corner cases conveniently on just four words. A schematic representation of the prototype is shown in Fig. 12.10.

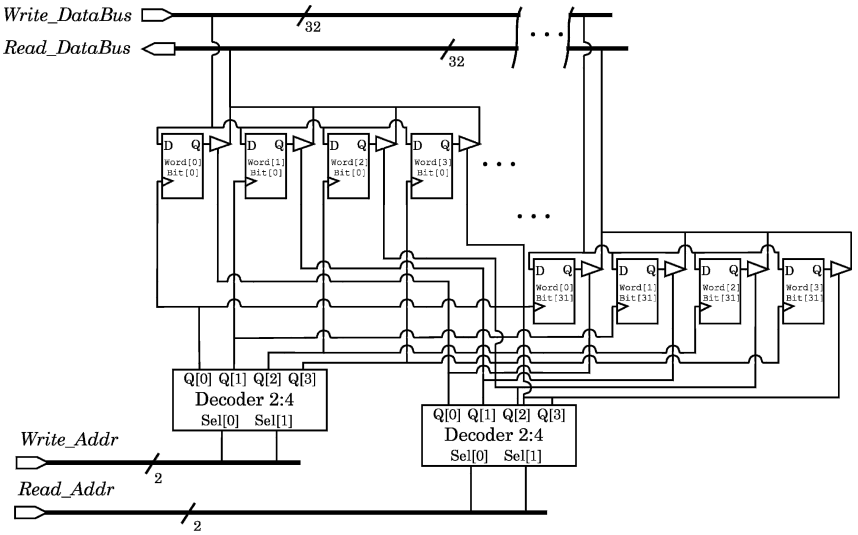


Fig. 12.10 Sketch of the 4-address generate'd memory prototype. Parity is not shown; naming is descriptive, only

E. Now, stop to make sure things are under control. If you haven't already, define a module-level `parameter`, `DHiBit`, to control indirectly the value of `genvar j`. You may do this somewhat as follows:

If the data bus is declared directly by use of the `parameter DHiBit` as a `reg[DHiBit:0]`, the number of data wires (bits) then may be calculated by the compiler as

```
localparam DWidth = DHiBit+1;
```

Then,

```
genvar j; ... for (j=0; j<DWidth; ...
```

Likewise, you may define `AHiBit` to set the number of address lines. The compiler then can calculate,

```
localparam AWidth = AHiBit+1;
```

Given this, the number of storage locations (addresses) is,

```
localparam NumAddrs = 1<<AWidth; // Same as 2**AWidth.
```

You then may use the simulator compiler to set the upper value of `genvar i`,

```
genvar i; ... for (i=0; i<NumAddrs; ...
```

For example, with `AHiBit` at `1`, the address bus width is `2`, and the number of addresses is `1<<2 = 4`. It's also fine to calculate `AWidth-1`, etc., for the upper limit of the memory array index range.

F. Add functionality according to the requirements below, until your RAM model seems complete. Simulate with just a few addresses for verification. Then, set `AHiBit` up to `4` to simulate a full `1k` $\times$ `32` RAM (see Fig. 12.11). Synthesis and optimization of the full RAM may take quite a while (see the answer synthesis script files for why); so, defer synthesis until after including the RAM in the FIFO of the next Step.

RAM `Mem1k`×`32gen` Requirements

A verilog 1k×32 static RAM model (32×32 bits) with parity. Call it, "`Mem1kx32gen`".

Create a structural memory core from D flip-flops using `generate` statements, one for generating the vector of FFs, and another, operating on the first, to generate the addressable array of vectors. You may use behavioral or RTL verilog to manipulate this core (which may be put in its own submodule, even in a separate file, if you like).

Use verilog `parameters` for the 5-bit *address bus* size and total addressable storage. Parity bits are not addressable and are not visible outside the chip.

Use two 32-bit data ports, one for *read* and the other for *write*. Supply a *clock*; also an asynchronous *chip enable* which causes all data outputs (read port) to go to '`z`' when it is not asserted, but which has no effect on stored data. The system clock should have no effect while chip enable is not asserted.

Supply two direction-control inputs, one for *read* and the other for *write*. Changes on read or write have no effect until a positive edge of the clock occurs. If neither is asserted, the previously read values continue to drive the read port; if both are asserted, a read takes place but data need not be valid.

Assign some time delays for data and address bus changes, and supply a *data ready* output pin to be used by external devices requiring assurance that a read is taking place, and that data which is read out is stable and valid. Don't worry about the case in which a read is asserted continuously and the address changes about the same time as the clock: Assume that a system using your RAM will supply address changes consistent with its specifications.

Also supply a *parity error* output which goes high when a parity error has been detected during a read and remains high until an input address is read again.

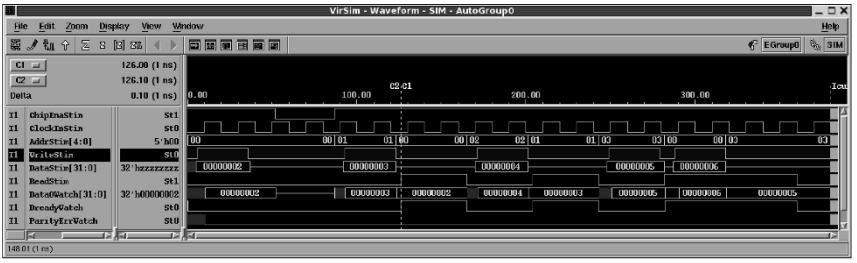


Fig. 12.11 Cursory simulation of the `generate`'d 32-word `Mem1kx32gen` RAM

Step 5. Implement a FIFO using a `generate`'d RAM. Create a new subdirectory in the `Lab15` directory, and copy your FIFO model from `Lab11` into it. This

model should consist at least of three files, `FIFO_Top.v`, `FIFOStateM.v`, and `Mem1kx32.v`.

Use your new verilog `generate`'d RAM to provide the storage for a FIFO by substituting `Mem1k×32gen` for `Mem1k×32`. This might be too big for the demo version of Silos.

Simulate. After verifying the functionality, it would be feasible to synthesize the memory alone; but, why not synthesize the whole FIFO as follows:

First, with area optimization only, and no design rule constraint (fanout, load, or drive limits) or speed constraint of any kind. This is a fairly large module for synthesis (about 30,000 transistor equivalents); a register file this large usually would be implemented as a hard macro rather than being synthesized in random logic.

Notice that the synthesizer will report that none of the parity bits are connected to anything. We have omitted parity detection in this model, so we should expect these warnings.

Second, synthesize with no design rule, a zero-area constraint, a 500 ns clock period, and just a modest 50 ns max delay speed constraint on the outputs. A substantially stiffer speed constraint or any design rule constraint will prolong optimization time noticeably in this design; the cause is the `generate`, not the design itself. Do not apply an input or output delay constraint on clocked data, because this also will lengthen the optimization time greatly.

Third, ***optionally***, if time permits, try synthesis using this procedure: Define a 500-ns clock and apply a zero-area constraint, then compile with no other constraint. Then, in the same synthesis script run, set a don't-touch on the memory module, apply the design rules and other constraints just below, and compile a second time with incremental mapping, only.

Optional rules and additional constraints for incremental compile:

```
set_drive      10.0 [all_inputs]
set_load       30.0 [all_outputs]
set_max_fanout 30   [all_designs]
#
set_max_delay    50 [all_outputs]
set_output_delay 1  [all_outputs] -clock Clocker
set_input_delay  1  [all_inputs]  -clock Clocker
```

The generated FIFO netlist will not simulate correctly – not because of the `generate` but because we have not yet designed a synthesizable FIFO state machine. We shall postpone netlist simulation of the FIFO until later in the course.

12.2.1 Lab Postmortem

Contrast the use of index values in arrayed instances vs. `generate`'d instances.

Can a `genvar` name conflict with that of an integer in the same module?

Does a `generate` block create its own name space?

12.2.2 Additional Study

Read Thomas and Moorby (2002) section 3.6 on scope of names and hierarchical names.

Read Thomas and Moorby (2002) sections 5.3–5.4 on arrayed instances and `generate` blocks.

Optional Readings in Palnitkar (2003)

Look over Appendix C for the various verilog compiler directives. You may be able to guess what half of them do just by their names.

Read section 7.8, on `generate`.

Chapter 13
Week 7 Class 1

13.1 Serial-Parallel Conversion

We put aside our serdes project a while ago to strengthen our understanding of verilog; now we return to it to implement another part of the deserializer.

13.1.1 Simple Serial-Parallel Converter

We have implemented the serialization frame encoder (Lab 6, Step 8; *Week 2 Class 2*), and we studied the processing, which is to say, the decoding, of the incoming serial stream when we developed a way of synchronizing a PLL with our framed packet protocol; this was toward the end of Lab 10 (*Week 4 Class 1*).

So, in terms of our project, we have only a little more to do beyond putting together some things we have done already, and modifying a couple of modules for synthesis. However, let's look some more into the parallelizing of a serial stream at this point.

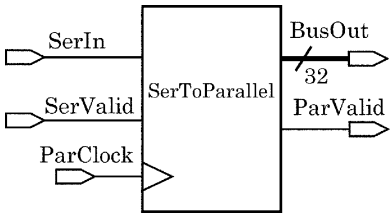


Fig. 13.1 Generic serial-parallel converter

The general functionality is given in Fig. 13.1. A parallel bus of output latches or flip-flops should be present, as well as a clock or other synchronizer, and at least one serial input. A purely combinational deserializer is possible, but it would be difficult to use for anything without output synchronization.

There has to be a serial clock, but this may be the same as the parallel clock. The serial clock, if not the same as the parallel clock, may be generated either from the serial side directly, or, as in our serdes project, derived from the serial data.

There should be an internal register to hold partially deserialized data, but this shouldn't be the same as the one latching the parallel output, unless the design is to allow the fully parallelized data to be available on the output for a duration of less than one clock.

Optional functionality might include a flag announcing when parallel data on the output bus are valid; however, this functionality in principle could be achieved by clock-counting. There should be a `SerValid` input from the serial side to flag when serial data are available to convert. This `SerValid` flag might toggle with each serial bit or byte received; or, instead of such a flag, or we might provide a serial clock based on the assumption that the serial stream can be synchronized with it. If we want to be able to start up the device with a zeroed shift register, we may provide an optional parallel-side reset to do this. We shall assume that the converter could have no control over the serial line's transmitter, so it would not be meaningful to provide a serial reset.

Summary of Serial-parallel Conversion Functionality

- Latched parallel data out.
- Parallel-side clock in.
- Serial data in.
- Serial-side clock in (optional).
- Internal deserialization register.
- Parallel-valid output flag (optional).
- Serial-valid flag (optional).
- Parallel-side reset (optional).
- No serial-side reset.

In our serdes project, the serial clock is embedded in the serial stream and must be decoded by the receiver without any `SerValid` flag. This has the obvious disadvantage of requiring some startup overhead before the clock phase and frequency can be established; it has the more-than-redeeming advantage of not requiring any separate clock, with attendant differential phase-lag, separate cross-talk, or related noise issues. So long as the data are usable, the embedded clock will be usable, too. This permits a data stream independent of any other signal in the design; in practice, this is an important factor permitting GHz frequency-range data transmission almost with a zero error rate.

13.1.2 Deserialization by `Function` and `Task`

If we don't worry about frame boundaries, parallelization is almost trivial in verilog: We shift in the serial data until we have enough for a parallel word; we unload the shift register onto the parallel bus; and, we continue shifting. We may assume that parallel clocking is fast enough that no serial data will be lost.

Here's a verilog representation of a simple, generic deserialization:

First, we do the shift. This can be implemented by a function, which can return the shifted value at some reasonable delay time, but which can be allowed to perform the shift itself in zero simulation time. Verilog has a built-in shift operator, so all the function has to do is be sure to append the new bit being shifted in:

```
parameter ParHi = 31;
...
function[ParHi:0] Shift1(input[ParHi:0] OldSR, input NewBit);
  reg[ParHi:0] temp;
  begin
  temp    = OldSR;
  temp    = temp<<1;    // MSB goes lost.
  temp[0] = NewBit;
  Shift1  = temp;
  end
endfunction
```

When calling the `Shift1` function, we pass it the current shift register and the new serial bit to be shifted in. This function can be simplified; it is written as above for instructional reasons.

Second, we do the conversion. Unloading the shift register onto the parallel bus involves a `ParValid` flag and probably a few delays, so why not implement it as a task?

```
parameter ParHi = 31;
...
reg ParValidFlagr;
reg[ParHi:0] ParSR, ParBusReg;
//         rise, fall
assign #( 1,    0 ) ParValidFlag = ParValidFlagr;
...
task Unload32; // Copies the parallel SR to the output bus.
  begin        // Also clears the SR for the next word.
     ParValidFlagr  = 1'b0;  // Lower the parallel-valid flag.
     ParBusReg      = ParSR; // Transfer the data.
  #5 ParSR          = 'b0;   // Clear the SR.
     ParValidFlagr  = 1'b1;  // Raise the flag.
  end
endtask
```

The delays are included just for illustration. Keep in mind that delays in procedural code should be avoided for design purposes; module output lumped delays, if any, should be estimated to account for delays synthesized from procedural code.

Third, we regulate the shift. We have to put things together in a way that the shift-register shifts in the bit on the serial line on each clock, provided the serial bit is valid and the device is not being reset.

Actually, there was no design reason in the second step above to clear the shift register after each conversion; a shift-counter will be used to determine unloading of new data, not left-over data, to the parallel bus. However, starting each conversion with a clear register does make it easier to see new data arriving during simulation.

The `temp` register in the `Shift1` function above was included for the same reason of clarity during simulation; `Shift1` could be written more minimalistically this way:

```
function[ParHi:0] Shift1(input[ParHi:0] OldSR, input NewBit);
  begin
  OldSR  = OldSR<<1;
  Shift1 = {OldSR[ParHi:1],NewBit};
  end
endfunction
```

An `always` block to assemble our code fragments is shown below.

```
always@(posedge ParClk, posedge ParRst)
  begin : Shifter
  if (ParRst==1'b1)
       begin
       N          <=     0; // N counts the bits shifted.
       ParSR      <=   'b0; // The shift register.
       ParBusReg   <=  'bz; // The parallel out bus.
       ParValidReg <= 1'b0; // ParValid.
       end
  else if (SerValid==1'b1) // Ignore the serial line if 0.
       begin
       ParSR <= Shift1(ParSR, SerIn); // function called.
       N     <= N + 1;
       if (N>ParHi)    // If 32 bits shifted.
         begin
         Unload32; // task called.
         N <= 0;
         end
       end // SerValid.
  end // Shifter.
```

13.2 Lab Preface: The Deserialization Decoder

The next lab will begin with a couple of very important exercises on deserialization. After that, there will be a first cut at implementing the deserialization decoder of our serdes project.

It is essential to understand what the deserialization decoder (`DesDecoder` module) is supposed to do: ***It identifies data frames which were encoded in the serial input data stream***. It also generates a 1 MHz clock meant to be synchronized with the sender's clock.

In our design, each frame is 16 bits wide and ends with an 8-bit pad pattern. The `DesDecoder` decodes the packets; it does not do a simple logical decode, such as did the 4–16 decoder of our previous lab. In decoding the packets, it also extracts the sender's clock, which is implied by the format of the incoming serial data stream.

Our deserializer spans two independent clock domains. The serial stream comes into the deserializer at an approximately known frequency depending solely on the clock in the sender's clock domain. However, the deserializer also clocks data out of its FIFO using a clock in the receiver's clock domain. The FIFO input is in the sender's domain; the FIFO output is in the receiver's domain.

Looking at our `Deserializer`'s PLL, it is clocked by a free-running 1 MHz clock which the `DesDecoder` attempts to synchronize to the sender's clock. This clock emulates, in verilog, an on-chip clock controlled by a variable-capacitance oscillator. This clock is concerned only with the deserialization and with the FIFO input. The PLL's comparator receives two different 1 MHz clocks: One is the free-running one which is multiplied by 32 to clock in the serial data; the other is a clock extracted by the `DesDecoder` from the serial data stream. If the free-running clock was perfectly synchronized with the serial data stream coming in, it would be perfectly in the sender's clock domain, and every packet could be identified correctly.

Any `DesDecoder` failure to decode a packet means that there is no guarantee that the free-running clock is indeed in the sender's domain. The PLL comparator keeps track of any frequency difference between the incoming stream and the free-running clock; however, the PLL does not adjust its frequency unless the `DesDecoder` can confirm that it has identified a packet. Whenever the `DesDecoder` identifies a packet, it allows the PLL to adjust its frequency slightly, and it synchronizes an edge of the free-running clock with the identified packet boundary.

Our design uses integer arithmetic to equate the two sender-domain clocks, the one actually in use by the sender, and the free-running clock of the `Deserializer` PLL. Because 1/32 of the period of a 1 MHz clock is not an integer value in ns but only can be rounded near to an integer, our PLL never will be truly locked in the way an analogue PLL could. However, it can be locked in approximately, and this can be close enough that several packets can be extracted correctly before the serial stream drifts out of synchronization.

Before beginning the lab, you may wish to review our original plan for the serdes project as described for *Week 2*, *Class 2*. In the present lab, we shall *not* extract the sender's clock (that will be later); however, we shall decode the packets being sent.

Here again, in Fig. 13.2, is the block diagram of the conceptual serial-parallel converter (deserializer) as described in Week 2 Class 2. We shall implement a simplified, first-approximation Deserialization Decoder in this lab.

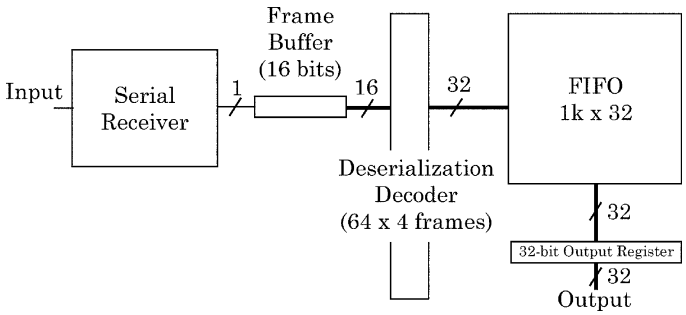


Fig. 13.2 Week 2 serdes `Deserializer` data flow

Recall what was done in Step 4 of Lab 6 of *Week 2 Class 2*: We decided then on a packet format in which a frame of 16 bits would be used to represent each 8-bit byte of serial data.

The data bits would be contiguous and bounded below (later in the stream, which arrives MSB first) by a pad byte consisting of three `'0'` bits, then a 2-bit count locating the preceding data byte in its 32-bit word, then another three `'0'` bits. A packet of 32 bits of data then would look like this, each `'x'` representing a data `'1'` or `'0'`:

```
64'bxxxxxxxx00011000xxxxxxxx00010000xxxxxxxx00001000xxxxxxxx00000000,
```

with serial arrival being from left (earliest) to right (latest).

13.2.1 Some Deserializer Redesign – An Early ECO

According the block diagram of Fig. 13.2, which was meant to be conceptual, we can count on the incoming data's having been deserialized and collected into 16-bit words before the Decoder. However, given our serial protocol, no framing can be done in the Frame Buffer until the serial clock has been decoded; so, as shown, it is not obvious that the 16 bit buffer would not contain data crossing encoded-byte boundaries.

The Frame Buffer, then, interpreted as a serial-parallel converter, first has to be aligned on data+pad boundaries. This might be done in the diagram of Fig. 13.2 by including a PLL feedback from the Decoder to the Serial Receiver.

But, it is unclear how this feedback could be separated from deserialization; so, probably, for data flow purposes, the best plan is to fold the Frame Buffer into the Decoder logic and no longer view it as an independent block.

The Decoder should manipulate the digitized serial stream using its own registers and buffers. Our `Deserializer` design data flow block diagram then should be amended as shown in Fig. 13.3.

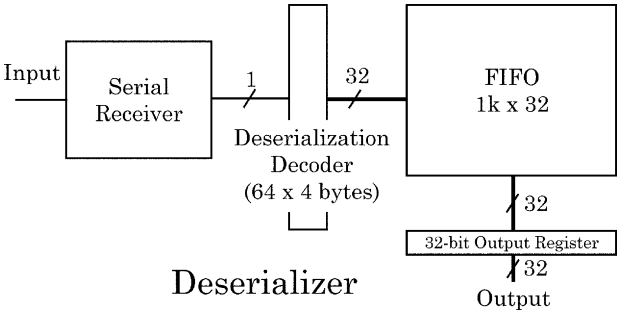


Fig. 13.3 Amended serdes `Deserializer` data flow

We are not interested in PLL clock synchronization in this exercise. If we assume that the PLL is located in the Serial Receiver block, we can delete the Frame Buffer block; we then have no reason to worry about distribution of the PLL clock at the block level, and we need not show any feedback from the Decoder.

This block diagram, after all, is just data flow and need not include any representation of the clocking scheme.

Because it takes 64 bits to encode 32 bits of data, it will take two `ParClk` cycles, at 1 MHz, to process each 32-bit decoded word. These considerations are discussed in detail in Step 4 of Lab 10. We only assume here that we can shift in the serial data from port `SerIn` at 32 Mb/s, using an externally supplied, 32 MHz serial clock named `SerClk`.

13.2.2 A Partitioning Question

Going beyond data flow, another question here is one of second thoughts: Should we indeed establish frame synchronization here, in the Deserialization Decoder, or should we leave it to PLL-related clock-extraction code in the Serial Receiver block? The verilog already mostly has been written (see the final code example in Lab 10 of *Week 4 Class 1*). Frame synchronization is essentially equivalent to serial clock extraction.

If we leave serial clock extraction *in the Serial Receiver*, then the PLL will be localized entirely there, and there will have to be considerable digital structure, maybe a shift register or small register file, to handle the pad-pattern extraction. But, serial data arriving at the Deserialization Decoder (`DesDecoder`) will be accompanied by frame-boundary flags from the Serial Receiver, and parallelization in the `DesDecoder` block could be very simple and generic.

On the other hand, if we put serial clock extraction *in the `DesDecoder`*, the PLL may be located either in the Serial Receiver block or in the `DesDecoder`. If the former, synchronization feedback will have to be provided by the `DesDecoder` to the Serial Receiver; if the latter, we have mixed digital and analogue in the

DesDecoder block, requiring special layout procedures and other DesDecoder design overhead.

The answer we shall adopt here, is to put the PLL in the Serial Receiver block, and to do the frame synchronization (serial clock extraction) in the DesDecoder. This will isolate analogue functionality to the Serial Receiver block and will reduce duplication of digital effort in deserializing the data. The DesDecoder will have to extract a (nominally) 1 MHz clock from the serial stream being parallelized and provide it to the PLL. This is shown in Fig. 13.4.

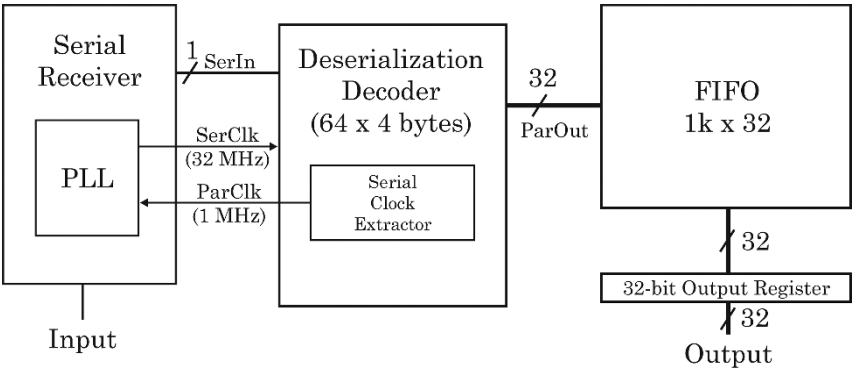


Fig. 13.4 Final Deserializer detailed block diagram

A 1-MHz clock is a low-frequency signal in our design, so no special precaution will have to be taken, except possibly to account for the transport distance delay, a small phase lag, from the DesDecoder to the PLL.

ECO completed; now we move on to the lab exercise

13.3 Serial-Parallel Lab 16

Do this work in the Lab16 directory.

Lab Procedure

After some design warm-up, we'll discuss, and then implement, a first-cut Deserializer for our serdes project.

Step 1. Generic deserializer. Using the lecture material as desired, implement a deserializer which fulfills the generic description above and realizes the above SerToParallel block diagram's I/O's. The generic requirements are just to clock in the serial data until 32 bits have been acquired, and then to copy them onto the parallel bus, using the same clock (possibly the opposite edge). Assume that the data are unframed and should be grabbed in 32-bit words so long as the SerValid flag is asserted.

Simulate the design at least for three deserialized, 32-bit words. Use `$random` for serial data (see Figs. 13.5 and 13.6). Do not spend time synthesizing this design.

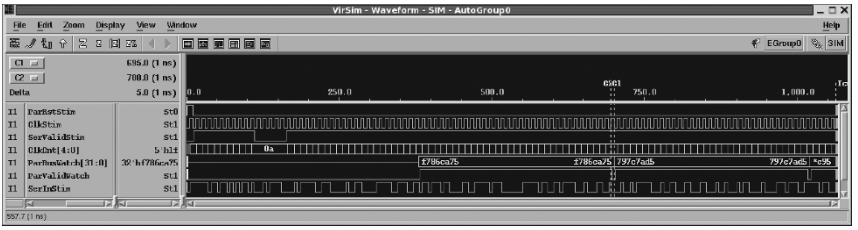


Fig. 13.5 The generic deserializer

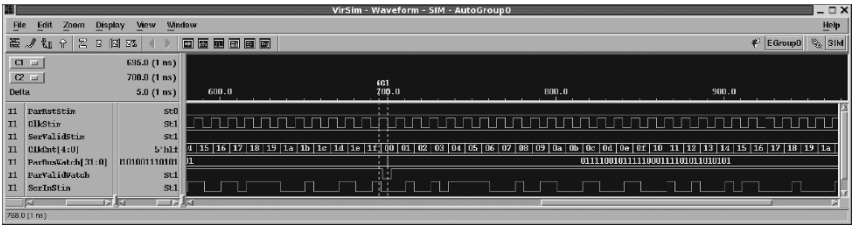


Fig. 13.6 Generic deserializer close-up of the serial bit counter wrap-around

Step 2. Deserialization data-stream synchronization. Modify your generic deserializer from Step 1 so that whenever the serial stream contains 12 successive `'0'` bits in a row, those bits should be rejected, and deserialization should cease. After ceasing, with any incomplete word data (prior to the 12 0's) saved, the device then should wait until 12 `'1'` bits in a row arrive; after the 12th `'1'`, the saved data should be revived and deserialization should resume with the shift register where it left off.

One good approach to Step 2 would be to start by just shifting in a stream of bits and identifying the two bit-patterns by setting two flags, `Found_stop` and `Found_start`.

After you have simulated identification correctly, create a parallelizing register and copy incoming bits to it until `Found_stop` is asserted. After `Found_start` subsequently is asserted, continue copying. The code asserting `Found_stop` could deassert `Found_start`, and *vice-versa*. Every time 32 bits have been parallelized, offload them to an output bus and set a `parallel_valid` flag.

There are numerous different possible ways of implementing this design. Probably, there should be a shift-register which continues shifting while deserialization to the parallel bus is stopped.

This design also should be simulated (Fig. 13.7) but not synthesized.

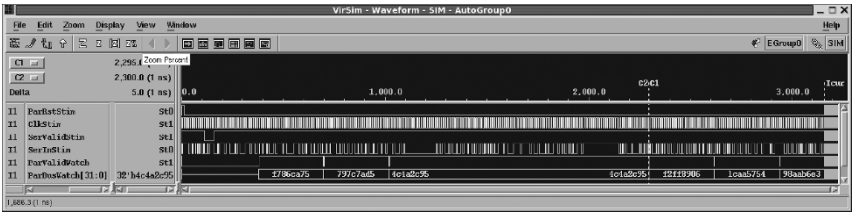


Fig. 13.7 Synchronizable, but unsynthesizable, generic deserializer, as described in the lab text

Step 3. Deserialization Frame Decoder. We return now to our serdes project.

To establish data framing, we'll allow, optionally, a less rigorous criterion than when we locked in the PLL in *Lab 10*:

As a *synchronization criterion*, the current packet may be considered locked in on the data byte received immediately after a pattern of `8'b000_00_000` has been shifted in on `SerIn`. Deserialization may be performed throughout every period of synchronization. This criterion optionally might be stiffened to allow lock in only when all four pad patterns are in the shift register.

However, regardless of synchronization, the `ParClk` should be toggled on every 16th bit received on `SerIn` following the most recent synchronization, so that there will be 4 edges and thus two `ParClk` cycles for every 64 bits received, regardless of current synchronization state.

The *synchronization loss* criterion optionally may be that the embedded, 2-bit sequential pad number (`..., 2'b11 -> 2'b10 -> 2'b01 -> 2'b00, ...`) is found to miscount, or just that the first pad (`pad_00`) can't be located 64 bits after the previous one. Either way, synchronization loss should have no effect on `ParClk`; the receiving `PLL` clock is free-running unless commanded to (re)synchronize.

Deserialization should be stopped during desynchronization, and incomplete data packets should be discarded. Deserialization should resume with the first data byte following resynchronization by either criterion above.

It is understood that resynchronization might cause a `ParClk` glitch. But, the `DesDecoder` can be designed not to malfunction because of such a glitch; and, for other design blocks on the digital side, that's one reason we have a FIFO in this deserializer.

In summary, the `DesDecoder` block will be supplied a digital data stream clocked in by an externally-generated signal, `SerClk`, but this clock will not be made available to the deserializer. The `SerClk`, and a simulated `SerIn` data stream, using our 64-bit packet-framing protocol, should originate in a testbench module in the lab exercise. See Fig. 13.8.

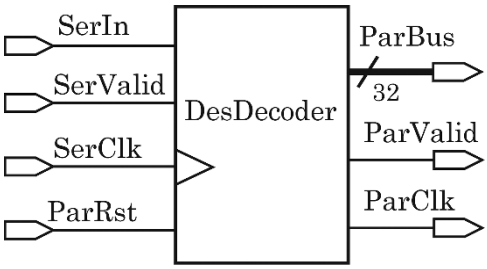


Fig. 13.8 Deserialization Decoder: First-cut block diagram

So, implement the `DesDecoder` to extract a nominal 1 MHz clock from the serial data stream and use it to clock out the properly-framed, parallelized data in 32-bit words. Do not consider PLL functionality; just provide a 32 MHz serial clock input along with the stream of serial data: `SerIn` = serial data; `SerClk` = serial input clock; `ParClk` = output clock, extracted from the serial input stream; `ParBus` = 32-bit output parallel word, clocked by `ParClk`.

To complete this Step, use anything you wish from *Lab 10*, or previous work, and implement the deserialization decoder block as shown here. Keep in mind that it may be simpler to start this design from scratch and only copy small fragments of your previous work. In particular, the shifting of the shift register can be much simpler here than for the Step 2 problem of this lab.

Consider breaking down the problem into several small tasks, setting up the calling of these tasks, and then implementing the bodies of the tasks.

Simulate your result, only (see Figs. 13.9 and 13.10); we shall synthesize this subblock of our serdes later in the course.

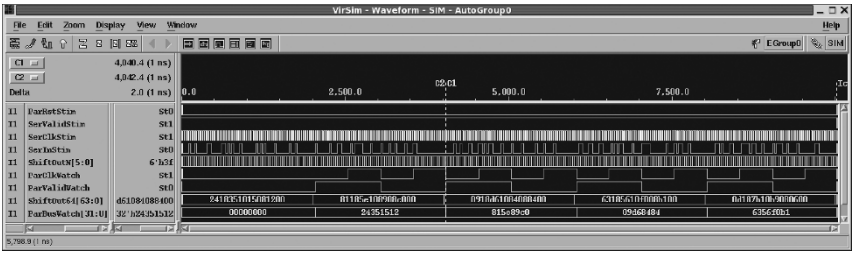


Fig. 13.9 The deserialization decoder (`DesDecoder`), not yet synthesizable

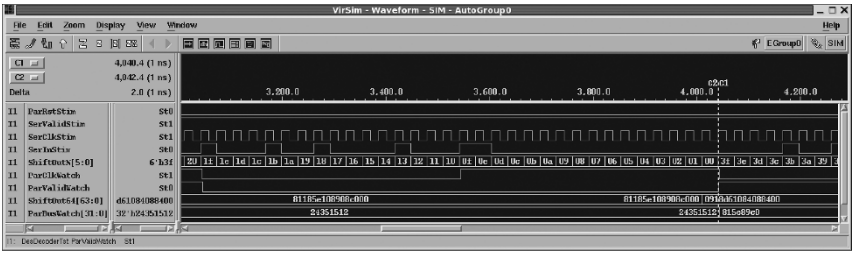


Fig. 13.10 The `DesDecoder`, zoomed in on the serial bit counter wrap-around

13.3.1 Lab Postmortem

Think about problems in changing the packet width (to 128 bits, 36 data bits with parity), etc.

13.3.2 Additional Study

Reread this lab's instructions and review previous labs referenced in this one.

Chapter 14
Week 7 Class 2

14.1 UDPs, Timing Triplets, and Switch-level Models

We again put aside our serdes project to study some verilog in depth. This time, we shall look into the basics of the design of small devices at and below the gate level of complexity.

14.1.1 User-Defined Primitives (UDPs)

The UDP `primitive` is a verilog feature which gives a user a way to create SSI (Small-Scale Integrated) device models which simulate quickly and use little simulator memory. The target component for a UDP is any specialized kind of latch (flip-flop) or a complex combinational gate.

UDPs exist at the same design level as modules, and they have no functionality that a module can not have. Usually, several UDP models will be placed in one library file; they may coexist in such a file with models implemented as `modules`. However, often, these `primitives` are instantiated in a `module` wrapper to give them timing, multiple outputs, or other essential functionality.

UDPs are allowed to have only one output and any number of inputs; the verilog Std 1364 specifies a limit of at least 9 inputs for combinational UDPs and at least 10 for sequential. Every I/O must be one bit wide. The functionality is by a look-up table.

UDP's are not synthesizable.

The verilog structure of a UDP is: `primitive` keyword, UDP name, I/O declarations, a `reg` declaration if the device is sequential, (traditionally) an initialization block, and finally a `table` defining the logic. If an ANSI header is used, initialization may be done in the header; however, here we shall initialize only in the `initial` block. The `table` columns are somewhat different for combinational vs. sequential UDPs.

In exchange for simplicity and speed during simulation, a UDP may not include delays, '`z`' states, or bidirectional (`inout`) ports. A '`z`' on a UDP input is handled

internally as an 'x'. Modern library components may be difficult to model as UDP's because of the lack of delay or other technology-dependent functionality.

A typical `table` row for a combinational UDP has this organization:

(*inputs* in declared order) : (*output*) ;

For example, three UDP `table` rows for a three-input *and* gate:

```
table
...
1 0 0 : 0;
1 1 1 : 1;
1 1 x : x;
...
endtable
```

Of course, verilog already includes a primitive representing an *n*-input *and* gate; a more practical UDP implementing an *and-or* combination would be as follows:

```
primitive u2And1Or(output uZ, input uA1, uA2, uOr);
//
// Models uZ = (uA1 & uA2) | uOr.
//
table
// Output on right; inputs in declared order:
//  and'ed inputs      or'ed input
//    uA1 uA2            uOr        uZ
       0   0              0     :    0;
       1   0              0     :    0;
       0   1              0     :    0;
       1   1              ?     :    1;
       ?   ?              1     :    1;
endtable
endprimitive
```

Note on terminology: Compound combinational gates often are included in an ASIC library. Whether or not implemented as UDP's, they are named according to their functionality and number of logic-term inputs: "A" for *and*, "O" for *or*, "I" for inversion, etc. Consistent naming facilitates library maintenance. For example, a cell evaluating `(A&&B)||C` might be named, ***AO21***. A cell evaluating `!((A&&B)||(C&&D&&E))` might be named, ***AOI23***.

For a sequential UDP, there are two columns in the `table` delimited by colons. The left column, surrounded by colons on both sides, represents the current state of the one, 1-bit, storage register allowed and maps to the output port; the rightmost column represents the next state, given the current state and all inputs.

For example, here is a UDP which may used to implement a D flip-flop:

```
primitive uFF(output reg uQ, input uD, uClk, uRst);
//
initial uQ = 1'bx; // Not same as a module initial block.
//
table
// Output on right; inputs in declared order:
//                current next
// uD uClk uRst   uQ      uQ
   0  (01)   0 :   ?   :  0  ;  // Clock in 0
   1  (01)   0 :   ?   :  1  ;  // Clock in 1
   0  (0?)   0 :   0   :  0  ;  // Default to keep same 0
   1  (0?)   0 :   1   :  1  ;  // Default to keep same 1
// Unclocked:
   ?  (1?)   0 :   ?   :  -  ;  // Ignore negedge.
  (??) ?     0 :   ?   :  -  ;  // Retain state.
// Reset asserted:
   ?   ?   (01):   ?   :  0  ;  // Posedge reset
   ? (??)    1 :   ?   :  0  ;  // Ignore clock edge
  (??) ?     1 :   ?   :  0  ;  // Ignore clock state
   ?   ?     1 :   ?   :  0  ;  // Ignore clock state
endtable
endprimitive
```

The `table` row entries in any UDP should be separated horizontally by whitespace for readability, but they need not be. UDP's derive from the days in which it was time-consuming work for a workstation computer to parse a few extra blank characters as spacers – and designers, used to punching and collating Hollerith cards to enter a program, couldn't read the input very well if they wanted to. Anyway, each row ends with a semicolon (';').

Edge sensitivity in a sequential UDP is described by the two states of an edge, written inside parentheses, initial state to the left; thus, "`(01)`" represents a rising edge, and "`(10)`" a falling one. Each `table` row is allowed only one edge. When an input change affects both a level and an edge column, the edge is evaluated first, then the level; this means that the level prevails in the event of a conflict. As in a `case` statement, '`?`' means a don't-care input. A hyphen ('`-`') means no change on an output.

The UDP table should be complete, because a state or edge definition in one row opens up possibilities for the simulator to misinterpret any other possible permutation of the values; thus, it is important to add a row for every possible foreseeable event. This becomes complicated for sequential UDPs, in view of the edge possibilities.

Several more rows would have to be added to the UDP above, if it became necessary to include '`x`' output states selectively.

Like the verilog builtin primitive gates (`and`, `or`, etc.), UDPs may be instantiated without an instance name. An instantiation may include a delay expression.

To summarize what we have said about UDPs:

UDP's are look-up table-defined primitives. Keyword is **`primitive`**.
Structurally interchangeable with **`module`**s.
Advantages:

- May be instantiated without an instance name.
- Accept delays when instantiated.
- Fast compilation and simulation.

Limitations:

- Allowed only one, 1-bit output.
- Only 1-bit inputs (up to at least 9 of them).
- No timing or parameter declaration.
- No `specify` or other internal block.
- No '`z`' or bidirectional functionality.

Not used in VLSI design.
Not synthesizable.
Sometimes used in verilog simulation library development.

14.1.2 Delay Pessimism

We move on, now, to a discussion of verilog delays. It is important to understand that ***functionality*** and ***timing*** are almost-orthogonal features of a simulation.

Functional edges. When a '`1`' level occurs after any other level, including an '`x`', the functionality is that of a rising edge, and an `always` event expression or any other edge expression will treat it functionally as a `posedge` event. Likewise, when a change results in a '`0`', it is functionally a falling edge, and functionally it means a `negedge` event.

Timing edges. When a (*rise*, *fall*) timing expression is associated with a change from '`0`' to '`1`', the rising-edge (`posedge`) delay will be given by the *rise* value. The `negedge` delay for a change from '`1`' to '`0`' will be given by the *fall* value. But, this correspondence with functional edges ceases when '`x`' (or '`z`') levels are involved.

When the change is ***to*** an '`x`' level, neither the *rise* nor the *fall* in a (*rise*, *fall*) or (*rise*, *fall*, *to_z*) expression has a specific meaning. Instead, the smallest available delay value is used by the simulator. This quick change to an unknown lengthens the duration of unknown states and thus is called "pessimistic" in regard to knowing the hardware value, which must be '`1`' or '`0`'.

When the change is ***from*** an 'x' level, the largest available delay value is used, again lengthening the duration of unknown states, and, thus, again, being "pessimistic".

Thomas and Moorby (2002) Table 6.6 (section 6.5.2) also explains delay pessimism.

14.1.3 Gate-Level Timing Triplets

In *Week 4 Class 1*, we saw how strength might be assigned to a gate output by putting one or two strength keywords in parentheses. For example, an NMOS *or* gate:

```
or (strong0, weak1) or_01(out_or, in1, in2, in3, in4);
```

We also have seen the same approach to assign delays, except that the delay values were preceded by a '#'. For example,

```
or #2 or_01(out_or, in1, in2, in3, in4);
```

Parentheses are optional around the single delay value. If both strength and delay are assigned, the strength specification is to the left of the delay specification:

```
or (strong0, weak1) #(2,1) or_01(out_or, in1, in2, in3, in4);
```

As we have seen, delay values also may be assigned in multiples of two or three in parentheses. The interpretation of such values is as follows:

1 value: `#t inst_name(...);`	Every change on the output(s) is scheduled with this delay.
2 values: `#(tr,tf) inst_name(...);`	Every rise is scheduled with the first delay value *tr*; every fall with the second delay value *tf*.
3 values: `#(tr,tf,tz) inst_name (...);`	The first two are the same as for 2 values; the third delay is the delay to 'z' state for gates which are capable of it.

To repeat the verilog delay scheduling rules, when selecting multiple delay values, as for wires, a *rise* in the table above *only* is a change from '0' to '1'; a *fall only* is a change from '1' to '0'. For changes involving 'x' or 'z', the values specified are used "pessimistically" by the simulator: When only two values are given and the gate is capable of 'z', the delay to 'z' is the shorter of the two. In all cases of two or more values, the recovery from 'z' or 'x' is the longest value, and the delay to 'x' is the shortest value.

A Second Dimension of Delay. In the old days, when designs were mostly board-level assemblies of relatively small digital IC's and discrete passive components, logic simulators had to account for temperature, supply voltage, and fabrication differences across the board. Minimum and maximum delays were estimated separately for each chip. The resulting timing differences were simulated by assigning 'x' wherever the simultaneously-estimated minimum vs. maximum delays allowed it (=pessimistic). See Fig. 14.1.

However, most modern designs use VLSI IC's structured in blocks with latched (clock-synchronized) outputs, so this kind of pessimism has not been useful during simulation, even in recent deep-submicron designs. The chip itself tends to be more or less uniformly fast, typical, or slow; so, only one condition or extreme has been applicable in any one simulation evaluation. The min-max approach, however, usually is used during *static timing verification*, which we shall touch upon briefly later.

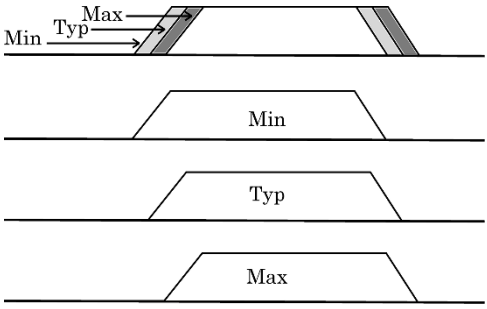


Fig. 14.1 Board-level simulation represents edge delay spreads as unknowns (top); IC-level simulations are run with no spread and each delay state separately (lower three). The sloping edges represent individual uncertainty intervals caused, for example, by skew or jitter

In recent VLSI designs, with pitch down to about 90 nm, the simulator has not been used to choose the condition, best, typical, or worst case; the designer has made this choice. Gates in libraries for logic synthesis still have to be characterized for temperature, supply voltage, and process variations, so there still is a need for a simulator to handle multiple possible delays on a gate. But now, devices being simulated almost always are assigned just one, single, specific, global range of delay, *minimum*, *typical*, or *maximum*, representing the expected global uniformity of a given, operating IC. The simulation then is repeated several times, each time with a different global alternative, *min*, *max*, or *typ*. VCS, as other verilog simulators, is told which global value to use by setting a command-line option when it is invoked. The default is *typical*.

There is some indication that verification of very large designs (90 nm and below) may require two, or all three, of each triplet to be used in a single simulation run in order to display local uncertainty of timing in a simulation waveform. This may become necessary, for one reason, because of the temperature differences which can develop across a chip as a function of operating time or mode of operation. The more that is put on a chip, the more likely that different functional blocks will be operating under different conditions. Thus, the old board-level simulation displays, showing a range of unknowns around every edge, again may become common.

Regardless of simulator design or practice, in the verilog, to assign these triplet values to a gate output, they simply are substituted, separated by colons, for the single values which we have been using up to now. So, when indicated by characterization of the synthesis library, the table above may be modified to replace ***t*** with ***t_min*:*t_typ*:*t_max***; and, each of ***tr***, ***tf***, and ***tz*** in that table above then will be triplicated correspondingly.

The first *or* gate example above then may be changed to,

```
or #(1:2:3) or_01(out_or, in1, in2, in3, in4);
```

Likewise, if a `bufif1` were used to model a gate with a relatively slow turnoff, instantiation might require a delay specification given by,

```
bufif1 #(1:3:4, 1:2:4, 6:7:8) triBuf_2057(OutBit, InBit, CtlBit);
```

The simulator does not impose or require order in the values in a triplet; in principle, one might find `min:typ:max` specified with `min` $\geq$ `typ` $>$ `max`.

This isn't all there is to the calculus of delay in verilog; but, we shall put off internal delays in library components and other devices until later in the course.

14.1.4 Switch-Level Components

We introduced the different verilog strengths, and some of the different types of nets, in *Week 4 Class 1*. The net types and built-in gate types were studied further in *Week 5 Class 2*. We now shall go beyond this in modelling devices at the *switch level*, which is to say, at a level in which gates are treated as made up of individual substructures (transistors) which may be simulated as switching on and off.

To model at the switch level, we require switch-level primitives. These are supplied in verilog as the *MOS*, *CMOS*, *bidirectional*, and *source* switches.

Here is a list of all the available switch-level primitives:

MOS switches:
`nmos, rnmos` (**like** `bufif1`)
`pmos, rpmos` (**like** `bufif0`)
`cmos, rcmos`
Bidirectional pass switches:
`tran, rtran`
`tranif1, rtranif1`
`tranif0, rtranif0`
Switch-level net:
`trireg`
Power sources:
`pullup, pulldown`

MOS Switches. MOS stands for Metal-Oxide-Silicon, the main layers used to fabricate devices in this technology. It represents an advance in semiconductor technology over the more power-hungry bipolar (P-N junction; current-operated) semiconductors. MOS transistors have a source-drain potential which supplies energy for amplification, and a gate which turns source-drain current on or off electrostatically.

A P-channel transistor (`pmos`) conducts more current when its gate is more negative; an N-channel transistor (`nmos`) when its gate is more positive. Furthermore, P devices, which have hole majority carriers, are slower than N devices of the same size; so, in CMOS technology, the P side generally is made larger than the N side to make the response speeds more nearly equal. Thus, CMOS P devices are fabricated to operate geometrically nearer the high supply voltage rail than ground, because, there they can be operated with delay comparable to that on the N side. Anyway, for a variety of reasons, P devices are fabricated near the *supply1* rail and N devices nearer the *supply0* rail. Because of this device gravitation, the elementary logic gates in CMOS technology tend to be `nand` and `not` rather than `and` and `buf`.

The MOS primitives are **`nmos`**, **`pmos`**, **`rnmos`**, and **`rpmos`**. The **`r`** means "resistive", and the **`r`*** primitives are meant to represent physically higher-impedance devices than the others.

All the MOS devices individually are functionally identical to the `bufif1 (nmos)` or `bufif0 (pmos)` primitives we have used already in lab, except that the `r*` primitives have outputs which always reduce the strength applied at their inputs. Only `small`-capacitor or `highz` strengths are not reduced. See *Week 4 Class 1* for a table of verilog strengths. The `rnmos` and `rpmos` strength-changing rules are given in section 10.2.4 of Thomas and Moorby (2002), or in IEEE Std 1364, section 7.12, as follows:

***Resistive* MOS Strength Rules**

Strength		Strength Keyword	
In	**Out**	**In**	**Out**
supply	pull	`supply0/1`	`pull0/1`
strong	pull	`strong0/1`	`pull0/1`
pull	weak	`pull0/1`	`weak0/1`
large cap	medium cap	`large0/1`	`medium0/1`
weak	medium cap	`weak0/1`	`medium0/1`
medium cap	small cap	`medium0/1`	`small0/1`
small cap	small cap	`small0/1`	`small0/1`
highz	highz	`highz0/1`	`highz0/1`

CMOS Switches. The name stands for Complementary Metal-Oxide-Silicon. These devices are composed functionally of paired `nmos` and `pmos` transistors, just as are actual CMOS gates fabricated on a chip. A **`cmos`** switch even has two control inputs, just as a CMOS transistor on a chip would have two gates. Of course, `rcmos` switches are composed functionally of paired `rnmos` and `rpmos` switches.

Verilog assumes a positive-logic regime (‘1’ = higher voltage; ‘0’ = lower), so an N device turns on when its gate is at logic ‘1’; a P turns on when its gate is at logic ‘0’.

There are limits to the accuracy of a digital simulator at this level. For example, a `cmos` switch can be imagined to represent two MOS transistors in parallel. This does a fine job as a switch level model of a `bufif1`, as shown in Fig. 14.2.

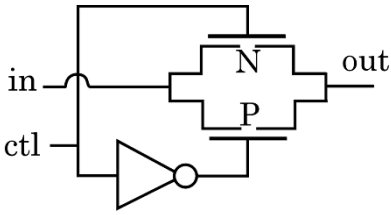


Fig. 14.2 CMOS `bufif1` modelled as `nmos` and `pmos` in parallel

But, in verilog, the `nmos` and `pmos` primitives pull up their outputs (to ‘1’) with the same strength as they pull down (to ‘0’); actual P vs. N devices differ in strength at the two logic levels (N pulls low stronger than high, and *vice-versa* for P).

Section 10.2.1 of Thomas and Moorby (2002) gives a nice switch-level model of a static RAM cell, and there is a model of a shift register in section 10.1, but there is some question as to what the purpose might be of such models, when a device of any comparable size can be modelled easily in SPICE, with extremely high accuracy.

Even so, CMOS devices are much closer to being symmetrical than individual PMOS or NMOS devices; for this reason, they are preferred in modern designs.

In this vein, consider the two different arrangements of transistor elements which could implement an inverting buffer (verilog `not`) on a chip. Switch-level modelling does a fine job. By tying a `pmos` input to `supply1`, and an `nmos` input to `supply0`, it is possible to invert the control input, which is just what an inverting buffer does at the switch level.

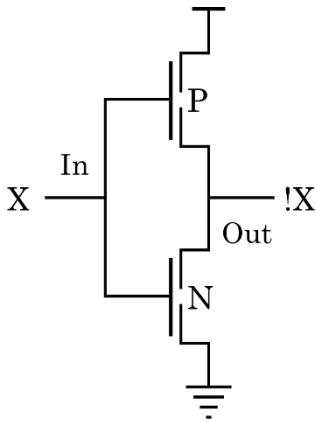


Fig. 14.3 CMOS `not` gate modelled as `nmos` and `pmos` in series

Looking at the CMOS `not` gate model in Fig. 14.3, one might think that by reversing the N and P devices, the gate would become a noninverting buffer. This could be done, and the logic indeed would be noninverting; however, a P device operating near `supply0` is extremely slow and inefficient; likewise an N device operating near `supply1`. Thus, for these technical reasons, simple noninverting buffers never are used in CMOS technology. If one requires a noninverting buffer in CMOS, the usual solution is to drive the input of a big inverting buffer with the output of a small one, the two inversions yielding a net noninversion.

The verilog `cmos` primitive approximates reality better than either `pmos` or `nmos`, in that a `cmos` *also* does not reduce the strength of an input (except that `source` becomes `strong`). Because a `cmos` has two control inputs, it has a different port definition than a `bufif`*x*. The ports declared for a `(r)cmos` are as follows:

```
(r)cmos  optional_inst_name ( out, in,  N_ctl,  P_ctl)
```

In the event that one control is off and the other on, a `cmos` still will turn on; so, the second control presumably is intended to model requirements for on-chip connectivity, rather than to model functionality.

Bidirectional Pass Switches. These primitives model charge-transfer gates (pass transistors) and thus are called **`tran`** (Fig. 14.4), **`tranif1`**, and **`tranif0`**, with the corresponding high-impedance **`rtran`**, **`rtranif1`**, and **`rtranif0`**. They are not allowed to be assigned delays. Unlike `trireg` primitives, they don't store state. They merely pass a logic level applied at one terminal to the other terminal. They are used to model nets of transistors connected in arbitrary ways.

Fig. 14.4 Schematic symbol of a `tran` switch

Instantiation of a `tran` transfer gate follows the same format as `buf`, but with exactly two ports, both `inout`. A `tranif`*x* has two `inout` ports and a third, control port; thus, it is analogous to a `bufif`*x*, except that it is bidirectional. More information on applications of transfer gates may be found in the layout-related References and in the Additional Study suggestions below.

Source Switches. There are two of them, `pullup` and `pulldown`. Each accepts just one output net as argument. The `pullup` drives its net high with constant `source1` strength; the `pulldown` drives it with `source0`. Whereas the transistors being modelled in verilog at switch level usually are assumed to operate in enhancement mode (normally off), these would be assumed to be large transistors in depletion mode (normally on), or direct connections to an IC power or ground rail.

14.1.5 Switch-Level Net: The `trireg`

The functionality of the `trireg` net type is described in IEEE Std 1364, sections 7.13.2 and 7.14. In a word, a `trireg`, like a capacitor modelled as a time-limited `reg`, just stores a logic state.

A `trireg` is unique in that it makes special use of the `small`, `medium`, and `large` strength values. For a `trireg`, these strengths are meant to represent the size of a capacitor which stores charge whenever the drive to the `trireg` enters the high-impedance (‘`z`’) state. However, a `trireg` driven at ‘`z`’ never enters the ‘`z`’ state; instead, it holds its last non-`z` state. If and only if the `trireg` has been assigned a delay, this last non-`z` state decays from ‘`1`’ or ‘`0`’ to ‘`x`’ as soon as the delay lapses.

The strength of a `trireg` is used to determine which of several `triregs` in contention will determine the delay to ‘`x`’; this is the only use of the charge strengths, `small`, `medium`, and `large`. When strengths are equal, the rule of pessimistic simulation is used to determine `trireg` decay to ‘`x`’ just as it is to determine the result of other timing conflicts.

A delay value may be assigned to a `trireg` net when the net is declared. The delay format is different in one way from that of a three-state component: When three delay values are given, the first two refer to rise and fall, as for a three-state component; however, the third value refers to time to ‘`x`’, not time to ‘`z`’. A `trireg` can not enter a ‘`z`’ state, although it could be initialized to one. A `trireg` with a delay value decays to ‘`x`’ after the given delay as soon as the net’s last driver has turned off (to ‘`z`’).

If a `trireg` net has no delay associated with it, it continues forever to drive its output(s) at the strength declared for it, at the logic level with which it last was being driven (‘`1`’, ‘`0`’, or ‘`x`’). Example of a `trireg` declaration:

```
trireg (medium) #(3, 3, 10) medCap1;
```

Connectivity of `trireg` nets may be established by continuous assignment, or by port-mapping to switch-level component instances.

Here is an example of the use of `trireg` nets:

```
pullup(vdd);
trireg (small) #(3,3,10) TriS;
trireg (medium)#(6,7,30) TriM;
trireg (large) #(15,16,50) TriL;
// Pass transistor network:
tran (TriM, TriS); // left always wins.
tran (TriM, TriL); // right always wins.
// NMOS network:
rnmos #1 (TriM, TriS, vdd); // input has no effect.
rnmos #1 (TriS, TriL, vdd); // input controls output.
rnmos #1 (TriM, TriL, vdd); // Contention on output.
```

14.2 Component Lab 17

Do the work for this lab in the `Lab17` directory.

Lab Procedure

Keep in mind that many VLSI simulators will not simulate switch-level verilog correctly; Silos generally will work, except for resistive primitives.

Step 1. Combinational UDP. Design a UDP in a module named `AndOr2Not4` which evaluates this logic function: `X = ( ~a | ~b ) & ( ~c | ~d )`, in which `a - d` are input names, and `X` is the output name.

SSI discrete component databooks or large ASIC libraries might include such a device.

Suggestion: Include a verilog comment line naming the table columns to help reduce entry errors.

Instantiate your component in a testbench module and simulate it to verify functionality.

Step 2. Sequential UDP. Design a UDP named `AndLatch` which functions as a simple transparent latch but has two data inputs `and`ed together before being latched.

Instantiate this latch in a module and simulate it to verify functionality.

Step 3. Switch-level model of an inverter. Create a module named `Nottingham`. Give it one 1-bit input and two 1-bit outputs. Combine a `pmos` and `nmos` primitive as described in the presentation above (Fig. 14.3) to model a `not`. This amounts to a CMOS inverter. Drive one of the two outputs with this composed `not`.

Instantiate a verilog `not` gate and use it to drive the other `Nottingham` output. This will allow you to compare a verilog `not` with your `nmos-pmos not` in a simulation, both driven by the same input.

You could test such a design by feeding both outputs to the inputs of an `xor` gate: A '`1`' on the `xor` output would indicate that the two gates were not identical logically.

Step 4. The `cmos` control inputs. Add a third output to the module in the previous Step and connect a `cmos` to drive it from the module input. Declare local control nets routed through new I/O's from your testbench.

Simulate to fill in the `cmos` truth table below. Can you predict what the output will be under each of these conditions? ("*unconn*" = '`z`' state)

cmos Truth Table			
out	**in**	**n-ctl**	**p-ctl**
	1	1	1
	1	1	0
	1	0	0
	1	0	1
	0	1	1
	0	1	0
	0	0	0
	0	0	1
	1	x	x
	1	x	0
	1	x	1
	1	1	*unconn*
	1	*unconn*	0

Step 5. Pass transistor mux model. The easiest demonstration of pass transistor logic is a multiplexer: The select input is decoded to turn on just one pass transistor; the turned-on logic level then is transferred to the output, where it wins the contention against the other output(s), which must be in 'z' state.

For a 2-input mux, all it takes is one `tranif1` and one `tranif0` with outputs tied together, and a one-bit select input to both control inputs.

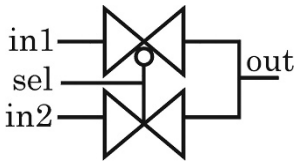


Fig. 14.5 Two-input mux modelled by `tranif`'s

The relevant verilog for the 2-input mux design shown in Fig. 14.5 would look something like this:

```
tranif0 UpperTran(Out, In1, Sel);
tranif1 LowerTran(Out, In2, Sel);
```

For this exercise, create a module named `TranMux4` and design a 4-input mux for it, using nothing but pass transistors and nets. Simulate to verify your design.

After verifying your design, replace the `tranif`*x* switches with `rtranif`*x* switches and simulate again. (This may not work in Silos).

Step 6. `nand` and `nor` gates. Create a module named `Nand_Nor`. Give it three inputs and two outputs. The outputs should be named `Nand` and `Nor`. Use the schematic of Fig. 14.6 to enter a switch-level model of a 3-input *nand* gate and a 3-input *nor* gate. Simulate to verify your design (see Fig. 14.7).

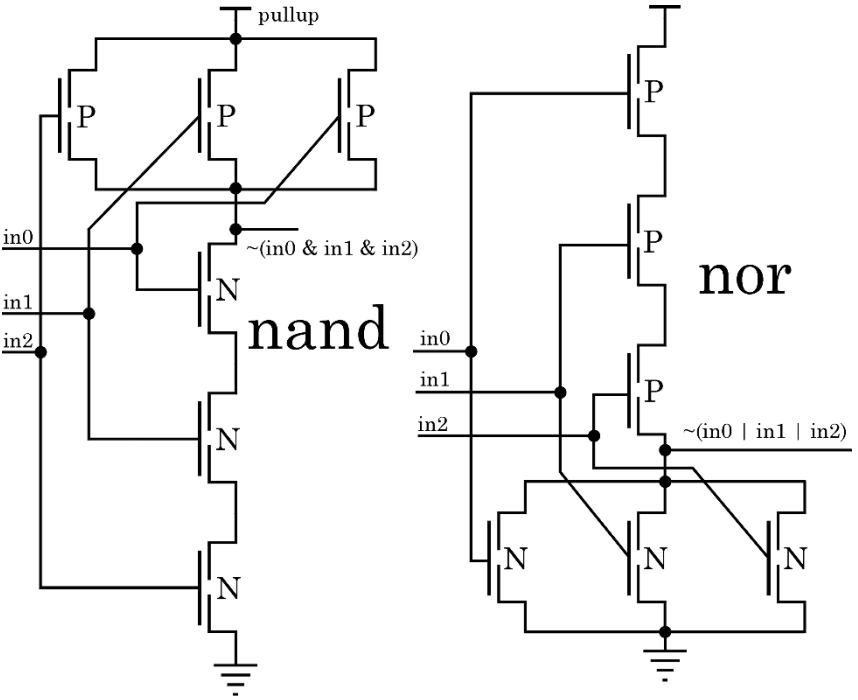


Fig. 14.6 A CMOS 3-input `nand` gate and a 3-input `nor` gate

After this, add a CMOS inverter on each output to change the functions to *and* and *or*.

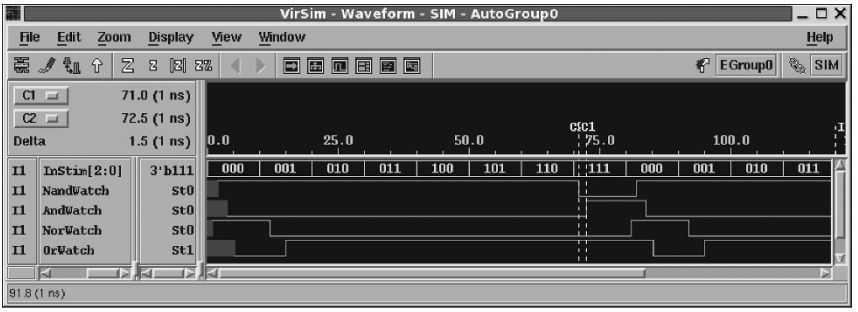


Fig. 14.7 A `Nand_Nor` switch-level simulation

Step 7. Trireg pulse filter. Create a new module `RCnet` and use `trireg` switches as capacitors to implement the RC network shown in Fig. 14.8.

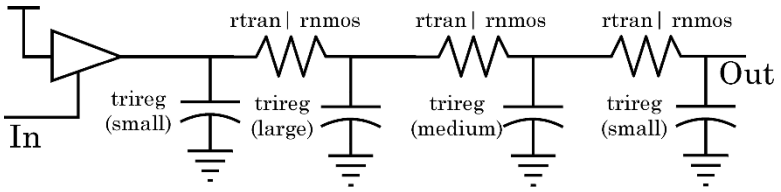


Fig. 14.8 RC network for digital simulation

Use an enabled `rnmos` to represent each resistor shown, *in* on left; *out* on right. A `large trireg` gets a `#(15,15,50)` delay, a `medium #(7,7,30)` and a `small #(3,3,10)`. Simulate an input pulse to see (Fig. 14.9) how well these digital switches approximate analogue functionality.

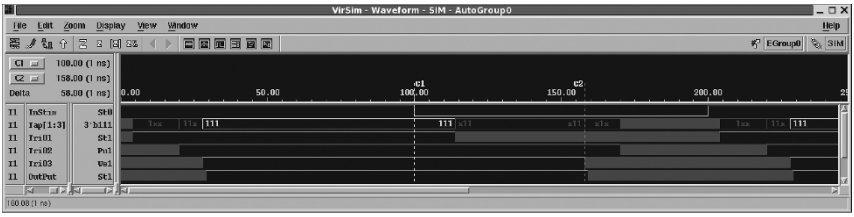


Fig. 14.9 Simulation of a delay line using enabled `rnmos` resistor models

Replace each `rnmos` with an `rtran` to specify delays but with less interesting strength (Fig. 14.10).

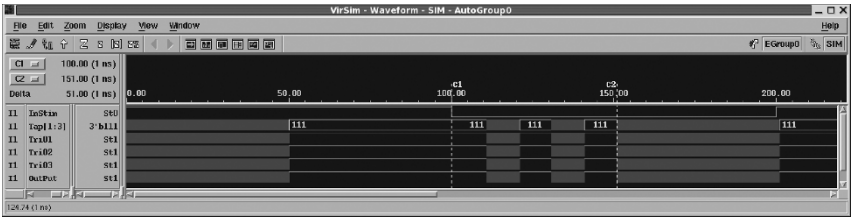


Fig. 14.10 Simulation of a delay line using enabled `rtran` resistor models

14.2.1 Lab Postmortem

What do the three delay values mean when specifying delay on the output of a `trireg`?

Can an `rtran` be used to simulate a delay?

14.2.2 Additional Study

Read Thomas and Moorby (2002) section 6.5.2 on delay conflicts and pessimism.

Read Thomas and Moorby (2002) sections 6.5.3 and 6.5.4 on time units and timing triplets.

(Re)read Thomas and Moorby (2002) chapter 10 on switch-level modelling. However, ignore the "minisimulation" code.

(Optional) Read Thomas and Moorby (2002) chapter 9 on UDP's.

Optional Readings in Palnitkar (2003)

Read chapter 5.2 on the basics of gate-level delays.

Read chapter 11 on switch-level modelling.

Read chapter 12 on UDP's. Notice especially section 12.5 which summarizes features of UDP's as contrasted with modules. Do section 12.7, problem 1, a 2-1 mux. Compare your result with the solution on the Palnitkar CD.

Study the section 11.2.3 model of a switch-level latch or flip-flop. There is a model of a flip-flop named `cff.v` on the Palnitkar CD; simulate it to see how it works.

Chapter 15
Week 8 Class 1

15.1 Parameter Types and Module Connection

15.1.1 Summary of Parameter Characteristics

- Parameters are unsigned integer constants by default.

```
parameter Name = value;
```

- May be declared signed (but many tools reject it).

```
parameter signed Name = -value;
```

- May be declared real (but not synthesizable and often rejected).

```
parameter real Name = float_value;
```

- May be typed by vector index range.

```
parameter[6:0] Name = 7_bits_of_value;
```

- Declaration allowed anywhere in module, but `localparam` preferred in body.

```
localparam Name = value;
```

- Must be assigned when declared. Width defaults to enough for the value assigned.

15.1.2 ANSI Header Declaration Format

```
module ModuleName #(parameter Name1 = value1, ...)( port decs);
```

In the module header, we have advocated only the ANSI declaration of parameters, with pass of value by name. For example,

```
// Declaration:
module ALU #(parameter DataHiBit=31, OutDelay=5, RegDelay=6)
            (output[DataHiBit:0] OutBus, ...);
 ...
// Instantiation:
ALU   #(.DataHiBit(63), .RegDelay(7)) // OutDelay gets the default.
  ALU1 (.OutBus(ResultWire), ...);
 ...
```

15.1.3 Traditional Header Declaration Format

```
module  ModuleName (PortName1, PortName2, OutPortName1, ...);
  parameter  Name1 =  value1; ...
  direction[Name1:0]  PortName1;   // direction = output, input, inout.
  direction[range]  PortName2; ...
  reg[range]  OutPortName1; ...    //  output assigned procedurally.
  ...
```

The traditional module header ends with the first semicolon, just as does the ANSI header. However, the rest of the traditional header does not end at any well-defined place in the module. This makes specification of the module interface error-prone and sometimes ambiguous.

15.1.4 Instantiation Formats

ANSI and traditional instantiation formats are identical.

15.1.4.1 Instance Parameter Override By Name

ModuleName #(.*ParamName1*(*value1*),.*ParamName2*(*value2*)...)

moduleInstName(.*PortName1*(*NetName1*), ...);

15.1.4.2 Instance Parameter Override By Position

ModuleName #(*value1*, *value2*, ...)

moduleInstName(.*PortName1*(*NetName1*), ...);

Override by position is not a recommended practice.

By default, a parameter is like an unsigned constant `reg`. When such a parameter is assigned to a variable, or used in an expression, it takes on the width and type of the destination. However, an index range may be specified when the parameter is declared, making it an object of a certain width. For example, `parameter[7:0] CountInit = 8'hff;` declares a specific width, keeping the designer aware of what happens when the parameter is assigned to a variable of a given width.

Also, a parameter, like a variable, may be declared `signed`; if so, arithmetic involving it may become signed arithmetic, if the other operand(s) also are signed types. A width or signedness modifier in a module-header parameter declaration can not be overridden, although the value may be changed in an instantiation.

Examples:

```
// Note: 377 = 12'h179; -377 = 12'he87.
parameter signed[31:0] mul_coeff = -120*Pi; // Gets -376.9911 = -377.
//
// If the next were overridden by -120*Pi, it would get 32'hffff_fe87:
parameter signed[31:0] div_coeff = 32'h0000_0179;
```

Overriding a header-assigned default during instantiation is the only recommended way of changing the declared value of a parameter.

Note: The Silos demo simulator which came with Thomas and Moorby (2002) or Palnitker (2003) may not recognize signed parameters.

15.1.5 ANSI Port and Parameter Options

Using the ANSI format, it is legal to pass values by position or by name, but never by a mixture of both. When passing by position (again, not recommended), all values to the left of each position must be provided. Example (compare above):

```
module ALU #(parameter DataHiBit=31, OutDelay=5, RegDelay=6)
            (output[DataHiBit:0] OutBus, ...);
...
ALU    #(31,5,8) // Must supply first two to change third one.
   ALU1 (.OutBus(ResultWire), ...);
...
ALU    #(.DataHiBit(31),5,8) // Illegal.
   ALU2 (.OutBus(ResultWire[63:32]), ...);
```

15.1.6 Traditional Module Header Format and Options

We briefly review here the traditional, *verilog-1995* module header format. This format is based on the old, pre-ANSI *C* language function header format ("K&R" *C*),

invented by software pioneers Kernighan and Ritchie. While obsolescent, it still is generated by many automation tools such as netlist writers or file converters. The format is essential to understand because of these tools; manual editing of a verilog netlist may be required to obtain a design which can be fabricated correctly.

The header declaration lists the names, only, of the I/O's, if any. Immediately following the header, the parameters are declared, and the directions and widths of the I/O's are specified. After that, the types of the I/O's are specified; primarily, this means that outputs not driven internally by wires are assigned to `reg` type. So, just as in ANSI format, parameter values may be used to assign widths to I/O's.

Example of traditional declaration:

```
module ALU (OutBus, InBus, Clock); // Parens & contents optional.
parameter DataHiBit=31, OutDelay=5, RegDelay=6;
output[DataHiBit:0] OutBus;
input ...
reg[DataHiBit:0] OutBus;
...
```

The more modern ANSI format avoids redundant name assignments and reduces the possibility of an entry error, so it should be preferred.

Override in instantiation is done the same way whether the header has been written in ANSI or traditional format.

15.1.7 Defparam

There is a `defparam` construct in verilog which permits override of any parameter value in the design by any module, however distant or unrelated. The syntax is just,

`defparam` *hierarchical_path_to_parameter* = *new_value*;

This is a very risky construct, and it may be removed from the verilog language standard at a later date. It makes no sense within a given module, because within any one module, the parameter itself can be assigned just as easily (and under the same allowed circumstances) as it can be `defparam`'ed.

The `defparam` is the main reason for the existence of `localparam`'s: A `localparam` is identical to a `parameter`, except that it can not be overridden by `defparam`. A `localparam` is not allowed in an ANSI module header – because it also can't be overridden there.

It is recommended never to use `defparam` in a design. Like a *goto*, or like hierarchical references in general, it tends to introduce more of defects than it does of functionality.

15.2 Connection Lab 18

Do this work in the `Lab18` directory

Lab Procedure

Step 1. Traditional port mapping. Type in and rewrite the following module as **non**`ANSItop` with its header in traditional port mapping format:

```
module ANSItop #(parameter A=1, B=3, parameter signed[4:1] List=4'b1010)
                (output[3:0] BusOut, output ClockOut
                , input[3:0] BusIn, input ClockIn
                , input[1:0] Select
                );
reg ClockOutReg;
assign #(2,3) ClockOut = ClockOutReg;
...
endmodule
```

Fill in some sort of module functionality – anything you want.

At the end of this Step, you should have two modules, in files `ANSItop.v` and `nonANSItop.v`, which will compile correctly in your simulator.

Step 2. Parameter overrides. Create a new verilog file named `ParamOver.v` and instantiate both `ANSItop` and `nonANSItop` from Step 1 *twice each* in a new module `ParamOver` (you may use `ParamOver` for all of Steps 2 – 4). For each of `ANSItop` and `nonANSItop`, override once by position and once by name as follows: Override `B` to be `20` and `List` to be `-2`.

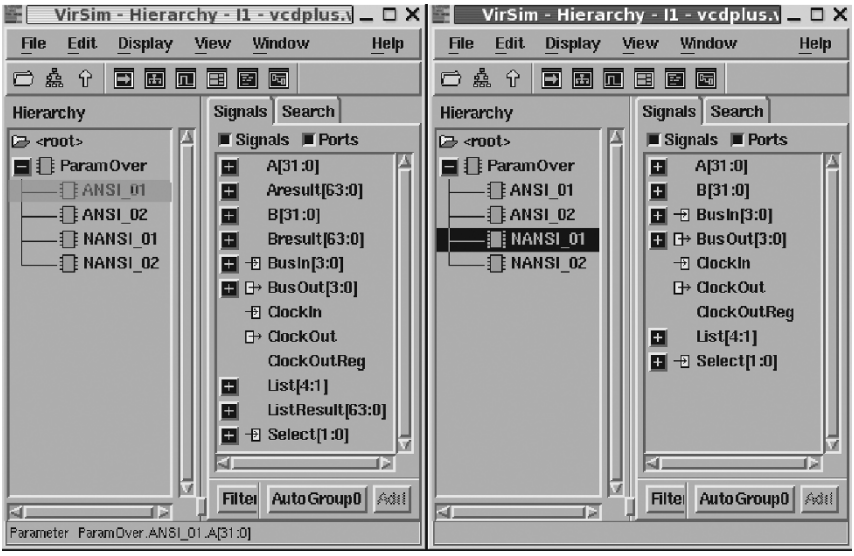


Fig. 15.1 `ParamOver` hierarchy views: Branch panes on right are for `ANSI01` (left window) and `NANSI01` (right)

View the hierarchy in the simulator (see Fig. 15.1). QuestaSim may be invoked for a look at the parameter values, because VCS doesn't display the compiled values of parameters. For best results, assign each parameter value to a 64-bit `reg` in an `initial` block in each module and use the simulator to display the `reg` value at time 0. With a width of 64 bits, the entire parameter value should be easily understood – aligned, of course, on the LSB of the `reg`. A `reg` used only in an `initial` block isn't doing anything in the design, so it will be removed during synthesis.

Side question: What was the default value of `List` in Step 1, expressed as a signed decimal number?

Step 3. Parameter width override. Instantiate `ANSItop`, overriding `A` by position to be `8'hab`. Check the result by assigning the value to a 64 bit reg in an `initial` block and using the simulator.

Step 4. Parameter type override. Instantiate `ANSItop` again, this time declaring `A` and `B signed`, but otherwise with the same default values.

Override A by name to be `8'hab` and B to be `-120*`π (don't bother to write out π exactly). See Fig. 15.2.

Check the result by assigning the value to a 64 bit `reg` in an initial block and using the simulator. Change the display formats to 2's complement to see the signed integer values.

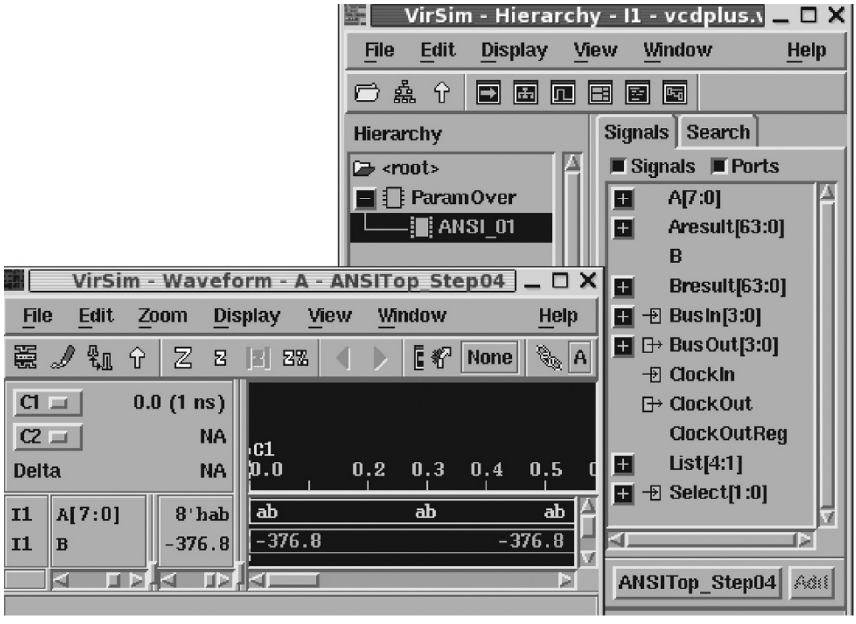


Fig. 15.2 Parameter real-valued expression in 2's complement wave display format

Step 5. `'define` and ordering problems. The text of a skeleton `HierDefine` design is located in the `HierDefine` subdirectory of `Lab18`. Change to the `HierDefine` subdirectory to find the files, which are named `HierDefine.v`, `Level2.v`, `Level3.v`, and `Level4.v`. The bidirectional data bus is configured at each level so that its width is halved each time and it is distributed differently to each instance. This might be a datapath design, for example part of a Fourier transformer.

A block diagram is given in Fig. 15.3 (see also the next Step):

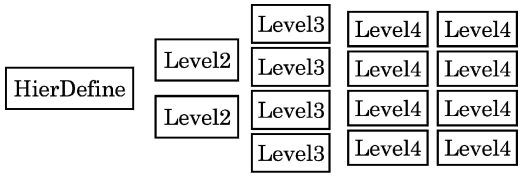


Fig. 15.3 Block diagram of the `HierDefine` instance hierarchy

A. Set up a `.vcs` compilation file in correct order which would allow VCS to compile this design for simulation. The files contain "`...`", indicating omitted functionality; you will have to comment these to compile the (nonfunctional) design (Fig. 15.4).

What happens if the order of two entries is reversed in the `.vcs` compilation file? What if `Level2` included a duplicated set of macro definitions, `'define Wid 16` and `'define ResWid 16`?

B. Suppose the design would work in different contexts if we doubled or halved all bus widths; how should `HierDefine` be modified, still using `'define`, to make such reuse convenient?

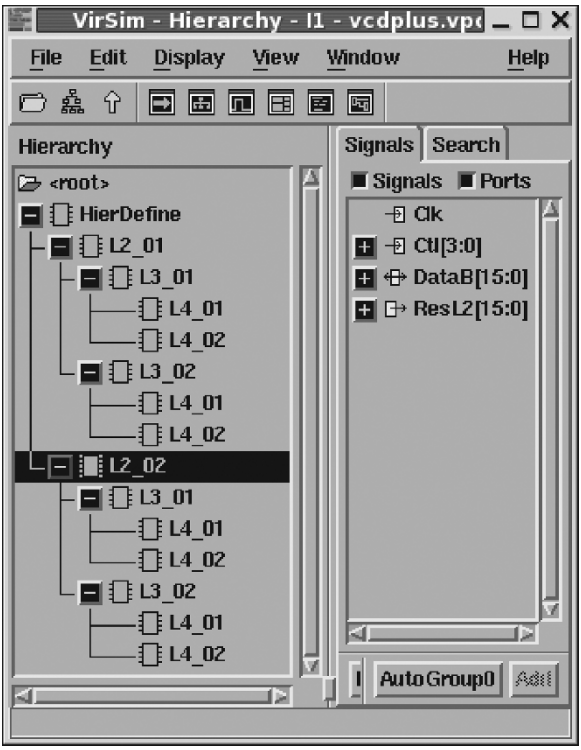


Fig. 15.4 VCS display of the `HierDefine` hierarchy

Step 6. Parameter passing in depth. Assume a design `HierParam` with 4 levels of hierarchy, an exact copy of the `HierDefine` of the previous Step, except that parameters have been declared and there are no `` `define `` assignments.

The width of `DataB` must be halved at each level of hierarchy, as implied by the structural fanout at each level. This is shown in Fig. 15.5.

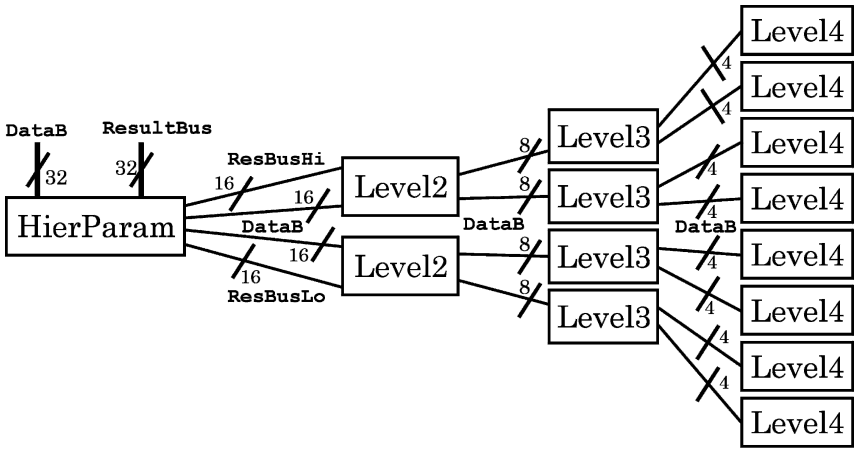


Fig. 15.5 Block diagram of the `HierParam` instance hierarchy

All the module declarations have been collected into one file, `HierParam.v` in the `Lab18` directory, to make the problem more easily visible.

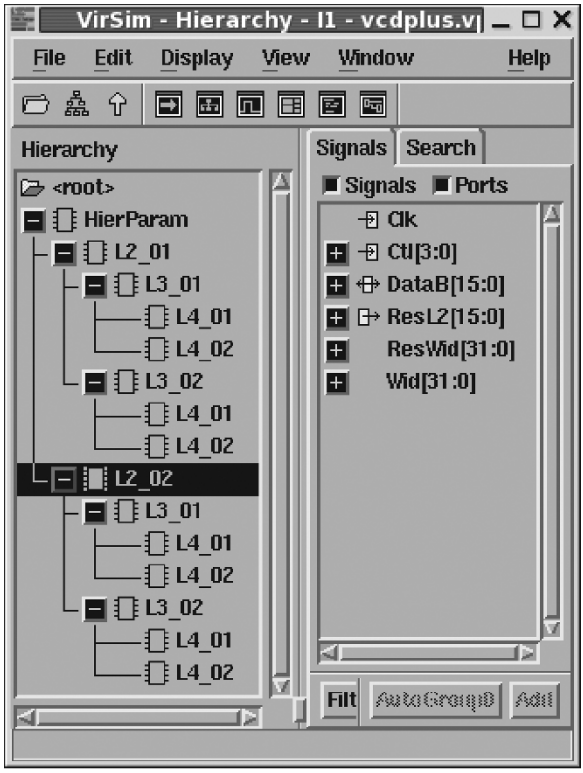


Fig. 15.6 VCS display of the `HierParam` hierarchy

Modify `HierParam.v`, overriding the parameter default assignments, so that the value of the width of the `DataB` bus is changed consistently in every module by assigning just one parameter value at the top level. Don't change the default assignments declared in the file. Check your answer by compiling in a simulator and looking at the hierarchy (compare Fig. 15.6).

15.2.1 Connection Lab Postmortem

What if a designer wants to assign delays and pass parameters to the same instance?

15.3 Hierarchical Names and Design Partitions

15.3.1 Hierarchical Name References

We have studied hierarchy in several contexts, mainly module instance hierarchy and hierarchical name references in generated instances.

The hierarchy in verilog begins where the compiler (simulator or synthesizer) sees it, whether we are preparing for simulation or synthesis. If a submodule is being designed and tested, the names in it will be rooted at the top submodule; when this submodule is linked in to the larger design, and that design is loaded into the simulator or synthesizer, its name references will be rooted elsewhere.

For this reason, hierarchical references (`A.B.C` ... etc.) should be restricted to the current module (as in a `generate` statement – recall *Week 6 Class 2*), or to the submodules of the module in which they appear. A hierarchical reference from the current module downward, toward the leaf cells of the design, can be controlled by the designer. This direction of reference is reasonably safe, because the current module depends on its instances for its functionality; changing the leaf cell (instantiation) structure means a redesign; in this redesign, the hierarchical names may be modified as necessary.

On the other hand, upward references easily are broken beyond repair. Any module has some functionality as such; and, so, it may be reused not only in its specified location (if any is specified) but also elsewhere. If reused this way, it is likely that the new instantiating module will differ from the previous one, probably invalidating hierarchical upward name references. The extreme case would be a verilog model of a library component; it should be designed for use anywhere; so, any hierarchical upward reference in it would render it nonfunctional.

15.3.2 Scope of Declarations

Here's a review of the scope of identifiers of all sorts. We introduced some of these concepts in *Week 6 Class 1 & 2*.

By ***scope***, or name scope, is meant that region, in the verilog, of visibility. Visibility implies a name conflict when the visible object's identifier is redeclared or redefined. Objects of wider scope conceal the names of those of narrower scope. For example, two different modules (wide scope) each may contain a different net named `Clock`: The names of nets have narrower scope than names of modules, so the net names declared in one module are not visible in another module.

Inside a procedural block (including inside a `task` or `function` declaration), a name must be declared before its first use ("before" means above the use in the file). However, in a `module`, a *named block* declaration will be found by the compiler if it is anywhere in the `module`.

Here is a listing of name scopes which are reasonably distinguishable:

- `` `define `` macro identifiers: No real scope limitation or hierarchy; defined *everywhere* after the compiler encounters them, in compilation order (which may be unrelated to design structure).
- `module` names: Global scope, shared with UDP's (`primitives`) only. One level of name scope; no hierarchy.
- `module` instance names. These may be extended to any number of levels of hierarchy. The top instance must be in a `module`; but, with this exception, nothing but instances may exist in a module instance hierarchy.
- concurrent block names: These include any named `initial` or `always` blocks within a module. The `specify` blocks to be covered later also fall here. Concurrent blocks are allowed only one level of name scope, which means no hierarchy. However, `generate` blocks are in this scope and may contain a `generate`d hierarchy after unrolling; such a hierarchy may include named `always` and `initial` blocks.
- procedural block names: These include `task`s and `function`s, as well as any named procedural `begin-end` block, such as one in a procedural `for` loop. These objects may be mixed with one another and extended to any number of levels of hierarchy.
- element names: Variables (`reg`, `wire`, etc.) and constants (`parameter`s, `localparam`s, `specparam`s, etc.). These exist within a scope, but they define no scope of themselves.

Other language constructs such as expressions or literals (including delay values) have no name; and, so, scope is irrelevant to them

We shall look briefly at verilog `config` blocks later. A `config` is assigned an identifier (name) and specifies a collection of design objects, but it has no design functionality. A `config` may be located anywhere a module may be located. A `config` is more like a makefile, or a system disc file or directory, than a named block with a meaningful relationship to a hierarchy. The names in a `config` generally refer to objects in libraries external to the design.

15.3.3 Design Partitioning

Partitioning of a design may be done to reduce the complexity of the task of any one design engineer, to allow concurrency in the design work to speed the design to completion, or to achieve some design-tool related goal.

The most important single consideration in partitioning is that of the interfaces between the parts. Each different part has to have stand-alone, specific functionality differentiating it from all the others. When a part is to be used in several different places in a design, all uses have to be taken into consideration when writing design specifications for that part.

The interfaces have to be logical and intuitive, so that different designers, or the same designers at different times, recognize and easily understand the objectives

of the partitioning. The interfaces also have to be well thought-out so that the nets crossing partition boundaries have completely specified types, widths, timing and sequential protocol. Any change in an interface must be considered in a system context and should be made with great reluctance, because it may imply redesign of several or all parts involved.

An elaboration of the verilog language, *System Verilog*, has a construct called an *interface* as one of its features. This construct essentially predefines module headers and adds design I/O functionality; it may be applied unmodified to several modules, thus guaranteeing I/O consistency among them.

In verilog (or System Verilog), partitioning always should be done on module boundaries. A module is the smallest verilog design unit which can be compiled separately. If a partition has to be redrawn, new modules separating the different parts should be declared, so that the parts might be compiled and tested separately. This allows the designer working on one partition to write a behavioral model or testbench representing the other(s), permitting effective debugging independent of other parts of the design.

Some rules of thumb:

Clock domains. Partitioning should be done to separate clock domains. Parts with different clock rates or sleep-mode states should be designed separately from one another. Synchronizers (see below) should be used wherever a foreign clock enters a new domain; this makes module inputs the logical place to insert such devices.

Voltage islands. Parts operating at different supply voltages should be separated; the gates generally have to come from different libraries and often different vending organizations. It is true that some CMOS libraries can be used in a range of voltages, for example 3 V to 5 V, but the power consumption or speed will be optimal only in one narrow voltage range. Separation also allows intelligent and efficient selection of voltage level-shifters.

Level-shifters usually are located in their own partition, or in the voltage partition of the shifted levels.

Output latches. Each part of any substantial complexity should have clocked, latched outputs, making its internal timing separate from, and independent of, that of any other part.

Control timing. When different parts pass control signals to one another such that the sequence of these controls is significant (for example, wait states on a read from a memory), the sequence may have to be enforced by latches on inputs or outputs, or by clocking on different cycles or opposite edges.

IP reuse. Large commercial IP ("Intellectual Property") blocks may be purchased separately and make up a large fraction of any modern VLSI IC. Microcontroller cores and memories are typical examples. Each of these blocks should be allocated its own partition in the design.

Test considerations. The partitions usually should allow for internal scan insertion and possibly boundary scan; in any case, parts should allow observability of crucial intermediate results. Latches on outputs make ideal components to be replaced by scan components with little or no performance decrement; this means that the same simulation test vectors may be used before and after scan insertion.

Synthesis considerations. In general, logic optimization will work best on blocks of random logic in a certain size range, somewhere between about 300 to 50,000 transistors. This roughly would be about the same as 30 to 5000 instances, depending on the library, or 75 to 15,000 gate-equivalents. A block too large may take too long to optimize, lengthening the debug cycle; a block too small will not give the optimizer enough to work on, so that the designer's original input will not be improved. Partitioning should be planned to combine or split such logic, with the synthesizer's capabilities an important factor in the decision.

It's probably best for the designer to assume reliance on automation in the first approximation – the initial cycle of write, debug, and synthesize. After seeing the first working area and timing result, automated tuning followed by manual constraint tweaking would be a typical progression. After fully constrained synthesis, manual edits occasionally may be necessary. Manual editing of a synthesized netlist can produce results better than anything the tool can accomplish; but, on the other hand, optimization by the tool may be obstructed by too many "don't touch", manually-inserted structures.

A synthesizer usually allows for optional shifting of gates across sequential logic boundaries, thus combining random logic across the boundary and improving opportunities for optimization. In a flattened netlist, this kind of retiming may improve certain module instances more than others, providing refinements not available in a partitioning context. Except within a single module, this feature should be used only in the final design stages, because breach of the design partitioning is irreversible, and localization of defects or manual corrections may become very difficult after logic has been shifted in or out of what originally were separate verilog partitions.

However, keep in mind that a back-end tool such as a floorplanner can be made to reconstruct in the physical layout the original verilog source hierarchy, even from a flattened netlist. This generally is possible because the naming convention imposed during uniquification and flattening by the synthesizer or optimizer is consistent with the original verilog module names.

15.3.4 Synchronization Across Clock Domains

A clock domain is defined by a clock which runs independently (by a different oscillator) of other clocks. A derived or generated clock created by PLL or frequency divider may be said to be in a different domain from the original clock, but we do not use this definition in the present lecture. The reason is that derived or generated clocks are phase-locked to their original clock, and problems of synchronization are limited merely to considerations of skew and jitter.

When data have to be transferred from one clock domain to another, problems arise which do not occur in a single-clock synchronous design. Specifically, when two clocks run at different rates, and are not derived from one another, any clock state can be simultaneous with any other at a given component input. This means that data (or control) from one clock domain can be sampled in an intermediate logic

state, far enough away from a '1' or '0' that the gate's response (slew rate) may be very slow. Actually, intermediate-state sampling in general would be expected to occur frequently, and at random, given typical GHz clock rates in modern designs.

A simple mechanical analogy may help conceptualization of this problem: If a fencing foil is dropped 1 m at a random, almost vertical angle onto a hardened steel floor, there is almost no chance it will happen to balance upright on its tip for one full second before falling over. However, if such a drop was repeated a billion times, by simple chance several of the foils will have balanced upright for one second. Repeated enough of times, a foil or two will be found to have balanced for 10 seconds or even for a minute.

On a fine enough time scale, an input always can be sampled so that it leaves the output undetermined, no matter how long the wait. See Fig. 15.7 for an illustration of oscilloscope-like, progressively increasing, time resolution.

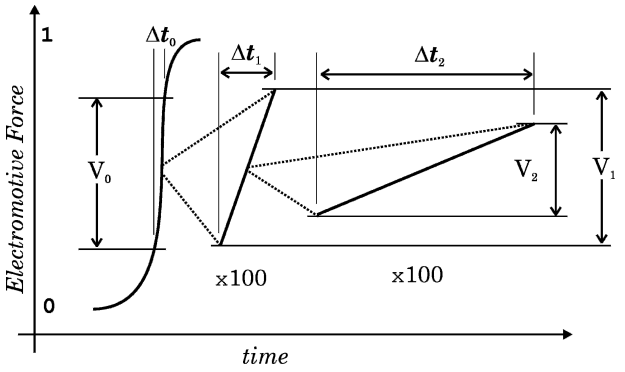


Fig. 15.7 Each V spans a range such that the output will go to '1' at the top and '0' at the bottom. Figure not to scale. If $\Delta t_0 = 100\Delta t_1$ and $\Delta t_1 = 100\Delta t_2$, then $V_1 \cong V_0/100$; $V_2 \cong V_1/100$, which implies approximately that the probability p that the gate output will be intermediate during the respective Δt will be $p_1 = 100p_0$ and $p_2 = 100p_1$

So, from time to time, probably many times per second, a gate in one domain will sample an input voltage so exactly centered in its input range, that the gate will be almost perfectly balanced and can not switch to propagate either a '1' or a '0' before the next clock cycle in its own domain. As time passes after the balanced sampling, the gate eventually will switch one way or the other, but this may be too late for the correct logic level to be propagated.

This problem is overcome by latching inputs in the receiving clock domain. As shown in Fig. 15.8, an output flip-flop or latch is triggered by the sending clock, the data are latched, and the latched logic level then is sampled in a synchronizing flip-flop on a receiving-clock edge. However, as explained above, there is a remote chance that the synchronizing flip-flop will not have settled before its value is propagated, perhaps ambiguously, into the receiving logic. This chance is eliminated by using two synchronizing flip-flops in place of one, creating a little two-stage shift register.

The input balance point of the second flip-flop almost certainly will not be at the same voltage as the balanced output of the first. Then, if balanced, the old data in the first flip-flop has almost an entire clock cycle to settle before being sampled by the second one, which is in a well-defined state anyway. The probability that the second flip-flop will be balanced on the balanced output of the first one is remote enough to be ignored or to be compensated, if necessary, by occasional ECC operations.

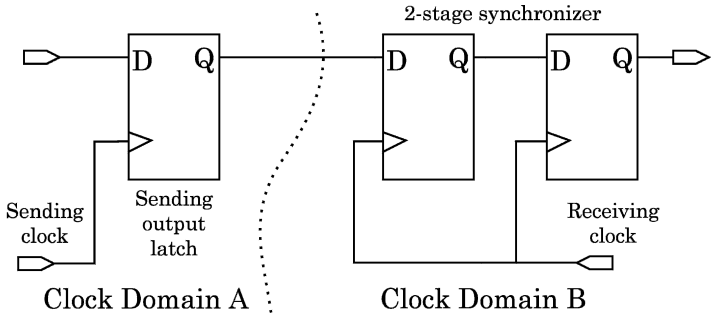


Fig. 15.8 Synchronization across two clock domains. Clocks A and B run at different and independent frequencies. Domain B must be prevented from clocking indeterminate data values from A

15.4 Hierarchy Lab 19

Do this work in the `Lab19` directory.

Lab Procedure

Step 1. Create a file named `MiscModules.v` and type in the following empty modules (headers only):

```
module Wide(output[95:0] OutWide, input[71:0] InWide);
endmodule
module Narrow(output[1:0] OutNarrow, input[1:0] InNarrow);
endmodule
module Bit(output Out, input In);
endmodule
```

Step 2. Connecting a 1-bit signal to a 2-bit port. Using the file in Step 1, instantiate `Bit` in `Narrow`, and connect it to the `Narrow` I/O's:

```
module Narrow(output[1:0] OutNarrow, input[1:0] InNarrow);
Bit Bit1( .Out(OutNarrow), .In(InNarrow) );
endmodule
```

Do not use a part-select. What do you think will happen? Only one bit in the `Narrow` busses can be connected: Which one?

Test your assumptions by implementing something trivial in `Bit`, for example,

```
assign Out = 1'b1;
```

Simulate `MiscModules` to see which bit is assigned.

Step 3. A hierarchy paradox. Without changing anything else in Step 2, instantiate `Narrow` in `Bit`; leave `Narrow` with no port map (I/O connection). Now you have two modules, each instantiated in the other. Do you think this is legal? Check it by compiling `MiscModules` for simulation.

Step 4. Connecting a 2-bit signal to a 1-bit port. Comment out the `Bit` instantiation in `Narrow` from Step 2, leave `Narrow` instantiated in `Bit`, and connect the `Narrow` ports analogously to the opposite connections in Step 2 (with no part-select or bit-select). Again, try to predict which bit will drive the output. Add a simple statement to `Narrow`, and simulate to test this.

Step 5. Repeat Step 4, using `Narrow` and `Wide` instead of `Bit` and `Narrow`. You should be able to predict the results. Optional: Simulate to see this result.

Step 6. Clock synchronization simulation. Set up a testbench module named `ClockDomains` which has two clock generators, one with half-period of 2.011 ns (the slow clock), and the other with half-period of 1.301 ns (the fast clock). To resolve these periods in the simulator, set `` `timescale `` in the testbench to `1ns/1ps`.

As shown in the schematic of Fig. 15.9, write a model of a 3-bit RTL up-counter in a module, `Counter3`. Be sure to include a reset, and assign all three bits to the `Counter3` outputs. For this exercise, make the counter count clock edges, not cycles, using `always@(Clk)` rather than `@(posedge Clk)`. Instantiate this model twice in the testbench module. One instance will represent logic in the fast clock-domain, and the other, logic in the slower clock domain.

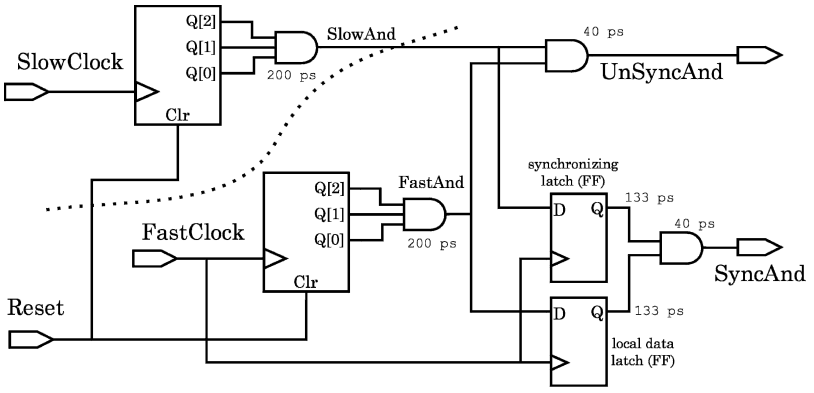


Fig. 15.9 Schematic representation of `ClockDomains`. Two uncorrelated clock domains are combined raw in `UnSyncAnd`. A synchronizing latch on `SyncAnd` makes slow-domain data more predictable when sampled in the faster domain. Delays are approximate

Clock one counter instance with the fast clock, and the other with the slow clock. Use a continuous assignment and-reduction operator to assign the and of the three counter bits of each counter to its own net variable. Attach a small delay to these two continuous assignment statements, maybe 200 ps or so. These two nets are to be compared in the fast clock domain; when both are '1', the fast domain will have to do something (we leave the operation undefined in this example).

A. To see the raw, unsynchronized coincidence of these ands, just *and* them both in another continuous assignment onto a net called UnSyncAnd. Attach another very small delay to this and statement, perhaps 1/5 of the other delay. Use decimal fractions (1/5.0 rather than 1/5) to force evaluation as reals and not integers. Simulate your model to verify it. You should see the UnSyncAnd occasionally going to '1' or glitching high. The different positive pulses will vary considerably in width because of the incompatibility of the clock frequencies (both half-period values are prime numbers).

B. To add a synchronizing latch (=flipflop) in the fast domain, use a simple RTL statement to sample the 3-input and from the slow clock domain when the local (fast) clock is high. Give this component a slightly shorter delay than that of the ands.

We don't require a synchronizer for the fast clock and, but we do require a latch to hold its value; so, also write an RTL latch just as above for the fast-clock and value:

```
localparam AndDelay = 0.200; // 200 ps 3-input and gate delay.
localparam LatchLagDelay = AndDelay/1.5;
...
reg HoldSlowAnd; // The synchronizing latch storage.
always@(posedge FastClock)
  if (FastClock==1'b1)
    #LatchLagDelay HoldSlowAnd = SlowAnd; // Sample the slow-domain and.
...
```

Then, complete the exercise by *and*ing both latched 3-input statements, one from each clock domain:

```
assign #(AndDelay/5.0) SyncAnd = HoldFastAnd & HoldSlowAnd;
```

Compare SyncAnd with UnSyncAnd in a simulation (see Figs. 15.10 and 15.11). You will find that the SyncAnd data are much cleaner and are glitch-free. Latching the fast side of the UnSyncAnd input makes little difference in the glitches and irregular pulses: Garbage in; garbage out.

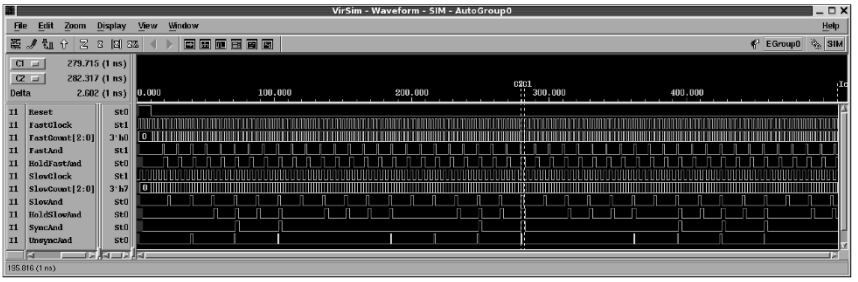


Fig. 15.10 The `ClockDomains` simulation, comparing the two *and*s

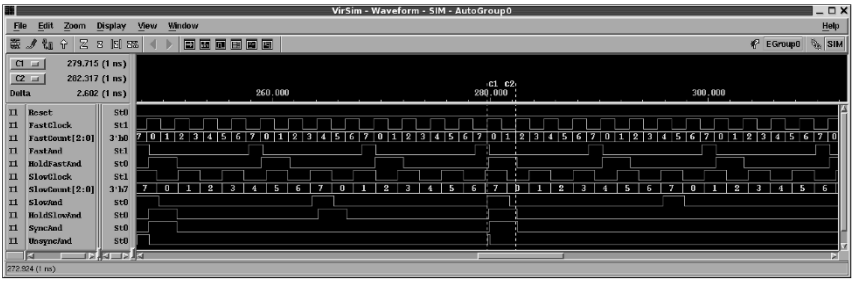


Fig. 15.11 Zoom in on `ClockDomains` narrow pulse

Given that the clock frequencies are fixed, tuning the delays can introduce or eliminate potential glitches in the synchronized data. Also, some glitch-like pulses in the `UnSyncAnd` data can be admitted in the synchronized data by shifting or tuning the sampling windows. These are arbitrary issues when dealing with disjoint clocks, because synchrony is essentially arbitrary when the clocks are independent. It is the interface that defines synchrony, not the clock domains taken either separately or jointly.

15.4.1 Lab Postmortem

We have recommended keeping delays out of procedural blocks for synthesis reasons and, instead, putting the resultant delays in continuous assignments to module `output` or `inout` ports. How does this affect the partitioning practice of latching all data outputs?

15.4.2 Additional Study

Review parameters, hierarchical names, and connection rules in Thomas and Moorby (2002) sections 3.6 and 5.1–5.2.

Palnitkar uses traditional module header declarations extensively; Thomas and Moorby (2002) describes the differences briefly in Preface pages xvii–xix.

Chapter 16
Week 8 Class 2

16.1 Verilog Configurations

The majority of designs begin with *C* models or behavioral (bus-transfer) models and, after verification of the partitioning and interfaces, proceed to further detail at RTL or gate level. This means that a module's implementation may change radically during the design process. Obviously, somewhere, there has to be a way of obtaining a complete list of the files comprising a design, or the design could not be used for anything. To substitute different versions of a module, designers have relied on makefiles or shell scripts. As an alternative to depending on the filing system or other nonverilog functionality, here we introduce a way of managing design versions and formats within the verilog language.

16.1.1 Libraries

The definition of a "library" of modules or technology-specific gates is vendor specific: Synopsys has its own definition, and other EDA vendors have theirs. Although file formats and byte-ordering make any compiled object nonportable, still, there has been some demand for a way within the verilog language for organizing a design. Verilog provides for a ***library mapping file*** as a way of mapping a library to the filing system. An example of this, for Synopsys synthesis, was given at the start of the generic DC compilation script provided at the beginning of this course. In the simplest case, the name of the library simply is paired with a directory containing the library contents; all subsequent references to the contents then are by the library name.

16.1.2 Verilog Configuration

The verilog ***configuration***, new in *verilog-2001* (IEEE Std 1364, section 13), is meant to make module versions easily interchangeable, and to make libraries more portable.

J. Williams, *Digital VLSI Design with Verilog*,

A configuration is bounded by the keywords `config` and `endconfig`. It describes a library comprised of everything in a design useful for a regression test or other design activity. It is stored in a design file at the same level as a `module`.

The format of a `config` is as follows:

`config` *config name*`;`
`design` *design_top_name*`;`
`default` *list_of_libraries*`;`
`cell` *cell_name* **`use`** *library_name*`;`
`instance` *inst_name* [**`use`**] *instance_liblist* [`.cell`]`;`
`endconfig`

Keywords are in bold above; the other words indicate design-specific identifiers or lists. Only the `design` and the `default` are required. Notice that the keyword **`default`** (`case` statement) is reused here in a very different context. The referents introduced by the keywords in a `config` are as follows:

- **`config`**. This introduces an identifier for this `config` statement. The intention here is that interchanging `config`s allows the design to be compiled or otherwise used with different collections of library elements or module versions, for different purposes. For example, a `config` early in a CPU design might be named `CPU_BusXfer`; later, a new `config`, with different references to the design library, might be named, `CPU_RTL`; then, later, `CPU_FloorPlanned`, and so forth. Any of the `config`s might be used to obtain information on the design at any `config`-identified stage.

  As another example, different `config`s could be used for simulation than for synthesis.
- **`design`**. This specifies the top-level module in the design assumed archived in the library involved. Verilog hierarchical names are used when a library is involved. For example, if the library name was "`IntroLib`", then the top-level module might be identified by,

```
design IntroLib.Intro_Top;
```

- **`default`**. This specifies the library, or libraries, in search order, from which the instances in the design shall be taken, unless otherwise specified in the **`instance`** statement. For example, if we used the Synopsys synthesis class library's `class.db` file for everything, we could complete this `config` for that exercise simply by stating,

```
default class;
```

- **`cell`**. This specifies the library from which the named cell (module) shall be taken. The **`use`** clause is used to name the library cell itself.
- **`instance.`** If more than one library was involved, and a given module name was present in more than one, then the module or gate instances not to be found in the default library list are specified individually here. A **`use`** clause optionally

may be used to pick a specific cell. The name starts with the name as given in the design statement. For example, recall that we used an *xor* expression (^) in the Lab 1 exercise. Suppose we had a special synthesis model of an `xor` gate, in library file `special.db`, which we wanted to be configured for our present purposes. Then, the `xor` in a previously synthesized gate-level netlist of our introductory lab exercise might be specified this way:

```
instance IntroLib.Intro_Top.XorNor.xor_01 special;
```

Using the synthesis output library, the **`instance`** statement could be just,

```
instance Intro_Top.XorNor.xor_01 special;
```

Library and design configuration maintenance is a very tool-dependent and specialized topic, and we shall not dwell more on it here.

The verilog *configuration* adds almost no functionality to the language, because modern designs always are configured already in the filing system. The language-based *configuration* thus may increase design failure by establishing conflicting multiple points of configuration control. No tool known to the author has implemented the verilog `config`, and so there is no lab exercise on the topic of verilog *configuration*s.

16.2 Timing Arcs and *specify* Delays

We move on to study timing arcs inside a module.

16.2.1 Arcs and Paths

Delays across a module in verilog are said to be distributed in space along timing arcs; the *arc* refers to a phasor angle swept out during a clock period. The rationale is that eventually the module will correspond to a geometric object on a chip, and the timing between distinct, identifiable places on that chip will be important.

A timing arc is equivalent to a delay between two points in a module. These points are viewed as structural, so they can not be represented solely by expressions or assignment statements; they have to be locations. Timing arcs represent scheduled verilog events and also a temporal distance between them. The distance may span just a single net, or it may span a network of multiple gates of combinational or even sequential logic. A timing arc may exist between a clock and a data pin. In a structural model, timing arcs may be defined between ports or internal pins of any module. At any level, each of these timing arcs may be assigned a ***path delay***, the delay of the timing arc mapped to a path.

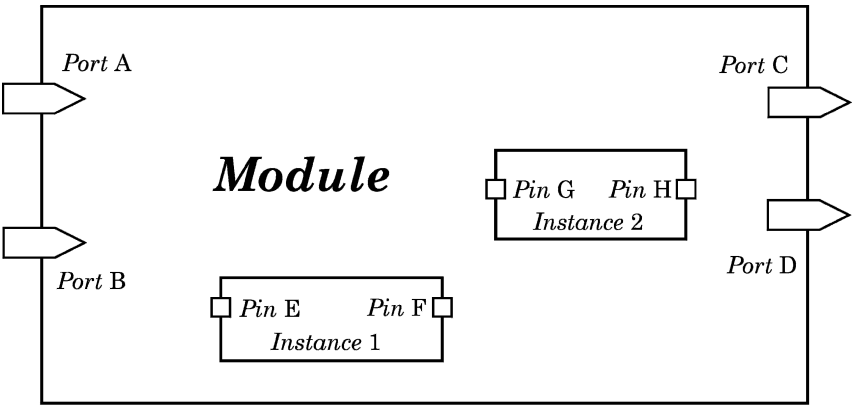


Fig. 16.1 Timing arcs are defined only between ports or pins. Assume A, B, E, and G are verilog inputs and the others are outputs

In Fig. 16.1, a timing arc may be defined between module ports A or B and C or D. However, in general, if there is an arc between A and C, there is no single arc between A and both C and D. Timing arcs also may be defined between A or B and E or G, between F or H and C or D, and between F and G. No arc is visible between E and F or G and H in the module shown; however, such arcs may be defined in the modules which were instantiated as Instance 1 or Instance 2. The delay on the paths between E and F or G and H may be calculated to determine arcs between, say, B and D and passing through one or both of the instances shown.

Although not shown in Fig. 16.1, any path must be represented by a causal connection, usually a simple wire or cloud of combinational logic – but possibly by layers of sequential logic, too.

The delay specifications we shall discuss here can be used by simulators or static timing analyzers.

16.2.2 Distributed and Lumped Delays

In our labs, we so far have assigned both distributed and lumped delays. A ***distributed delay*** is one which separately sums two or more delay values along a timing arc. Or, alternatively, a timing arc passing through two or more pins, each assigned a delay on a subarc, represents a distributed delay. A ***lumped delay*** is one which is summed by the designer and assigned solely to the final point on a timing arc, the intermediate subarcs not being assigned any delay (also not being assigned `#0`).

A simple comparison of distributed vs. lumped delays is as follows:

```
module DistributedDelay (output Z, input A, B);
wire Node;
assign #1 Z = Node;       // Output port delay.
and #(2,3) (Node, A, B); // Pin delay.
endmodule
//
module LumpedDelay (output Z, input A, B);
wire Node;
assign #(3,4) Z = Node;  // Total delay lumped on output port.
and (Node, A, B);
endmodule
```

The distinction between distributed and lumped delays is made only in the context of modules with substructure; the distinction is not very relevant to simple behavioral or RTL models. As a more elaborate example, consider an RTL model of a register composed of four three-state flip-flops and having no substructure:

```
`timescale 1ns/100ps
module FourFlopsRTL #(parameter DClk = 2, DBuf = 1)
                     (output[3:0] Q, input[3:0] D, input Ena, Clk);
reg[3:0] QReg;
wire[3:0] Qwire;     // Not used yet.
//
always@(posedge Clk)
  #DClk QReg <= D;
//
assign #DBuf Q = (Ena==1'b1)? QReg: 'bz;
//
endmodule
```

There is a delay from `Clk` to `QReg`, and another from `QReg` to `Q`, but the only timing arcs in this model are from `D`, `Ena`, or `Clk` to `Q`.

The (`parameter`) default delay on the `D->Q` path is 2+1 = 3 ns; the default delay on the `Ena->Q` path is 1 ns; and, the default delay on the `Clk->Q` path is 2+1 = 3 ns. Although the values can be summed, none of these defaults represent either distributed or lumped delay in any meaningful sense.

However, consider the more structural design below. This design, `FourFlopsStruct`, is functionally and timing equivalent to `FourFlopsRTL`; it contains some substructure implemented as arrayed instances:

```
module FourFlopsStruct #(parameter DClk = 2, DBuf = 1)
                        (output[3:0] Q, input[3:0] D, input Ena, Clk);
wire[3:0] QWire;
//
DFF #(.DClk(DClk)) DReg[0:3](.Q(QWire), .D(D), .Clk(Clk));
assign #DBuf Q = (Ena==1'b1)? QWire: 'bz;
endmodule // FourFlopsStruct.
// ------------------------------------------------------
module DFF #(parameter DClk = 2) (output Q, input D, Clk);
reg  QReg;
always@(posedge Clk) QReg <= D;
assign #DClk Q = QReg;
endmodule // DFF.
```

The `FourFlopsStruct` model has the same timing as `FourFlopsRTL`, but the delays as calculated for `FourFlopsStruct` are distributed delays. The `DReg` instance output delay value `DClk` could be considered a lumped delay, viewed by itself.

It is possible to distribute delays on nets, too, by assigning a delay when each net is declared. This is analogous to the way we have seen a strength assigned to a net. For example, "`wire #3 DataOut;`" schedules delayed changes in `DataOut` the same as though `DataOut` was being assigned from a temporary variable by a continuous assignment statement with the given delay.

The author never has seen net delays used in a design, although, in layouts in today's deep submicron pitches, the net capacitances have come to account for a significant fraction of the delay in a design, the remainder coming from the internal gate delays. Gates have locations, so practice has been to assign timing in the verilog only to arcs between gates. The net delays in practice are accounted for when simulating with back-annotation from a floorplanned or placed-and-routed netlist.

Using what we know from previous models, `FourFlopsStruct` could be rewritten with lumped delays as follows:

```
module FourFlopsStructL #(parameter DClk = 2, DBuf = 1)
                         (output[3:0] Q, input[3:0] D, input Ena, Clk);
wire[3:0] QWire;
localparam DTot = DBuf + DClk;
//
DFF DReg[3:0] (.Q(QWire), .D(D), .Clk(Clk));
assign #DTot Q = (Ena==1'b1)? QWire: 'bz;
endmodule // FourFlopsStructL.
// ------------------------------------------------------
module DFF(output Q, input D, Clk);
reg  QReg;
always@(posedge Clk)
  QReg <= D;
assign Q = QReg;
endmodule // DFF.
```

Of course, as we might recall, the lumped delay `DTot` could be written to express rise and fall transitions separately, this way: (`DTotR`, `DTotF`). Furthermore,

technology considerations might be taken into account to provide the verilog simulator with minimum, typical, and maximum delay estimates; from previous study, we know this might be written, (DTotRmin:DTotRtyp:DTotRmax, DTotFmin:DTotFtyp:DTotFmax). But, this is as far as we can go with what we learned up to now.

16.2.3 specify Blocks

A specify block begins with the keyword **specify** and ends with the keyword **endspecify**. A specify block is at the same concurrent level in a module as an always or initial block. A specify block has no functionality, but it may be used to determine the details of timing required for (*a*) an accurate gate-level verilog library model or (*b*) any other module with precisely known, and precisely required, timing.

Timing in a specify block can be more precisely and flexibly controlled than timing added to declarations, statements, or instantiations. A specify block makes it possible to model timing without knowing or defining any functionality; functionality can be added later with no further timing editing at all. A specify block may be used for a static timing model of an encrypted IP block without revealing any functionality.

We shall study timing checks later in the course; but, for now, it's important to know that a specify block is the only place in a verilog model where a timing check may be stated. Possibly to prevent confusion over the '$' which begins the name of every timing check, system tasks, which also start with '$' ($display, $monitor, $dumpvars, etc.), are forbidden in a specify block.

A specify block may contain specparam definitions, timing checks, certain pulse filtering conditions, and module path delay specifications. We shall concentrate on the specparams and module path delays for now.

Names of ports declared in a module are visible within a specify block in that module; nothing in a specify block can reference declared reg variables or the port pins of components instantiated in a module. However, a net, parameter value, or localparam value declared in the same module may be used in a specify block.

In summary, specify blocks are at the same level in a module as always, initial, or generate blocks and are delimited by the keywords **specify** ... **endspecify**.

A specify block may contain:

- parameter or localparam references
- module port or net references
- specparam definitions
- module delay specifications
- timing checks.

A `specify` block may not contain:

- any other block
- any declaration other than of a `specparam`
- any assignment to a `reg` or net
- any instance or other design structure
- any `task` or `function` call, including system tasks or functions.

16.2.4 specparams

A `specparam` is a verilog parameter which usually is used only in a `specify` block. Although many tools will not allow it, a `spacparam` also may be declared outside a `specify` block.

This special kind of parameter exists for convenience of parsers and other tools which extract timing information from a model. There is only one unique feature to a `specparam`: It may be assigned multiple numerical values, for example in a declaration of the form,

`specparam` *Name* $= (x, y, z)$;

in which x, y, and z are numbers or timing triplets representing delays.

When creating a timing specification for a module, good practice is not to assign verilog literals to the delays; instead, the literals or other constants should be assigned to `specparam`s, which then may be referenced in timing expressions later in the `specify` block. The `specparam`s may be given mnemonic names and, of course, used in any number of timing expressions within that `specify` block.

For example, we introduce our first path delay:

```
module ALU (output[31:0] Result, input[31:0] ArgA, ArgB, input Clk);
...
specify
specparam tRise = 5, tFall = 4;
...
(Clk *> Result) = (tRise, tFall); // A simple full-path delay.
endspecify
...
endmodule
```

A `specparam` also may be assigned a timing triplet for simulator min-typ-max alternatives. For example,

```
specify
specparam tRise = 2:3:4, tFall = 1:3:5;
   ...
   (other stuff; maybe complicated)
   ...
  (Clk *> Q,Qn) = (tRise, tFall);
endspecify
```

16.2.5 Parallel vs. Full Path Delays

There are two main kinds of path delay statement, full-path ("*>") and parallel-path ("=>"). A full-path delay applies to all possible arcs between all bits of the ports named. A full-path delay may imply a timing-arc fanout (as in the `Clk *> Result` example above) or a fanin. Bit-select or part-select delays usually would be specified by full path.

A parallel-path arc only can exist between endpoints with equal numbers of bits; no fanin or fanout of the arc delay is allowed. This kind of path is most usual between scalar (single-bit) ports. When the ports are vectors, the bit delays are mapped one-to-one, in parallel and in declared order.

Path delay specifications of any kind are illegal when fanned-in logic such as a `wor` or `wand` net directly drives a port; such fanin's must be replaced by logically equivalent gates with single outputs if path delays are to be assigned.

Examples of path delay specifications:

```
module FullPath (output[2:0] QBus, output Z, input A, B, C, Clock);
... ( functionality omitted) ...
specify
  specparam tAll=10, tR=20, tF=21;
  (A,B,C *> QBus) = tAll;
  (Clock *> QBus) = (tR, tF);
endspecify
endmodule
// ------------------------------------------------
module ParallelPath (output Z, input A, B, C, Clock);
... ( functionality omitted) ...
specify
  specparam tAll=10, tR=20, tF=21;
  (Clock => Z) = tAll;
  (A => Z)     = (tR, tF);
  (B => Z)     = tAll;
endspecify
endmodule
```

An interesting feature of `specify` delay values is that there may be as many as six delay values for a path capable of turnoff. As stated in Thomas and Moorby (2002) section 6.6, a `specify` block assignment may assign six different delays to a path in the order, (*0_1*, *1_0*, *0_z*, *z_1*, *1_z*, *z_0*). This level of precision rarely is useful in modern design, because synthesizer or floorplanner back-annotation generally will be more accurate and less effortful than this level of designer guesswork in the source verilog.

16.2.6 Conditional and Edge-Dependent Delays

Any delay may be assigned to a path conditional on the nonfalsity of an expression. The usual verilog operators may be used in this expression, and the evaluation is nonfalse if a '1', 'x', or 'z'. If the expression is on a vector object, the (non)falsity depends only on the LSB. However, only simple statements are allowed; no *else*, *case*, or other constructs with alternatives are allowed. The keywords `posedge` and `negedge` have their usual meaning. For example,

```
// output[3:0] Z, input[3:0] A, input Clk, Clear are declared ports.
...
specify
specparam ClkR=2, ClkF=3, ClearRF=1, AThruR=4, AThruF=5;
//
if (Clk && !Clear)(A => Z) =          (AThruR, AThruF);
if (A[0] && A[3]) (A[1], A[2] *> Z) = AThruR; // Lists are OK
if (A[1] && A[2]) (A[0], A[3] *> Z) = AThruF;
if (!Clear)       (negedge Clk *> Z) = ClearRF;
if (!Clear)       (posedge Clk)*> Z) = (ClkR, ClkF);
                  (posedge Clear *> Z) = ClearRF;
endspecify
```

A path destination may be assigned a polarity, so that the delay is associated with a certain edge direction at the related port. This allows the data path to be used to determine the delay, as well as the output change. For example,

```
// output Q, Qn, input D, Clk: Q1 and Q2 are logically equivalent.
...
specify
specparam tR_Q = 5, tF_Q = 6.5;
...
( posedge Clk => (Q1 +:D) ) = (tR_Q, tF_Q );
( posedge Clk => (Q2 -:D) ) = (tR_Q, tF_Q);
endspecify
```

In the first statement above, `Clk` clocks in `D` to `Q`; if `Q1` was '0' and `D` is '1', then `tR_Q` is the delay; if `Q1` was '1' and `D` is '0', then `tF_Q` is the delay.

In the second statement, the '$-$' means that the `D` polarity is inverted, so the effect of `D`, when compared with the explanation of the first statement, is to choose the other delay. So, if `Q2` was '0' and `D` is '1', then `tF_Q` is the delay; if `Q2` was '1' and `D` is '0', then `tR_Q` is the delay.

Similarly, both parallel and full path delays may be assigned with polarity dependence. The '$+$' means noninversion; the '$-$' means inversion. So, "$-=>$" refers to a change on the left which is in the opposite direction from the resultant, parallel-path delayed, change on the right. The same principle applies to full path statements, using "$-^{*}>$" and "$+^{*}>$". For example,

```
...
specify
(clk -*> Q1, Q2) = t_ClkTog; // Delay when Q1 | Q2 goes opposite clk.
(clk +*> Q1, Q2) = t_ClkReg; // Delay when Q1 | Q2 goes same way as clk.
endspecify
```

These special features might be useful in switch-level modelling.

Delays may be given explicitly for transitions to and from 'x' or 'z' as well as the other logic levels. This leads to timing specifications not just of the two value (*rise*, *fall*) or three value (*rise*, *fall*, *turnoff*) variety, which we have seen, but also of six values (all transitions with 'z') or twelve values (all transitions with 'x' or 'z'). See Thomas and Moorby (2002) appendix G.8 for a list of these specifications.

16.2.7 Conflicts of `specify` *with Other Delays*

As mentioned in Thomas and Moorby (2002) section 6.6 and specified in IEEE Std 1364, section 14.4, when a `specify` block contains a delay assigned to a path also delayed outside the `specify` block (in a delayed assignment or delayed primitive instance statement), ***the greater of the two delays*** will be the one simulated.

This said, keep in mind that individual simulators are likely to be equipped with invocation or configuration options to select among the different possible sources of delay, overriding the default specified by the verilog language.

16.2.8 Conflicts Among `specify` *Delays*

The verilog 1364 Std, section 14.3.3, says that when several active inputs change which individually would schedule the same event at different delays, the ***shortest*** delay is the one used.

16.3 Timing Lab 20

Do this work in the `Lab20` directory, using the verilog provided.

Lab Procedure

The verilog for this exercise has been provided, minus timing, in the `Lab20/SpecIt` subdirectory. Before beginning this lab, copy every file in `Lab20/SpecIt` up one level to `Lab20`. The top-level schematic for the SpecIt design is given in Fig. 16.2; the two lower-level schematics are given in Figs. 16.3 and 16.4. As previously discussed, module names and instance names are in different name spaces, so naming an instance exactly the same as its module, while not often recommended, is allowed.

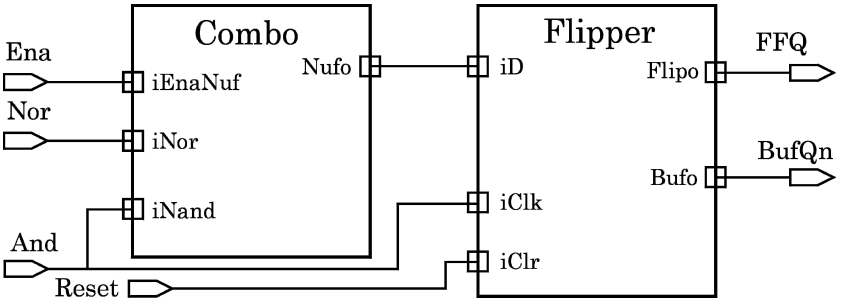


Fig. 16.2 `SpecIt` top-level schematic for timing path exercises. Port names and block instance names (= module names) are shown

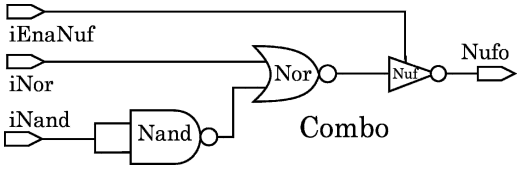


Fig. 16.3 The `Combo` schematic, showing gate instance and port names. Inputs begin with '`i`'; outputs end with '`o`'

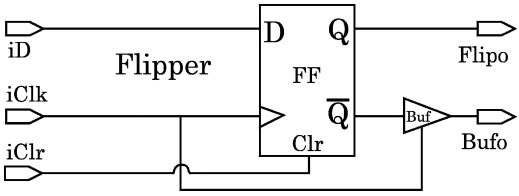


Fig. 16.4 The `Flipper` schematic, showing gate instance and port names. Inputs begin with '`i`'; outputs end with '`o`'

Step 1. Instantiate the `SpecIt` module in a testbench and supply test vectors by means of a 4-bit up-counter. Use the count bits as stimuli on the `SpecIt` inputs in this order: {*MSB*, ..., *LSB*} = { `Reset, Ena, Nor, And` }. Clock the counter with a 100 ns (10 MHz) clock. Verify correct syntax by loading the design in the simulator (see Fig. 16.5).

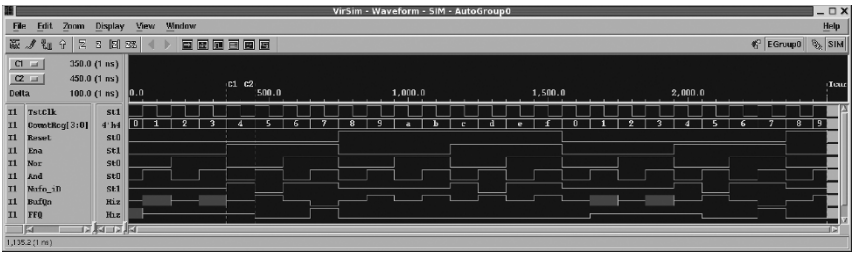


Fig. 16.5 `SpecIt` simulation to verify design correctness

Step 2. Lumped and distributed delays. In the `Nand`, `Nor`, and `Nuf` gate instantiations in `Combo`, assign output delays of 3, 5, and 7 ns, respectively. This will establish a distributed delay totalling 12 or 15 ns between `iNand` and `Nufo`. Check this by simulating briefly (Fig. 16.6).

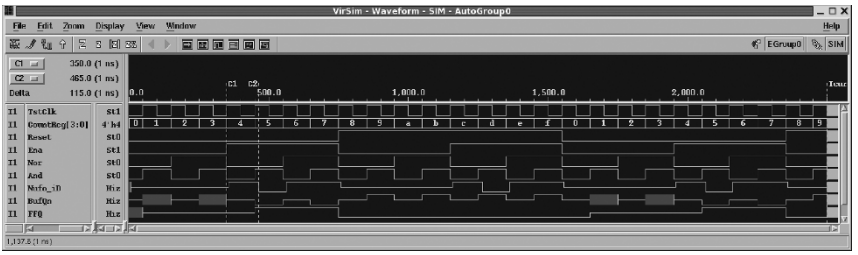


Fig. 16.6 `SpecIt` simulation with distributed delays

What happens when two inputs change at the same time and are distributed different delays?

Try setting a lumped delay of, say, `#10`, on the `Combo` instance in `SpecIt`. `Combo` has just one output, so this delay should be unambiguous. What happens in this potential conflict?

Step 3. Specified path and distributed delay conflict. Make a new copy of `Combo.v` (with the distributed delays of Step 2) in a file named `ComboSpec.v`, but *don't change* the module name. Change your `SpecIt` file list so `ComboSpec.v` is read by the simulator instead of `Combo.v`.

A. In `ComboSpec.v`, add a `specify` block to the `Combo` module which assigns a full-path delay of 10 ns from any input to the output. Use a `specparam` for the time value. Simulate to see what happens.

B. Change the `specify` block delay to 20 ns rise, 21 ns fall, 22 ns turnoff. What happens?

According to the verilog 1364 Std, when delays conflict this way, they must be resolved pessimistically: As previously mentioned, in a change between '`1`' and '`0`', the longest delay is used. As usual, in a change to '`x`', the shortest delay is used; and, in a change from '`x`', the longest delay is used. All the new `specify` delays are longer than the distributed delays on the `Combo` instances.

C. Leaving the Step 3B `specify` values alone, define a `Combo localparam` named `InstTime` and assign it a value of 25. Use `InstTime` to assign each of the distributed delays on the gate instances to 25 ns. Simulate. What happens?

D. Leaving the Step 3C distributed delays as-is, define another `localparam` `tZ`= 30, and use it to assign the value to your `specparam` for turnoff time. Simulate. This shows another benefit of using parameters to change the timing of a model.

Note: Your simulator may refuse to use a generic named constant (`parameter` or `localparam`) to define a `specparam`; this is incorrect and a minor nuisance. A workaround might be to avoid `specparams` and use `localparams` instead.

E. In the Step 3D model, try changing the full-path assignment to a parallel-path assignment without changing anything else. What happens? Your simulator should issue an error or at least a warning when you do this.

Step 4. Specified parallel and full delay conflict. Copy `Flipper.v` to a new file `FlipperSpec.v`, keeping the module names the same. Use your original `Combo.v` (no instance delay) in combination with `FlipperSpec.v` for simulation in this Step.

A. Use `specparams` in a `specify` block in the `Flipper` module (not the `DFFC` module) to assign a parallel-path delay to both outputs of `Flipper`: Use a sum in the specify block to represent that `Bufo` adds a delay of 1 ns to any change on its input or control, and that clocked delays are 5 ns to a rise on `Flipo` and 8 ns to a fall on `Flipo`.

For example,

```
specparam tClkQR=5, tClkQF=8, dBuf=1, tClr=2;
...
(posedge iClk => Bufo) = (tClkQR+dBuf, tClkQF+dBuf);
...
```

B. But, assume that a negative clock edge turns off `Bufo`; add a turnoff delay of 1 ns to the `specify` block. Use only rise and fall delays.

C. Also assign a parallel-path delay of 2 ns to a clear of `Flipo`, appropriately adding delay to `Bufo` for that clear. Simulate to see the result.

D. With the changes in **A** – **C** imposed, add a full-path delay of 30 ns in `Flipper` from `iClk` to both output ports (using a list). Simulate; 30 ns is large enough that you easily should be able to see a 30-ns shift in the waveforms, if one occurs.

E. Replace your 30 ns full-path assignment in D with one to 0 ns. Simulate to see the effect. Question: When there is a conflict between different delays to the same port pin, which delay is used by the simulator, the shorter or the longer one?

Step 5. Using your Step 4E design for `Flipper`, reinstall the delays into `Combo` that you had at the start of Step 2, and simulate. You should be able to see waveforms the same as in Fig. 16.7.

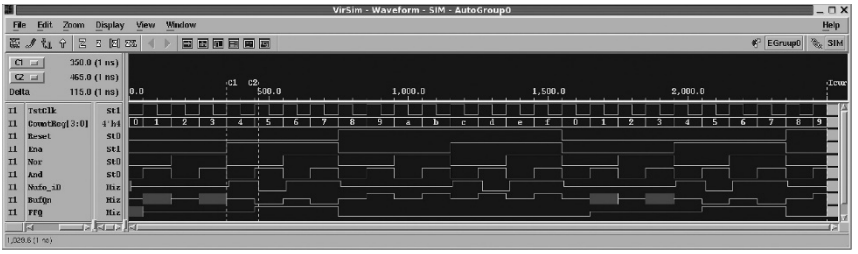


Fig. 16.7 A `SpecIt` Step 5 verilog source simulation

Synthesize `SpecIt`. Be sure to compile the correct files. Read the timing report; what happened to the timing in the modules?

Write out an SDF file (recall *Week 1 Class 1*) and look at it briefly in a text editor. Notice that there are timing triplets everywhere. We shall study this kind of file later in the course. If you should simulate the resulting netlist, you would see waves about the same as in Fig. 16.8.

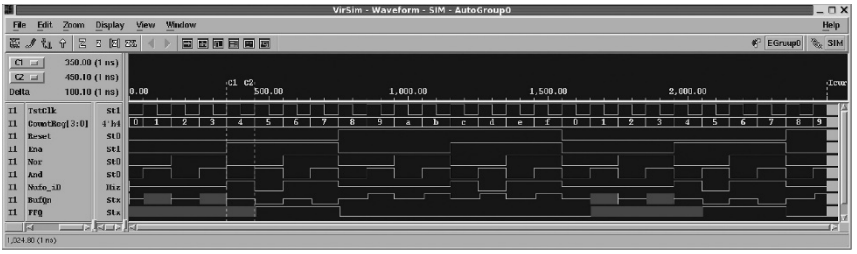


Fig. 16.8 The `SpecIt` Step 5 verilog netlist simulation with SDF back-annotated timing

16.3.1 Lab Postmortem

How does the simulator resolve conflicting delay scheduling times when the delays differ but the final state is the same?

What about when the final states differ?

16.3.2 Additional Study

Read Thomas and Moorby (2002) section 6.6 on `specify` block delays.

Optional Readings in Palnitkar (2003)

Read sections 5.2 and 6.2 on gate-level and assignment-statement timing.
Read chapter 10 on path and other delays.
Try section 10.6, problems 1–3.

Chapter 17
Week 9 Class 1

17.1 Timing Checks and Pulse Controls

17.1.1 Timing Checks and Assertions

Timing checks have the appearance of system tasks (both begin with '`$`'), but they are not system tasks (IEEE Std 1364, section 15). System tasks are simulated in procedural code; timing checks are concurrent and are allowed only in `specify` blocks. However, some system tasks, such as `$display`, can be used to create assertions which resemble the result of a timing-check violation.

As we saw a while ago (*Week 4 Class 2*), simple system tasks can be used to construct assertion checks in the verilog. While ***assertion*** often is taken technically to refer to something specialized, and while in some languages such as VHDL or System Verilog there is a builtin assertion mechanism, the functionality is just to check some condition on variables or other design objects during a simulation, and to create a warning message, and perhaps an error condition, when the assertion is not fulfilled. An error condition may stop the simulation.

In principle, a synthesizer or static timing analyzer could be provided with an assertion mechanism to issue a message or stop the program when some structure or other condition was encountered during a traversal of the verilog input or during creation of the output.

The *Liberty* library format used for netlist synthesis includes its own timing checks very much the same as those in verilog. However, these checks can not be run by a verilog simulator; they are used (*a*) during synthesis and optimization to avoid violation of constraints and (*b*) during static timing verification.

The difference between assertions, debugging, and code coverage is that *assertions* represent a designer's insight into what might go wrong; *debugging* proceeds after a flaw has been uncovered; and *code coverage* estimates the need for debugging. In this context, a timing check can be viewed as a builtin, specialized assertion mechanism. All a timing check does is issue a message to the computer console when some design constraint has been violated, or some device operating parameter has gone out of range. Simulators typically include an option to copy timing check messages to a log file.

J. Williams, *Digital VLSI Design with Verilog*,

A timing check itself does not change simulator waveforms or the values assigned to variables during simulation. However, there does exist a ***notifier*** mechanism in all verilog timing checks which may be used by the designer to change the course of the simulation as a result of a timing violation.

As already mentioned, timing checks are allowed only in verilog `specify` blocks; they can not be put in `always` blocks or in procedural code such as `tasks` or `functions`. Timing checks sit in their `specify` blocks, wired into the rest of the design, and they issue messages when the simulation goes wrong in some subtle, timing-related, way. The conditions they report often would go unnoticed by someone looking for violations in the simulator output waveforms.

To summarize the features of timing checks:

- They all use the syntax, $*name_of_timing_check* (*argument_list*);
- Functionally, they are predefined assertions.
- They differ from procedural assertions:
  - They are allowed only in `specify` blocks.
  - They include builtin triggering logic.
  - They run concurrently.
  - They are part of the simulator and so incur little runtime overhead.

17.1.2 Timing Check Rationale

Timing checks usually are the best ways of enforcing device hardware operating specification limits during simulation. In terms of the functionality of the design being simulated, all timing checks are based on a ***reference event*** and a ***data event***. These events may be scheduled on one, or on more than one, variable in the simulation. There is no connection here between usage of the word *data*, and "data" in the design as contrasted with "control" or "clock". The timing check imposes certain conditions on these two simulation events; and, so long as the conditions (equivalent to a logical expression) are true, the timing check does nothing. Generally, the reference event is conceived of as fixed in time, and the data event is conceived of as varying within or beyond a violation limit.

Two related technical terms are ***timestamp*** and ***timecheck***. In the timing check, whenever the first one of the events is scheduled in simulation time (it may be either the reference or the data event), a *timestamp* is recorded. If and when the other event is executed, a *timecheck* is done to check the conditions on the two times.

The time limit in these checks often defines an open interval in simulation time the endpoints of which do not trigger a violation. A time limit of 0 provides a convenient way to disable any timing check, with the sole exception of the skew-related checks.

By default, a timing check triggers only once per timestamp event, even when more than one violating timecheck event has been simulated. Some of the timing checks can be enabled optionally to print multiple messages for repeated violations, for example on different bits of a *timecheck*ed bus.

The limits controlling timing checks must be constants and usually would be defined by `specparams`. The design variables may be vectors; if so, any bit change(s) in the vector mean(s) the same as that change in a one-bit variable; so, only one violation can be triggered (by default) for each such vector change.

Parameters and other inputs may be passed to the timing check by position, only.

17.1.3 The Twelve Verilog Timing Checks

Here is a complete list, grouped according to expected applicability; however, these checks work the same way regardless of the design functionality of the variable(s) checked:

Clock-Clock Checks	Clock-Data Checks	Clock-Control Checks	Data Checks
`$skew`	`$setup`	`$recovery`	`$width`
`$timeskew`	`$hold`	`$removal`	`$period`
`$fullskew`	**`$setuphold`***	**`$recrem`***	`$nochange`

* Avoid these, if possible.

Each of these timing checks is detailed below. Only the required inputs are listed below, except for `$width`, which includes an optional glitch parameter; with that one exception, all optional inputs such as `notifiers` follow the listed ones. The notifier is explained below, but the reader should refer to the IEEE 1364 Std for full documentation of features available in timing checks, such as edge specifiers or remain-active flags.

In QuestaSim, timing checks are treated as assertions, and verilog warning assertions must be enabled explicitly for timing checks to be run.

17.1.3.1 Clock-Clock Checks

`$skew.` This check triggers on an excessively long delay between events on two variables. The delay value must be nonnegative. Typically, the reference (timestamp) event is a clock, and so is the data (timecheck) event. If the data event never occurs, there is no timing violation.

For example,

```
        ref. event          data event     limit expression
  $skew(posedge Clock, posedge GatedClock,     MaxDly);
```

`$timeskew.` This differs from `$skew` in that, by default, if the specified time lapses, a violation occurs whether or not there ever is a data event.

`$fullskew.` This check is the same as `$timeskew`, except that it allows two nonnegative delays. The first delay specifies a limit when the data event follows the reference event; the second, when the reference event is second.

For example,

```
            ref. event R      data event D     R-D limit  D-R limit
$fullskew(posedge Clock, posedge GatedClock,   MaxRD,     MaxDR);
```

17.1.3.2 Clock-Data Checks

$setup. This check triggers a violation when the reference event, usually a clock, occurs too soon after the data event. The time limit must be nonnegative. The trigger-time window begins with the data event, which is the timestamp event for this check.

For example,

```
     data event   ref. event     time limit
$setup(     D,       posedge Clk,   MinD_Clk );
```

$hold. This check triggers a violation when the data event occurs too soon after the reference event, which latter usually is a clock. The time limit must be nonnegative. The trigger-time window begins with the reference event, which thus is the timestamp event for this check.

For example,

```
     ref. event      data event   time limit
$hold( posedge Clk,       D,     MinClk_D );
```

$setuphold. This check requires two time limits, either of which may be negative; otherwise, the syntax follows that of `$hold`, with the setup limit first. It combines the functionality of a `$setup` and a `$hold` check when both limits are nonnegative. Because of error-prone complexity and thus likely false alarms, routine use of this timing check is not recommended. See explanation below.

17.1.3.3 Clock-Asynchronous Control Checks

Conceptually, these checks depend on internal delay characteristics of components to which several signals are applied; typically a clock train and an asynchronous control signal are assumed. The timestamp event is the reference or data event, whichever occurs first in the simulation. Refer to the waves in Fig. 17.1.

$recovery. Recovery refers to recovery time from deassertion of an asynchronous control such as a set or clear, and effective occurrence of a clock edge. Given that the control has been deasserted, how much time must be provided to recover clock functionality? The time limit must be nonnegative.

For example,

```
            ref. event   data event      limit
$recovery(negedge Clr, posedge Clk, MinClr_Clk);
```

$removal. Removal refers to time of occurrence of an effective clock edge and deassertion of an asynchronous control. Given that a clock edge has occurred, how

long before that edge does an asynchronous control have to have removed itself to permit the clock to be effective? The time limit must be nonnegative.

For example,

```
          ref. event   data event     limit
$removal(negedge Clr, posedge Clk, MinClk_Clr);
```

Note: `$removal` doesn't seem to work in the Silos demo version.

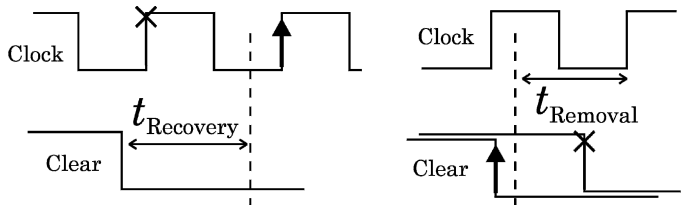


Fig. 17.1 Recovery and removal time limits. `Clear` is asserted high; violating edges are shown with 'x'; passing edges are shown with arrows

`$recrem`. This check permits two time limits, recovery limit first and then removal. When both limits are positive, it has the same effect as a `$recovery` and a `$removal` check, each with its respective limit. It allows for negative time limits. Because of error-prone complexity, routine use of this timing check is not recommended. See the discussion of negative limits. below.

17.1.3.4 Data Checks

`$width`. This check verifies a minimum width of a pulse on a single variable. Whichever edge is provided is used as the timestamp event, and a timing violation is triggered unless enough time has lapsed before the opposite edge occurs. The width value must be nonnegative. A second timing parameter is optional, the glitch threshold, which suppresses the timing violation if the *timestamp*ed pulse is found to be narrower than specified by the glitch threshold.

For example,

```
          ref. edge    width    glitch thresh.
$width(posedge Reset, MinWid,    MinWid/10);
```

`$period`. This check triggers a timing violation if the same edge recurs on a variable within too short a time. The time value must be nonnegative.

For example,

```
        ref. edge     period
$period(posedge Clk, MinCycle);
```

`$nochange`. This check requires an edge on the reference event (= timestamp) to define a level during which no data event (= timecheck) should occur. If a data event occurs during the reference level, a timing violation is triggered. Two offset times

also are required; the first shifts the violation level start event (timestamp edge) and the second shifts the violation level end event (timecheck). The offsets may be negative; a positive value increases the duration of the violation window; a negative value decreases it.

Examples:

```
// Require DBus constant during entire positive phase:
                ref. edge   data event   lead shift   lag shift
$nochange(posedge Clk,        DBus,             0,           0);

// Violation starts 1 before negedge; ends 2 after posedge:
specparam MinSetup = 1, MinHold = 2;
$nochange(negedge Clk, EBus, MinSetup, MinHold);

// Shift the check 1 unit later:
specparam MinSetup = -1, MinHold = 1;
$nochange(negedge Clk, FBus, MinSetup, MinHold);
```

Note: `$nochange` apparently is not implemented in any simulator with which the author is familiar.

17.1.4 Negative Time Limits

We have recommended against any use of the two timing checks, ***setuphold*** and ***recrem***, which permit specification of negative limits. This is because of complexity: A check on design timing should not be more complicated than the design; because, if it were so, an error in the check might cause time lost over false violations or even perhaps an overlooked malfunction in the design.

However, when including a block of IP accompanied by a verilog model which lacks adequate internal timing checks, it may be necessary for the designer to add timing checks on variables at the boundary of the IP block. The internal variables may not be accessible. Under these circumstances, a skewed or even negative time limit may be necessary for the check.

To see why this might be so, as an example, compare the timing in regard to the reference event for simple setup and hold on an internal sequential element such as is shown in Fig. 17.2.

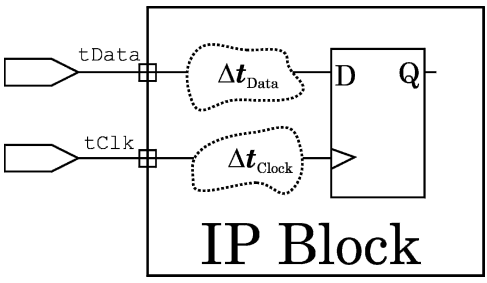


Fig. 17.2 An IP block with an inaccessible flip-flop. Delays within the block are shown as delta-delays. An accessible timing-check data event occurs on the block boundary at time `tData`; a reference event (clock) at `tClk`

There are three different conditions possible: (*a*) No differential delay through the IP logic up to the flip-flop; (*b*) additional delay on the flip-flop clock, perhaps because of a buffer tree; and, (*c*) additional delay on the flip-flop data, perhaps because of combinational processing.

The effects for a mild skew are shown in Fig. 17.3.

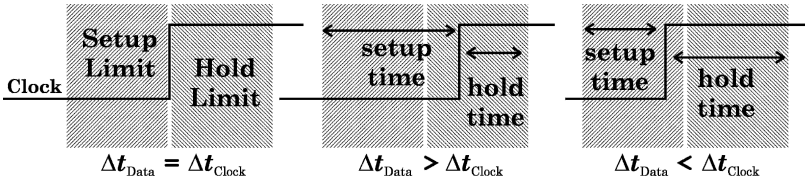


Fig. 17.3 Rationale for skewed time limits. Shaded regions represent fixed-width requirements of the sequential component. If the additional delays are equal, normal setup and hold timing checks have times equal to the limits. When the data are delayed more than the reference, the setup limit time must be increased and hold must be decreased; when reference is delayed more than data, the hold limit time must be increased and setup decreased

When the additional IP delay exceeds a setup or hold limit, one of the times goes through zero and becomes negative. This is shown in Fig. 17.4.

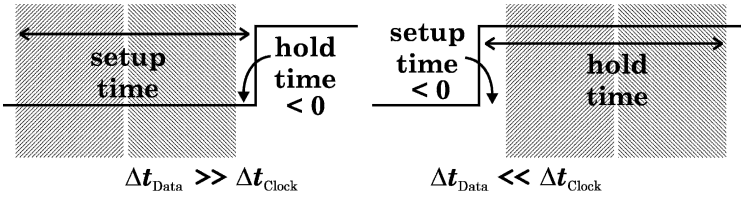


Fig. 17.4 Rationale for negative time limits. When the data or reference is delayed more than the setup or hold time limit for the isolated sequential component, the time at the IP boundary goes through zero and becomes negative

The present author argues against using `$setuphold` or `$recrem`, even though they permit direct entry of negative limits: When the internal delay differences are known, as they have to be to use `$setuphold` or `$recrem` negative values, the delay difference simply should be cancelled by delaying the timing-check data or reference event on a temporary net to cancel the difference. For example, if $\Delta t_{\mathtt{Data}} >> \Delta t_{\mathtt{Clock}}$, then, delay the clock to the timing check by the difference, thus cancelling it, and use the databook requirement directly in the timing check:

```
wire ClockToHold;
assign #tCancel ClockToHold = Clock; // tCancel from IP vendor or experiment.
specify
$hold(posedge ClockToHold, DataIn, tHold); // tHold from data book.
...
```

The tailored, delayed net (in the example, `ClockToHold`) will not be used elsewhere in the design and will be removed by the logic synthesizer (along with notifier `reg`'s, if any – see below). A delayed net may be used with the recommended, nonegative value, timing checks even where it does not cancel the entire IP delay – just so long as the cancellation is enough to avoid the need for negative limits.

17.1.5 Timing Check Conditioned Events

Certain logical conditions are allowed with the variables named in a timing check. These are applied by the operator `&&&`, which logically is the same as `&&` in other contexts. The event must be "true" by `&&&` if the check is to be run. The operators, `==`, `!=`, `===`, `!==`, or $\sim$ are allowed in the expression applied by `&&&`.

In a timing check, the conditioning expression RHS must be a scalar (1-bit) variable expression. A vector is allowed, but only the LSB value will be expressed.

For example, here we don't want a violation if `Ena` is low while `D` changes:

```
$setup( D&&&(Ena==1'b1), posedge Clk, MinD_Clk );
```

17.1.6 Timing Check Notifiers

It is possible to make simulator behavior depend upon a timing check. Perhaps, one would want the simulation to stop on a violation; or, maybe, a detailed message might be issued based on an assertion.

All timing checks allow at least one optional input in addition to those shown above, and the first optional input not given above always is the name of a ***notifier*** `reg`. This is a one-bit `reg` type declared visible to the timing check `specify` block, which is to say, in the `module` containing that `specify` block. This `reg`, of course, becomes part of the simulation model of the design; thus, anything possible on change of a `reg` value can be initiated by a timing violation through a `notifier`. Typically, a simulator system task such as `$stop` would be the only action a designer would want; or, perhaps an assertion failure might be programmed on `notifier` change.

When a timing violation is triggered by a timing check, the `notifier` value is toggled between '`1`' and '`0`'. If the notifier was '`x`' when the violation occurred, it is toggled to '`0`'; if it was '`z`', it remains '`z`'. This last feature provides a way to disable notification without disabling the timing check itself.

The `reg` passed to the timing check must have been declared, but it need not be used anywhere else in the design.

Example:

```
reg Notify;
...
always@(Notify) $stop;
...
 specify
 ...
 $setup( D, posedge Clk, MinD_Clk, Notify );
 endspecify
```

17.1.7 Pulse Filtering

Timing checks do nothing to alter the simulation, unless perhaps by the optional *notifier* feature. However, pulse filtering is an essential part of the simulator behavior.

In normal, default inertial delay, pulses shorter than a gate delay are removed from the schedule of events on the gate input. In verilog, this default pulse filtering is viewed as occurring because two different time limit settings, the ***error limit*** and the ***rejection limit***, happen, by default, both to be equal to the gate delay.

The verilog error limit always is greater than or equal to the rejection limit. If the rejection limit is different from (= less than) the error limit, three possible things may happen to an input pulse with an effect delayed on an output:

(a) the pulse is wider than the error limit: It is delayed but otherwise is left unchanged;
(b) the pulse width is between the error limit and rejection limit: It is delayed and narrowed to a width equal to the rejection limit; or,
(c) the pulse width is below the rejection limit: It is cancelled and never affects the output.

These limits may be set by simulator invocation options or in an SDF file; but, here, we ignore this and present only the `specify` block special `specparam`, `PATHPULSE`.

`PATHPULSE` is the only reserved word in verilog not in lower case. In a peculiar twist of syntax, it is prepended to the two variable names involved, which must name a timing path, with '`$`' delimiters. It changes the type of the timing path involved. The result is used as the name of a `specparam`. For example, to apply `PATHPULSE` to the path between ports `Ain` and `Bout`, one writes,

```
specparam PATHPULSE$Ain$Bout = (r_limit, e_limit);
```

in which `r_limit` is the rejection limit value and `e_limit` is the error limit value. An example is shown in Fig. 17.5. Notice that `PATHPULSE`, like other `specparams`, may be assigned more than one numerical value. Specifying the error limit is optional; assignment of one value in the line of code above would assign

the rejection limit value only, the error limit value being taken to be equal to the gate delay involved. The names in the path must be declared names and may not be bit-selects or part-selects.

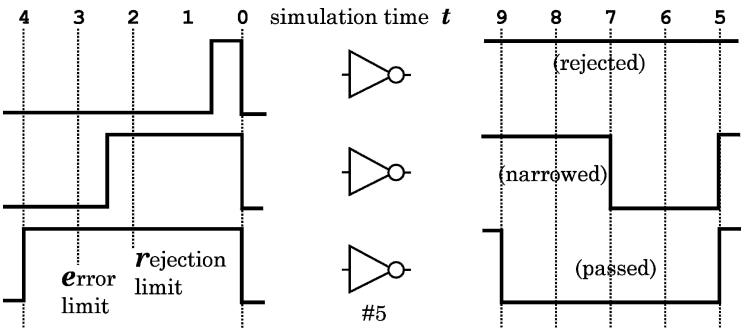


Fig. 17.5 Effect of PATHPULSE limits on pulse filtering. Assume PATHPULSE$ = (2, 3). With no PATHPULSE specification, all pulses would be rejected by simple inertial delay because of the gate delay of 5

It is possible to omit a path and write generically, "PATHPULSE$= (r_limit, e_limit);" or "PATHPULSE$ = r_limit;". This assigns the limit(s) to every path in the entire module. When both this and a PATHPULSE naming a path are present, the one naming the path overrides the generic one on that path.

Example:

```
module NewInertia #(parameter r_limit = 3, e_limit = 4:5:6)
                   (output Z, input A, B, C);
...
specify
  ... (delays; timing checks) ...
  specparam PATHPULSE$    =   r_limit ; // Module default.
  specparam PATHPULSE$B$Z = ( r_limit, e_limit );
endspecify
endmodule
```

A specify block may contain full path delay statements, omitted from the example code above, which name multiple paths; when this is so, the "wildcarded" PATHPULSE$ is applied only to the first path (first input to first output) in any such delay statement. Attempting to apply PATHPULSE selectively to any other path in such a delay statement, other than the first one, has no effect. Thus, PATHPULSE is most easily used for individual paths when those paths are one bit wide and when the timing specifications are parallel-path rather than full-path.

Note: PATHPULSE does not seem to work in Silos or QuestaSim, and it produces 'x' outputs instead of no change in VCS, when pulses are narrower than the rejection limit and the +pathpulse invocation option in force. Under these conditions, the VCS 'x' outputs are accompanied by warning messages.

17.1.8 Improved Pessimism

Recall the discussion of delay pessimism in *Week 7 Class 2*. Pessimism improves chances of success, but it can be wasteful.

In addition to `PATHPULSE`, there are four other relevant reserved `specparam` types: `pulsestyle_onevent`, `pulsestyle_ondetect`, `showcancelled`; and, the negation, `noshowcancelled`.

Unlike `PATHPULSE`, these are used by declaring lists of output variables assigned to them which are to be simulated with improved pessimism, which is to say, with increased use of 'x' scheduling. They must be declared in the `specify` block before any path which is assigned a delay including one of the variables listed.

`pulsestyle_onevent`. This is the default behavior: When a contention yielding an 'x' exists, the contending events are scheduled normally; and, at the simulation time at which the contention first occurs, an 'x' is scheduled.

`pulsestyle_ondetect`. This is more pessimistic: As soon as the simulator has computed the contention, an 'x' is scheduled. This shifts the leading edge of the 'x' level to some time earlier than when it would have been had the default had been in force. Essentially, the output of a gate goes to 'x' as soon as an input event arrives which would cause that 'x'. The on-detect edge of the 'x' tells the designer how early a possibly uncontrolled state has occurred in the simulated design. In a big design, triggering a `$stop` assertion on detection, rather than on event occurrence, can save considerable debugging time.

`showcancelled`. Sometimes, different rise and fall delays may cause the leading edge of an 'x' event to occur at the same time as the lagging edge, or even before, creating a zero or negative duration of the 'x' level. Such events are cancelled silently by the simulator, by default. Assigning an output pin to this `specparam` means that the simulator will schedule a zero or negative-width output pulse of 'x' at the original leading-edge output time; this pulse will be of the input-pulse width. If, in addition, this pin has been assigned to `pulsestyle_ondetect`, the output leading edge of the 'x' will be advanced to the detection time, widening the output 'x' pulse.

For debugging purposes, it is possible to select variables in a `showcancelled` list to be restored to default behavior by assigning them to the **`noshowcancelled`** `specparam`.

Note: None of these pessimism improvements seems to work in Silos, QuestaSim, or VCS. However, these simulators have invocation or interactive options providing similar functionality.

17.1.9 *Miscellaneous* time-*Related Types*

The verilog standard includes data types intended to be used in `specify` blocks. In addition to `specparams`, which are commonly used, these are `time` and `realtime` types.

A **`time`** is an unsigned `reg` type of predetermined width which is guaranteed to be at least 64 bits. It is meant to be used in testbenches or in conjunction with

long-duration timing checks. Assigning a `time` reg to a wide wire type yields a vector value which may be referenced in a `specify` block.

A **`realtime`** `reg` is identical to a `real` in all respects except name. It is reminiscent of the `tri` net, which is identical to a `wire` except in name.

Both `time` and `realtime` may be used anywhere in a `module` where a `reg` type is allowed. However, use of these types in design should be considered carefully. They add nothing to functionality and make the syntax a little more complicated. Instead of invoking a type with time-related associations, it is better to name the declared variables so that every use of their identifier recalls that they are time-related. Also, these types, having no unique functionality, may not be implemented in all design tools.

17.2 Timing Check Lab 21

Work in the `Lab21` directory. A subdirectory named `PLLsync` has been prepared for you there; it contains the entire `PLLsync` design from `Lab10` (*Week 4 Class 1*).

Lab Procedure

Be sure that your simulator is enabled to process `PATHPULSE` assignments and timing checks.

Recall that the `PLLsync` design has the following component blocks: At the top level, `PLLsync` and `Counter4`; both in `.v` files named for the modules. There is a testbench in `PLLsync.v` named`PLLsyncTst`. In a subdirectory named `PLL`, the PLL resides in five files: An include file and four others, the latter ones containing modules named `PLLTop`, `ClockComparator`, `MultiCounter`, and `VFO`.

The VFO in this model will exhibit its delay-adjusting functionality even with a constant clock frequency; we shall use this to demonstrate timing checks.

Step 1. Preliminary simulation. Change to the `PLLsync` subdirectory. Using the testbench provided, run the `PLLsync` simulation for a very long time, say 25,000 ns. Notice that with a `` `VFO_MaxDelta `` of 2 (in the PLL include file), the VFO period oscillates within a 4 ns range. If you display the `VFO_Delay` variable value in `VFO`, you will see it switch in increments of 1 ns between 14 and 18 ns (the value should average 15.625 ns for a PLL clock with frequency of 32 MHz). This means that the counter counts with a delay of between 28 and 36 ns.

Step 2. Setup check. Suppose we wish to use a positive edge on the `Sample` command to the VFO to sample the count value in the counter. We want to be sure to allow a setup time of 5 ns between these signals. Our recently renamed module, `PLLsync`, originates the `Sample` command as `SyncPLL`, and it also receives the counter count output as `Behavioral`. So: Install a `$setup` check in `PLLsync` which issues a violation at 5 ns. Make the limit depend on a `specparam` value. Simulate and watch the simulator console output window. Set the limit to 0 ns to disable this check.

Step 3. Hold check. Add a hold check, requiring 8 ns, to the Step 2 timing problem. Simulate. After seeing the violations, set the limit to 0.

Step 4. Recovery and removal. Modify your testbench to apply two successive `ClearIn` pulses to `PLLsync`; they should be separated by 50 ns, and each should last 500 ns. The separation should be centered approximately on a clock edge around a simulation time of 1000 ns or so.

Add a recovery check of 100 ns for `ClockIn` recovering from `ClearIn` on the falling edge of `ClearIn`; add a removal check of 100 ns on these edges. These values should trigger both timing violations. Simulate. Then, shorten the check times so no recovery or removal violation is reported.

Step 5. Width and period checks. In `PLLsync`, add a width check to issue a violation when `ClearIn` is low for less than 100 ns. Simulate to see the result; then, disable this check by setting the time to 0. Add a period check to be sure that the PLL clock output period is not less than 30 ns, based on a positive edge. Disable this check after viewing the result.

That's all for the `PLLsync` design for now. For the rest of this lab, copy your `DFFC.v` model (D flip-flop with clear) from the `Lab08` exercises (*Week 3 Day 2*) to the `Lab21` directory.

Do the next Steps in order; they depend on one another:

Step 6. Pulse filtering. What happens when the input changes too rapidly?

To answer this, first change the `DFFC` model:

Remove the timescale setting from the `DFFC` file (there will be one in the testbench). Change the old continuous assignment delays of `#1` on `Q` and `Qn` to negligible but nonzero ones of `#0.001` (1 ps).

Add a `specify` block after the `module` declarations, setting delays as follows. Use mnemonically-named `specparams`:

Path delay from clock posedge to `Q` = 1000 ps rise and 800 ps fall; to `Qn` 1100 ps rise and 850 ns fall.

Full delay from clear to `Q` or `Qn` = 700 ps rise and 900 ps fall.

Instantiate the `DFFC` as a toggle flip-flop in the testbench (`DFFC_Tst`) which has been provided for you in the `Lab21` directory: This testbench gradually decreases the clock period from 10 ns to 10 ps. It also provides a regular clear of 50 ns period and 50% duty cycle. It sets `` `timescale 1ns/1ps `` to resolve events on a 1 ps granularity.

To make a toggle flip-flop, wire the `Qn` to the `D` input.

A. Visual check. Simulate (the float calculations will make this take a noticeable time), and examine the waveform. What is the clock frequency at which your D

flip-flop functionality becomes unreliable? Hint: `Q` and `Qn` both must work. How does this relate to the delays in the model? See Fig. 17.6 and 17.7 for one result.

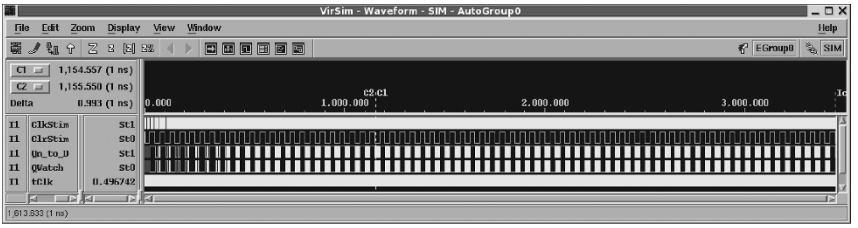


Fig. 17.6 Overview of the `DFFC` simulation with slowly increasing clock frequency

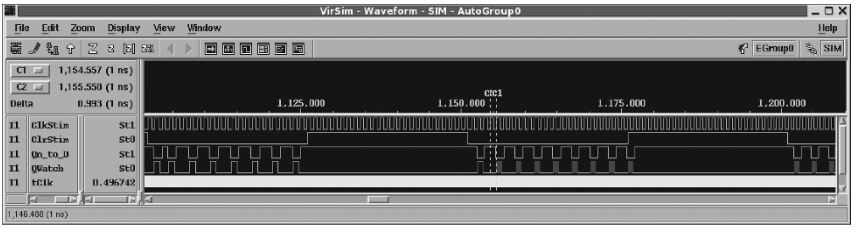


Fig. 17.7 Close-up of the first timing failure in this `DFFC` simulation

B. Reliability enforcement. Assume we want an error limit of 1100 ps and a rejection limit of 500 ps for any pulse on any path to `Q` or `Qn` in our `DFFC` model. Add one or more `PATHPULSE` `specparams` to the `DFFC` `specify` block to accomplish this. Simulate to see the result; then, comment out the `PATHPULSE`(s).

Step 7. Width. We would like to be sure that `Q` and `Qn` stay high for at least 1 ns when they go high.

A. Add `$width` timing checks for this. Simulate.

B. Change the minimum-width limit to 0.850 ns and simulate. This should produce a limited number of violations.

C. Declare a new `reg` named `Notify` in `DFFC` and connect it to your `$width` timing checks as the fourth calling parameter; assign a glitch reject of `0` as a placeholder third calling parameter. Add the following `always` block below your `specify` block:

```
...
$width(posedge Qn, twMinQQn, 0, Notify);
endspecify
//
always@(Notify) $stop;
```

Now run the simulation again. You can continue the simulation at your leisure, after each `$width` violation.

Change your minimum width `specparam` values to 0.500 ns to inhibit width violations, but leave in the width checks and `Notify` for now.

Step 8. Setup and hold. The pathological toggling at high clock frequency can be revealed several ways. One way is by adding setup and hold timing checks.

A. For the relationship between `Clk` and `D`, add a setup check for 1 ns and a hold check for 500 ps. Simulate (you may wish to stop early).

B. Connect the `$setup` and `$hold` notifiers to your `Notify` reg and simulate again. Decrease your setup and hold `specparam` values until all setup and hold violations vanish. Obviously, if the clock is going down to 10 ps, then it would be a good idea to start both limits here. What was the duration of the shortest one of each violation, and what was the earliest simulation time at which it occurred?

C. In the testbench, change the timescale to `10ns/1ns`. Run the simulation with setup and hold at the lowest limits which previously were causing violations. What happens? Notice how quickly (in wall-clock time) the simulation finishes with such a coarse time resolution.

Decrease the resolution limit to `10ns/100ps` and then `10ns/10ps`, simulating each time. What is the effect of the resolution on timing violations and the way they are reported?

Restore the timescale to `1ns/1ps`, and disable the old `$width` check, as well as `$setup` and `$hold`, before continuing.

Step 9. Skew check. Add a `$skew` check between clock and clear, so that a violation will occur if clock ever goes high more than 49.99 ns after clear does. Simulate. Then, disable this check; to do this, you will have to comment out the check or set the limit to 50 ns or more..

Step 10. Recovery and removal. These are perhaps the most easily misunderstood timing checks. They are designed to check on the relation of an asynchronous control, such as a set or clear, and a lower-priority synchronous control such as a clock.

A. Recovery check. Use `$recovery` to check when a `posedge Clk` has not been given at least 10 ps to recover after a `negedge` (deassertion) of `Clr`. Simulate. When does the first violation occur? Set the limit to 0 to disable this check.

B. Removal check. Use `$removal` to check that `Clr` has been removed (`negedge`) at least 10 ps before a subsequent `posedge Clk`. Simulate (see Fig. 17.8). Set the limit to disable the check when done.

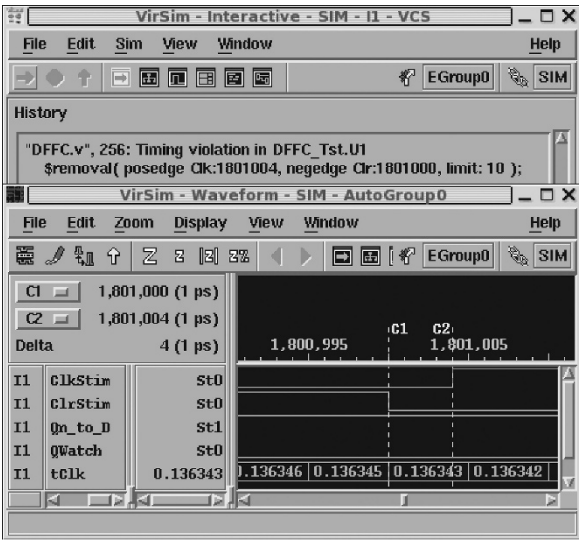


Fig. 17.8 A `$removal` timing-check violation, close-up

17.2.1 Additional Study

Read Thomas and Moorby (2002) sections 8.1 and 8.4.4 for some insight into inertial delay.

Optional Readings in Palnitkar (2003)

Read the section 10.3 on timing checks.
Do section 10.6, problems 6–8.

Chapter 18
Week 9 Class 2

18.1 The Sequential Deserializer

Let's review our serdes project and determine what remains to be done.

We introduced our serdes project in *Week 2 Class 2* and made significant progress on the `SerDes` PLL in that Week's PLL Clock Lab 6, including a serialization frame encoder in Step 8. We added serial frame synchronization in PLL Lock-In Lab 10 (*Week 4 Class 1*). We also completed a FIFO in FIFO Lab 11; however, this FIFO causes functionally incorrect synthesis because of unusual sensitivity lists and the resultant erroneous latch inference. Nevertheless, the FIFO does simulate correctly; and, because of this, we were able to refine the design of our deserialization decoder, `DesDecoder`, in Serial-Parallel Lab 16 (*Week 7 Class 1*).

In the present lab, we shall redesign the PLL for correct synthesis; we shall hold revision of the FIFO for later. We also shall assemble and simulate the entire `Deserializer` by the end of the present lab.

So far, Fig. 18.1 shows where we are in the overall design, mostly from a data flow perspective.

J. Williams, *Digital VLSI Design with Verilog*,

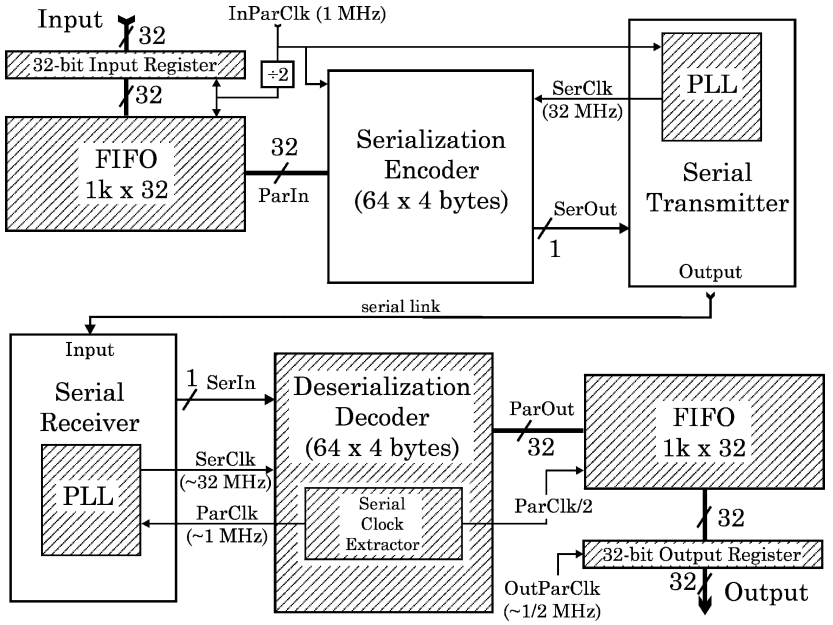


Fig. 18.1 Data flow and clock distribution in the serdes project. The upper half is the `Serializer`; the lower half, the `Deserializer`. Overall design updated as in Lab 16. Hatched blocks were completed separately in previous labs. Neither the PLL nor the FIFO will synthesize, but both simulate usably

We'll first fix the PLL so it will synthesize correctly; then, we can complete the `Deserializer` for simulation and almost-correct synthesis. After the PLL, most of the remaining tasks of today's lab will be organizational.

18.2 PLL Redesign

The main problem with the current PLL is in the VFO: It depends on programmable verilog delays in an oscillator implemented by one, delayed nonblocking assignment statement. To get a predictable (but wrong) netlist out of this design, we added a preprocessor macro switch which forced the synthesizer to see a delayed blocking assignment instead of a delayed nonblocking one.

There is no really good way to design an analogue device such as a PLL in a digital language such as verilog. We can not create a variable capacitor, something that charges gradually on each clock, to control a VCO (variable-capacitance oscillator) for our PLL. However, we can create a fast counter which adapts its count gradually on each clock.

18.2.1 Improved VFO Clock Sampler

Recall that we used a `Sample` pulse input, originating external to the PLL, in our 32x `Lab06`, `Lab08`, and `Lab10` PLL designs to reduce the rate at which the `VFO` frequency was adjusted. We did not use edge-averaging or any other technique for smoothing or refining the `ClockComparator`'s output. The sampling pulse idea, shown in Fig. 18.2, was not essential, but it did prevent the `VFO` from changing its frequency because of every tiny clock misalignment.

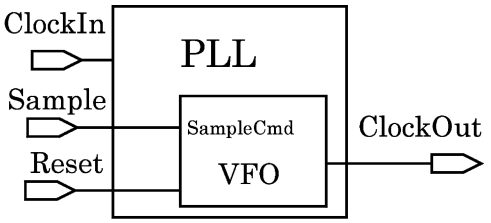


Fig. 18.2 The old `Lab06` `VFO` comparator-sampling command

To make our PLL self-contained, we shall not any more use an external sampling pulse for the VFO: We can build into the PLL itself a sampling clock which triggers a `Comparator`-based adjustment, once every cycle of one of the available, approximately 1-MHz clocks.

Looking ahead to synthesis, to generate a properly set-up sampling edge, we must allow our `Comparator` counters to settle at their current counts before each sampling. This set up may be provided easily by running the external input clock through a library delay cell (see Fig. 18.3) to retard the sampling edge for some reasonable, brief period. The counters then will be clocked by the other, VFO-generated, clock.

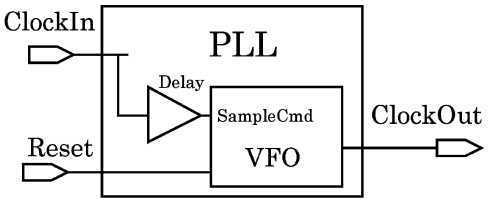


Fig. 18.3 The new `VFO` comparator-sampling command

The verilog for the connection to such a library delay cell would be as follows:

```
module PLLTop (output ClockOut, input ClockIn, Reset);
// ...
Library_DelayCell DelayU1 (.Z(SampleWire), .I(ClockIn));
//
// (dont_touch DelayU1  synthesis directives)
//
VFO VFOU1 ( .ClockOut(MHz32), .AdjustFreq(AdjFreq)
          , .Sample(SampleWire), .Reset(Reset) );
```

18.2.2 Synthesizable Variable-Frequency Oscillator

Attempting to synthesize the old, `Lab06` PLL will produce nothing usable. As you may recall, we used a verilog macro to prevent the synthesizer from seeing the delayed nonblocking assignments and to substitute a nonoscillating delayed blocking assignment.

The netlist from the synthesized `Lab06` VFO does have an oscillator, but it can't modify its frequency, and it will run at an unusably high frequency, depending on the synthesis constraints. Typical constraints produce a *nor* gate oscillating at a constant 6 GHz. The delayed blocking assignment workaround in the old PLL VFO amounted to an erroneously inferred latch and was synthesized to logic corresponding to the schematic of Fig. 18.4.

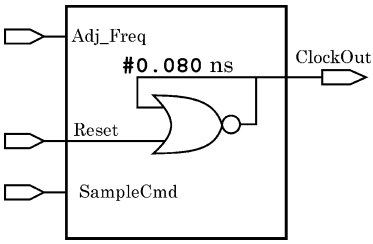


Fig. 18.4 The Lab 6 synthesized `VFO` netlist

For a redesign which will guarantee synthesis, we shall use the high speed components available in our synthesis target library to implement a very fast oscillator based on a chain of library delay cells. There will not be any verilog delay involved, except as characterized for the delay cells, plus the propagation delay of an inverting gate.

To control the fast-oscillator frequency and use it as the VFO, we can clock a fast counter and use the counter overflow as the PLL output clock. By varying the count at counter overflow, we vary the VFO frequency. We require a lower frequency than 6 GHz for a reliable fast counter.

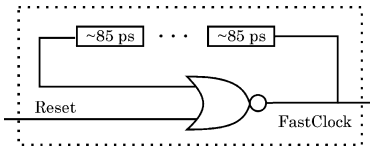


Fig. 18.5 The new, synthesizable `VFO` internal `FastClock`

The frequency is controlled by a delay line, which can be created of any configurable length by means of a verilog `generate` statement. Using several, calibrated library delays of about 80 or 90 ps each will tend to reduce the delay variation in fabrication of the required inverter. The oscillator output would be used internally by the VFO and is named `FastClock` in the schematic of Fig. 18.5 (the *nor* gate replaces an inverter, because it permits a clock phase `Reset` initialization).

The verilog for this oscillator follows:

```
reg FastClock;
wire WireToDelay, WireFromDelay;
assign WireToDelay = ~FastClock; // oscillation here.
// ------------------------------------------------------
// The always block allows initialization:
always@(WireFromDelay, Reset)
  if (Reset==1'b1)
       FastClock <= 1'b0;
  else FastClock <= WireFromDelay;
// ------------------------------------------------------
// The delays control the (fixed) fast oscillator speed:
LibraryDelayCell Delay0( .Out(Wire1), .In(WireToDelay) );
LibraryDelayCell Delay1( .Out(Wire2), .In(Wire1) );
...
LibraryDelayCell DelayN( .Out(WireFromDelay), .In(WireN) );
//(synthesizer dont_touch on all Delay* instances)
```

The delay line is shown unrolled in the code fragment above; it must be flagged with a synthesizer don't-touch to prevent its removal during synthesizer optimization. Because the delay line is an integral part of the VFO, the don't-touch is better done by a comment in the verilog rather than by a command in the `.sct` file. This verilog is entirely synthesizable either as unrolled in the code shown above or as `generate`d.

We can make a VFO based on the new oscillator by tapping the `FastClock` and using it to clock a fast, programmable counter, and using a selected count value on `VaryFreq` to define the VFO clock edges; this is shown in Fig. 18.6.

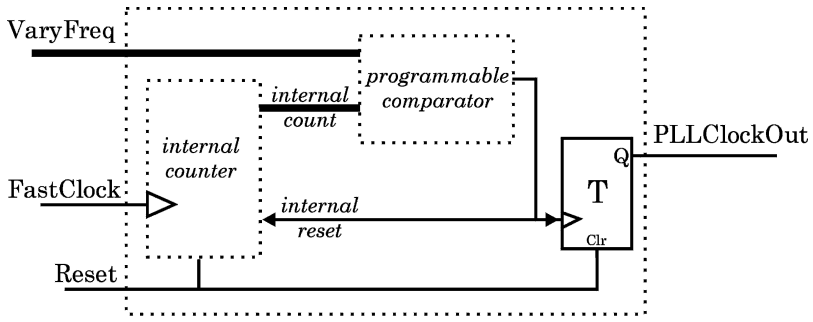


Fig. 18.6 The new `VFO` is based on an internal fast oscillator (`FastClock`) and programmable counter

The VFO internal comparator (not to be confused with the PLL `ClockComparator` module) stores the latched value of a programmable limit

which is used to reset the VFO internal counters. A `VaryFreq` adjustment is allowed to increment or decrement the programmable limit. The reset also toggles a flip-flop which creates the actual VFO output clock for the PLL.

The verilog would be somewhat like this:

```
// Assume a VaryFreq vector declared which
//                   sets the VFO frequency:
reg[HiBit:0] Count;
reg PLLClockOut;
always@(posedge FastClock, posedge Reset)
  if (Reset==1'b1)
     begin
     PLLClockOut <= 1'b0;
     Count       <= 'b0;
     end
else begin
     if (Count>=VaryFreq) // Programmable limit.
          begin
          PLLClockOut <= ~PLLClockOut;
          Count       <= 'b0;
          end
     else Count <= Count + 1;
     end
```

For our 32 × PLL, we need a library fast enough to oscillate and count up to some small integer value with enough precision to vary the frequency of `PLLClockOut` reasonably around 32 MHz. If we assume a 16 ns half-period and 1-ns precision, we need a counter which can count to about 20 in 20 ns, which implies a 5-bit counter clocked at around 1 GHz. This speed is attainable easily in 130-nm or lower library technology.

We shall look at a synthesizable 1 × PLL later.

18.2.3 Synthesizable Frequency Comparator

We also require a comparator which does not have a strange, latching sensitivity list. The current 32 × PLL is clocked by the input, approximately 1 MHz clock and samples the `VFO` (PLL output) clock independently in a change-sensitive block somewhat like this:

```
// OLD, unsynthesizable VFO:
always@(ClockIn, Reset)          // The input system clock.
  if (Reset==1'b1)
      ... (stuff) ...
  else if (CounterClock==1'b1)   // The PLL MultiCounter output clock.
            VarClockCount = VarClockCount + 2'b01;
       else begin
            case (VarClockCount) // The comparator object.
                2'b00: AdjustFreq = 2'b11;
                2'b01: AdjustFreq = 2'b01;
              default: AdjustFreq = 2'b00;
            endcase
            VarClockCount = 2'b00;
            end
```

Although this defined a PLL which simulated correctly, it is a classical case of latch inference which the synthesizer can not interpret meaningfully.

We can avoid latch inference entirely by using edge sensitivity. A reasonable improvement of the above, then, might be to write verilog for two very small counters, say, 2 bits each, one clocked by the external PLL 1 MHz input clock and the other by the PLL internal approximately 1 MHz counter overflow clock; the counts could be compared on the edge of either clock to estimate the relative speed of the clocks. We could issue an adjustment of the VFO frequency only on a difference in the two counts.

By delaying the external `ClockIn` as shown in Fig. 18.7, we can trigger a compare on the PLL internal clock edge, thus avoiding a clock-race condition and almost always ensuring proper set up on every compare of the two, approximately 1 MHz clocks. Note that this delay is different functionally from the one used for the PLL internal sample command.

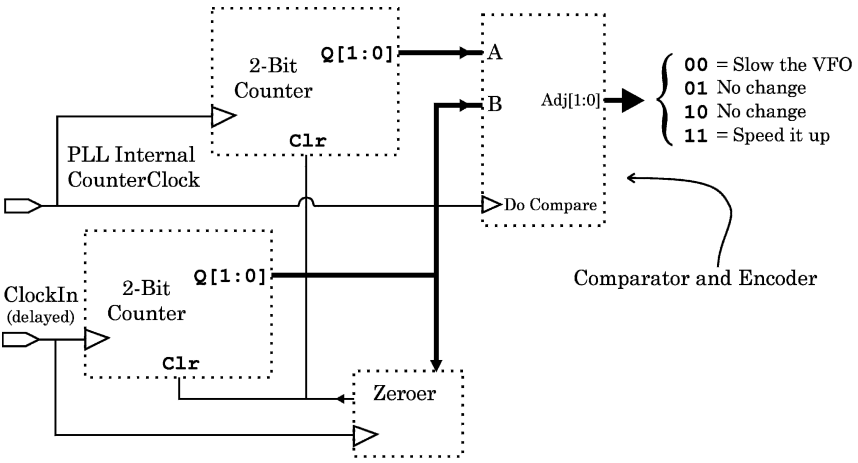


Fig. 18.7 The synthesizable, edge-sensitive `ClockComparator`. Dotted bundaries indicate functional blocks within the one module. The external `Reset` distributed to all sequential logic is omitted for simplicity

Two 2-bit counters would count continuously the positive edges of the 32 × PLL `MultiCounter` output clock (about 1 MHz) and the input 1 MHz clock which is used as a stimulus for the PLL. On every positive edge of the PLL clock, a comparison is made of the relative edge-counts of the two counters, generating a continuously-updated adjustment decision to be used by the VFO. The `Zeroer` monitors the `ClockIn` count and reinitializes both edge counts simultaneously, every time it receives a count of 3.

The VFO adjustment decision logic may be implemented with nested `case` statements located in an `always` block clocked by the PLL `MultiCounter` overflow clock:

```
case (ClockIn_EdgeCount) // The external ClockIn edges.
    2'b00: case (VFO_EdgeCount) // The MultiCounter overflow edges.
             2'b00: AdjustFreq = 2'b01;  // No change.
           default: AdjustFreq = 2'b00;  // Slow the counter.
           endcase
    2'b01: case (VFO_EdgeCount)
             2'b00: AdjustFreq = 2'b11;  // Speed up the counter.
             2'b01: AdjustFreq = 2'b01;  // No change.
           default: AdjustFreq = 2'b00;  // Slow the counter.
           endcase
    2'b10: case (VFO_EdgeCount)
             2'b10: AdjustFreq = 2'b10;  // No change.
             2'b11: AdjustFreq = 2'b00;  // Slow the counter.
           default: AdjustFreq = 2'b11;  // Speed up the counter.
           endcase
  default: case (VFO_EdgeCount) // Includes 2'b11 for initialization:
             2'b11: AdjustFreq = 2'b10;  // No change.
           default: AdjustFreq = 2'b11;  // Speed up the counter.
           endcase
endcase
```

18.2.4 Modifications for a 400 MHz 1 × PLL

All the above verilog is meant for the 1 MHz, 32:1 PLL of our serdes project. But, what about the 400 MHz, unsynthesizable 1 × PLL we discussed in *Week 2, Class 2* before `Lab06`?

The special concern is with the higher PLL output frequency. The internal counters described above will work unchanged, except that the VFO `FastClock` must be speeded up by limiting its delay chain to fewer delay elements. Also, the fastest available 90-nm library inverter should be instantiated structurally as part of the VFO oscillator.

In verilog, with `` `NumElems `` set to 3 instead of 5, the new `FastClock` generator becomes this:

```
reg FastClock;
wire[‘NumElems:0] WireD;
//
generate
  genvar i;
  for (i=0; i<‘NumElems; i = i+1)
    begin : DelayLine
    DEL005 Delay85ps ( .Z(WireD[i+1]), .I(WireD[i]) );
    end
endgenerate
//
always@(Reset, WireD)
  begin : FastClockGen
  if (Reset==1’b1)
       FastClock = 1’b0;
  else // The free-running clock gets the output of the delay line:
       FastClock = WireD[‘NumElems];
  end
// The instantiated inverter:
INVD0 Inv75ps ( .ZN(WireD[0]), .I(FastClock) );
```

Our TSMC library `DEL005` component provides a delay of about 85 ps. The required don’t-touch directives are omitted for brevity.

With the sampling setup delay element in the containing `PLL` module, the synthesizable `VFO` adjustment is as above:

```
always@(posedge ClockIn, posedge Reset) // ClockIn delayed for setup.
  begin : FreqAdj
  if (Reset==1’b1)
       DivideFactor <= ‘InitialCount;
  else begin
       case (AdjustFreq)
         2’b00: // Adjust f down (delay up):
                if (DivideFactor < DivideHiLim)
                   DivideFactor <= DivideFactor + ‘VFO_Delta;
         2’b11: // Adjust f up (delay down):
                if (DivideFactor > DivideLoLim)
                   DivideFactor <= DivideFactor - ‘VFO_Delta;
       endcase // Default: leave DivideFactor alone.
       end
  end // FreqAdj.
```

The `‘InitialCount` is set to half of the maximum count in the `FastClock`-creating programmable counter.

In the faster 1 × design, the `Comparator` decision logic can to be biased to be more sensitive to real changes:

```
always@(posedge PLLClock, posedge Reset) // The delayed PLL ClockIn.
  begin : EdgeComparator
  if (Reset==1'b1) AdjustFreq = 2'b01;
  else
  case (ClockInN) // Count from the PLL external input clock.
    2'b00: begin
           case (PLLClockN) // Count from the VFO output clock.
             2'b00: AdjustFreq = 2'b01; // No change.
             2'b01: AdjustFreq = 2'b00; // Slow the counter.
             2'b10: AdjustFreq = 2'b00; // Slow the counter.
             2'b11: AdjustFreq = 2'b00; // Slow the counter.
           default: AdjustFreq = 2'b01; // No change.
           endcase
           end
    2'b01: begin
           case (PLLClockN)
             2'b00: AdjustFreq = 2'b11; // Speed up the counter.
             2'b01: AdjustFreq = 2'b01; // No change.
             2'b10: AdjustFreq = 2'b00; // Slow the counter.
             2'b11: AdjustFreq = 2'b00; // Slow the counter.
           default: AdjustFreq = 2'b01; // No change.
           endcase
           end
    2'b10: begin
           case (PLLClockN)
             2'b00: AdjustFreq = 2'b11; // Speed up the counter.
             2'b01: AdjustFreq = 2'b11; // Speed up the counter.
             2'b10: AdjustFreq = 2'b10; // No change.
             2'b11: AdjustFreq = 2'b00; // Slow the counter.
           default: AdjustFreq = 2'b10; // No change.
           endcase
           end
    2'b11: begin
           case (PLLClockN)
             2'b00: AdjustFreq = 2'b11; // Speed up the counter.
             2'b01: AdjustFreq = 2'b11; // Speed up the counter.
             2'b10: AdjustFreq = 2'b11; // Speed up the counter.
             2'b11: AdjustFreq = 2'b10; // No change.
           default: AdjustFreq = 2'b10; // No change.
           endcase
           end
    default: AdjustFreq = 2'b10; // No change; allows initialization.
    endcase
    end
```

In your `Lab22_Ans` directory, there is a subdirectory named `PLL_1x_Demo` containing the synthesizable version of the optimized 400 MHz, 1 × PLL just described. This model does not, however do any decision averaging, and so it can not resolve delay times below that of one of its oscillator delay cells, (about 80 ps). This means that it never can lock in to a drifting clock, but it can come close.

Some 400 MHz waveform results are given in Fig. 18.8–18.11.

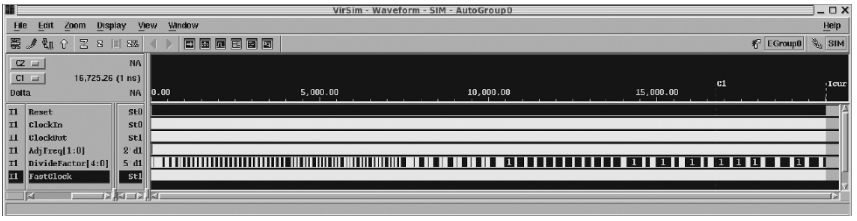


Fig. 18.8 The source PLL 1× verilog model: Entire 20 us simulation

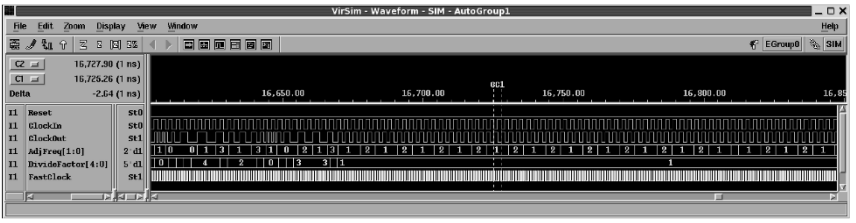


Fig. 18.9 The source PLL 1× verilog model: Lock-in detail near end of 20 us simulation

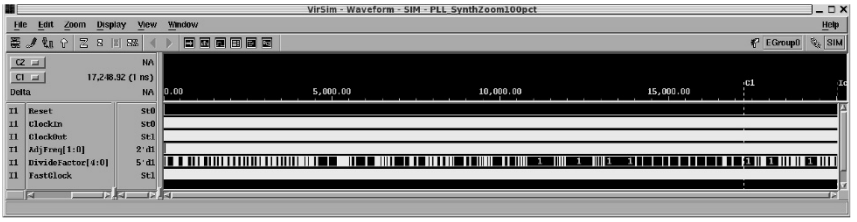


Fig. 18.10 The synthesized PLL 1× verilog netlist: Entire 20 us simulation

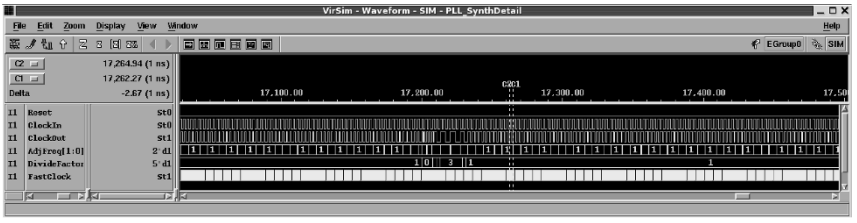


Fig. 18.11 The synthesized PLL 1× verilog netlist: Lock-in detail near end of 20 us simulation

18.2.5 Wrapper Modules for Portability

As a small digression, before starting this lab, it should be mentioned that module I/O names will be very important to keep straight for the rest of our project. In your work as a designer, it may be necessary to adapt I/O names arbitrarily. If it is

complicated or inconvenient to rename your own I/Os to match the required ones, a simple workaround is to instantiate your module in a 'wrapper' module with the correct I/O names. Keep the file name the same, but add the wrapper module declaration above the functional one. Just connect all ports together, and the wrapper names will be the only ones visible in the rest of the design.

For example, suppose you have implemented and tested a module declared this way:

```
module MyModule(output[31:0] OutBus, ..., input ClockIn);
```

But, suppose the project required that this output port be named "`DataBus`" and the clock be named "`Clock`". Just rename your module slightly and put a wrapper in your `MyModule.v` file. This is shown next.

```
// ---------------------------------------------------
// This wrapper renames the MyModule ports as required:
module MyModule (output[31:0] DataBus
                , ...
                , input Clock
                );
  MyModule_ WrapperU1 ( .OutBus(DataBus), ... (1-to-1 wiring) ...
                      , .ClockIn(Clock) );
endmodule
//
// -----------------------------------
// Begin original MyModule design (notice the underscore in the name):
//
module MyModule_ (output[31:0] OutBus, ..., input ClockIn);
 ... (valuable, tested functionality) ...
endmodule
```

In a large design, a wrapper is one of the few exceptions to the rule of declaring no more than one module in any one verilog source file.

18.3 Sequential Deserializer I Lab 22

Work in the `Lab22` directory.

Lab Procedure

Step 1. Reorganize the `Lab21` version of the PLL.

Your `Lab22` directory contains an answer subdirectory, and a symlink to a file containing verilog models for the technology library we are using in this course.

Change to your `Lab22` directory; from the old `Lab21/Lab21_Ans` directory provided, copy the `PLLsync` subdirectory and all its contents (`cp -pr`) to your new `Lab22` directory. The instructions in the present lab are somewhat complicated, so it will be best to use the answers provided rather than your own previous work.

The Lab21 file organization was as shown in Fig. 18.12.

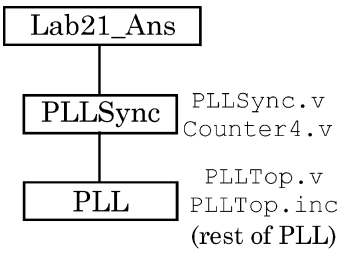


Fig. 18.12 Old Lab21 PLL file layout

In the copied files, move everything in the new Lab22/PLLsync subdirectory up one level, into Lab22. In Lab22, delete Counter4.v, which won't be used any more, and remove the now-empty PLLsync subdirectory.

Rename PLLsync.v to PLLTopTst.v; rename PLLsync.vcs to PLLTopTst.vcs; also, move PLLTop.inc up to Lab22.

The Lab22 reorganized files should be as shown in Fig. 18.13.

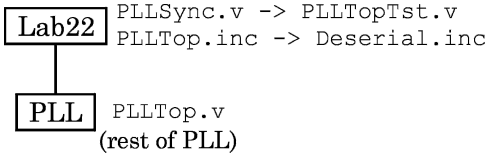


Fig. 18.13 Lab22 PLL reorganized layout

Looking ahead, the design files in our Lab22/PLL directory will become those in the PLL subdirectory of our deserializer design; so, "Lab22" mostly will become "Deserializer". For this reason, rename PLLTop.inc to Deserial.inc, and change the reference in PLL/VFO.v from PLLTop.inc to Deserial.inc. For simulation in VCS, don't put a path in VFO.v, because we plan to run VCS and DC from the Lab22 directory, so Deserial.inc will be in current context whenever we compile VFO.v.

In Lab22, edit PLLTopTst.v as follows:

- Delete all "Step*" defines, and change the `include to,

```
`include "Deserial.inc"
```

- Delete the old PLLsync module, but keep its Sample-pulse generating always block, which should be moved into the testbench module.
- Rename the testbench module to PLLTopTst.
- The device under test in the testbench now should be changed to an instance of PLLTop, and it should have an output pin named .ClockOut; the 1-bit wire mapped to .ClockOut should be named PLLClockWatch.

- The `PLLTop` instance should have a reset pin named `.Reset` and a `.Sample` pin driven by the `Sample`-pulse generating `always` block now located in the testbench.

Edit the `PLLTopTst.vcs` file so that it contains just the file name `PLLTopTst.v` and the (path)names of the verilog files in the `PLL` subdirectory.

You now should have a complete copy of the PLL design in your `Lab22/PLL` directory, with a PLL testbench in a separate file in `Lab22`. The includes for the PLL should be located in `Lab22/Deserial.inc`.

Verify briefly that you can simulate the entire PLL by invoking and running the simulator of your choice in the `Lab22` directory.

Step 2. Verify that the PLL is unsynthesizable to a correct netlist.

Change to your `PLL` subdirectory. Copy one of your `.sct` files from an earlier lab, say `Lab15`, into the `Lab22/PLL` directory. Modify the copy so that you can use it to synthesize the PLL. The PLL design at this point should consist of `PLLTop.v`, `MultiCounter.v`, `ClockComparator.v`, and `VFO.v`. Rename your `.sct` file to `PLLTop.sct` and edit it so it saves its synthesized verilog netlist into a file named `PLLTopNetlist.v`. Synthesize. The synthesis run should succeed, but, of course, the netlist can not be simulated correctly.

Change back to your `Lab22` directory to verify that the netlist does not simulate correctly. To do this, make a new simulator file named `PLLTopNetlist.vcs` in the `Lab22` directory with these contents:

```
PLLTopTst.v
PLL/PLLTopNetlist.v
-v  verilog_library_2001.v
```

in which *verilog_library*`_2001.v` is a symbolic link to the library vendor's verilog models (you may have to copy it from the CD-ROM `misc` directory); it has been modified for these labs to have nominally realistic delays. The `-v` option causes the file to be processed as a library to resolve design netlist references; this means that the simulator will compile only those library modules used in the design. VCS will be able to compile the resulting netlist; but, as we know, the synthesized VFO netlist can't adapt, because it will include a useless VFO more or less as in Fig. 18.14.

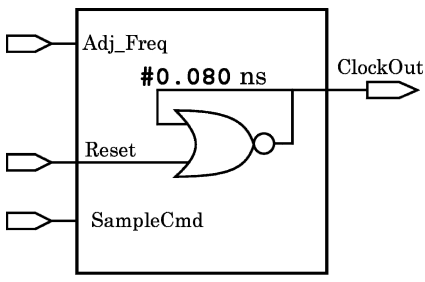


Fig. 18.14 Incorrect synthesis of the copied VFO with verilog delayed blocking assignment. The incomplete sensitivity lists caused `Adj_Freq` and `SampleCmd` simply to remain unconnected

Step 3. Rewrite the PLL so it becomes synthesizable.

Use the previous discussion in this chapter as a guide. You should rewrite substantially the `ClockComparator`, as described, and the `VFO`.

The `Sample` input port to the `PLL` should be removed, and a `Sample` pulse should be derived in `PLLTop` by passing `ClockIn` through a manually-instantiated delay cell and thence directly to the `VFO`. You should choose a verilog library delay cell simulation model in the verilog technology library file. Use an embedded synthesis script with don't-touch directives to preserve the delay cell instance(s), and their connections, from optimization.

The `AdjustFreq` code from `ClockComparator` to `VFO` should be modified so that `2'b00` requests a slowing down, `2'b11` a speedup, and other values no change; this will add some inertia to the new `VFO`'s frequency adjustments.

Some things to be alert for are mixed edge and change sensitivity lists, and task calls which have to be implemented with change sensitivity, only. The synthesizer will report this kind of violation very explicitly.

If you find this exercise to be consuming more than an hour or two, consider just copying the files in the `PLL` subdirectory from `Lab22_Ans` to your new `Lab22/PLL` subdirectory.

Your manually instantiated delay cell will have a delay of less than 100 ps; it would be advisable to set `` `timescale `` in your `.inc` file to `1ns/1ps`, for accurate simulation timing. The higher time resolution will slow the simulation somewhat; however, resolution of the exact value of the delay-cell delay is required for good verification.

Step 4. Verify that the PLL now synthesizes correctly.

To do this, just synthesize `PLLTop` and simulate the resultant netlist by invoking VCS in `Lab22` again. If you have succeeded (see Fig. 18.15), you should find that the `PLL` netlist will generate a clock output about as good as did the verilog source design.

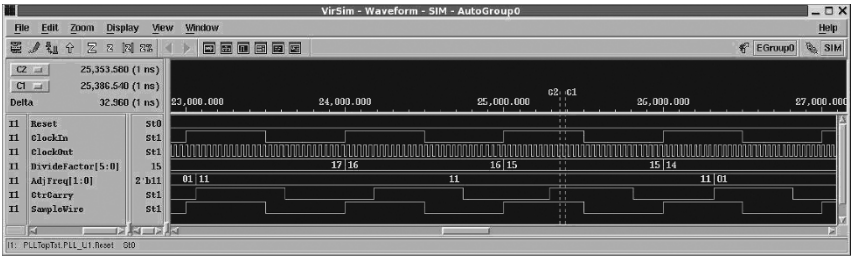


Fig. 18.15 Simulation of synthesized PLL netlist

Step 5. Organization of the `Deserializer` block. Our goal in the next few Steps will be to implement a first-cut `Deserializer` as shown in Fig. 18.16.

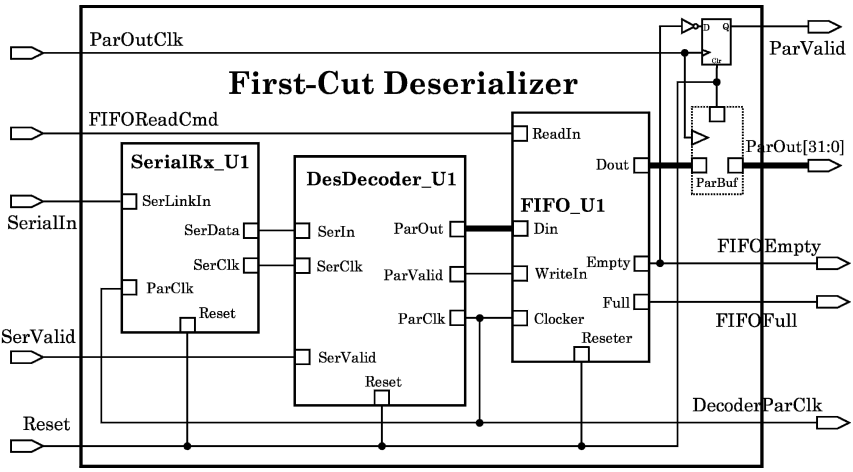


Fig. 18.16 Schematic of our first complete `Deserializer`

Change to the `Lab22` directory. The `PLLTop`* files there are no longer useful and should be moved to the `PLL` subdirectory or deleted. In the `Lab22` directory, create a new module named `Deserializer` in its own file. Prepare an empty testbench, `DeserializerTst`, in a separate file.

In the `Deserializer` module, create empty-port instantiations of the FIFO (`FIFO_Top`), the `DesDecoder` (of `Lab16`), and a new serial receiver module named `SerialRx`.

The resulting file organization should be as shown in Fig. 18.17.

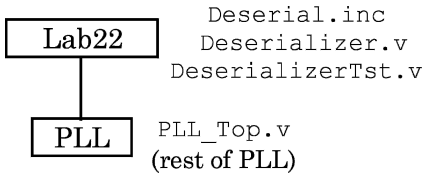


Fig. 18.17 File reorganization for the `Deserializer` implementation

From the data flow Fig. 18.1 and the schematic Fig. 18.16, `Deserializer` should have one input named `SerialIn`, another input named `ParOutClk`, and a 32-bit output bus from the decoder to the `FIFO_Top` named `ParOut`. Create these ports.

Create an output port for the PLL's decoded incoming clock, which originates at the `DesDecoder` instance's `.ParClk` pin and goes to the `FIFO_Top` instance's `Clocker` pin; to avoid confusion of this sending-domain 1 MHz clock with the input approximately 1 MHz parallel clock from the receiving clock domain, name the `Deserializer` connecting wire between these pins, `DecoderParClk` ("decoder's parallel-bus clock").

In view of the FIFO, we should require two more `Deserializer` outputs, `FIFOFull` and `FIFOEmpty`; these will not be useful to us in this lab, but they would be important in any system of which our serdes was part. Add these outputs.

We should add an external `SerValid` input in `Deserializer`, routed to the `DesDecoder` module, for test purposes. This would flag valid serial data from the sender. We also should have some sort of checking to generate a `ParValid` output from `Deserializer`. Finally, a `Reset` input should be provided to `Deserializer` so that the testbench can reset all the submodules.

Prepare to set the depth of the FIFO in `Deserializer` by declaring a `Deserializer` header parameter, `AWid` (= "Address Width" = width of register-file address bus). Assign a default value of 5 (for $2^5 = 32$ bits).

Set the width of the `Deserializer` output bus by another new header parameter, `DWid` ("Data Width"), with default value 32. This parameter then should be used everywhere in the design where the width of the parallel output bus appears. There is no special reason to have a 32-bit output bus as opposed to a 16-bit or 64-bit one, so, although we shall not alter the width in this course, our design will be parameterized to allow `DWid` to be changed to any power of 2. We could, in principle, use a different value for our future `Serializer` input width, making the two ends of our serial lane independent in width. A very, very nice feature of serial connection ….

The `DeserializerTst.v` testbench file should include `Deserial.inc` for global timing definitions. Add an include directive for this to the testbench file.

Step 6. The FIFO. Create a subdirectory `FIFO` in the `Lab22` directory, and copy in the complete FIFO design from `Lab11/Lab11_Ans`; this should consist of `FIFO_Top.v`, `FIFOStateM.v`, and `Mem1kx32.v`. The `Mem1kx32` should be the one with separate input and output ports. Remove the timescale definitions from all these files.

In `FIFO_Top`, add `AWid` and `DWid` parameters as in `Deserializer` in the previous Step. The `DWid` parameter may be used anywhere in the FIFO design to determine the width of a register or of a parallel bus. Declare these parameters and replace every width and depth in the FIFO and its submodules (`FIFOStateM` and `Mem1kx32`) so that these parameters control the values.

Add 1-bit `Full` and `Empty` output ports to `FIFO_Top`, too; these will be wired to the corresponding new `Deserializer` ports of Step 5, as soon as the FIFO can be instantiated. Also, to simplify the `FIFO_Top` wiring, remove the old `E_FIFO` and `F_FIFO` nets; instead, substitute the new output port names (`Full` and `Empty`) into their continuous assignment expressions and the instance port maps. Then, delete the three nets, `ReadWire`, `WriteWire`, and `ResetFIFO` and replace them with the net names already used in `FIFO_Top` to assign to them. If your `Mem1kx32` instance is clocked with an inversion operator ("…, `.ClockIn(~Clocker),`…"), remove the inversion operator ('~' or '`!`'), so that the memory responds to a rising-edge clock.

The FIFO_Top.v file includes a testbench. Parameterize this testbench with AWid and DWid as above. Be sure that the FIFO_Top instantiation has .Full and .Empty output pins, properly mapped in the FIFO testbench module. The testbench should have a `include of Deserial.inc. After these edits, separate the testbench into its own file, FIFO_TopTst.v, for possible future debugging.

You may have other testbenches commented out in Mem1kx32.v and in FIFOStateM.v; these testbenches should be deleted, because, if necessary, the FIFO_TopTst testbench may be used to exercise these submodules. If you find a bug in one of the FIFO submodules which requires a regression to a single submodule, you can recopy a local testbench from the earlier labs.

In Mem1kx32, add a Reset port and code which initializes all memory locations to 0. This will suppress startup parity errors when we use the memory later in simulation.

Finally, simulate with FIFO_TopTst briefly to verify that FIFO is complete and that the include file is referenced correctly.

Step 7. The Deserializer file structure. Create a subdirectory DesDecoder in the Lab22 directory for the deserialization decoder, and another subdirectory, SerialRx, for the serial receiver.

Copy the DesDecoder.v from Lab16_Ans into the DesDecoder subdirectory. Remove the `timescale from DesDecoder.v, and pass the module just one parameter, DWid. Change the name of the output bus from ParBus to ParOut, rename the ParRst port to Reset, and delete the testbench, if it still is present. If they are present in your copy, remove the unused SHIFT and RESET localparams.

Then, create a new file, SerialRx.v, in a new Lab22 subdirectory named SerialRx. This file should contain just the following:

```
module SerialRx(output SerClk, SerData
                , input SerLinkIn, ParClk, Reset
                );
assign SerData = SerLinkIn;
//
PLLTop PLL_RxU1 ( .ClockOut(SerClk), .ClockIn(ParClk), .Reset(Reset) );
//
endmodule // SerialRx.
```

This finally shows where we shall instantiate our PLL: In the SerialRx module.

After all these changes, the resulting directory structure should put the DesDecoder, FIFO, PLL, and SerialRx blocks all directly under Lab22. This is shown in Fig. 18.18.

The Lab22 top level will become the Deserializer directory of our next lab exercise.

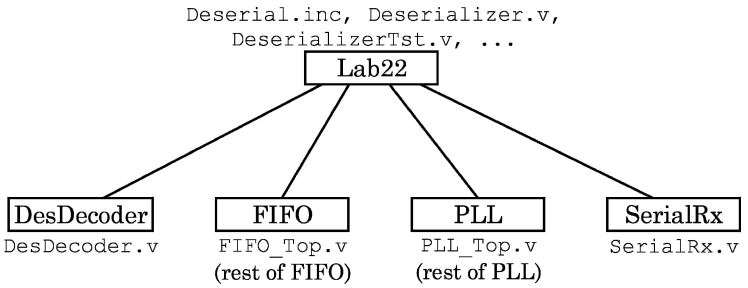


Fig. 18.18 New `Lab22` directory. Files are named near their directory (block)

In the `Deserializer` module, instantiate the `FIFO_Top` module, and connect it with `DesDecoder` and `SerialRx` instances as well as possible for now, according to the design dataflow Fig. 18.1. Wire the `SerialRx` with port names as shown previously.

Because the width and name context of the `DesDecoder` and `SerialRx` instances strongly constrains their ports, create module templates for submodules of `Deserializer.v` first by wiring up the instance port maps, postponing module declarations. This way, you will have all connections immediately visible for your wiring. The result will look something like this (before wiring):

```
module Deserializer
...
// ------------------------------------------------------
// Structure:
//
FIFO_Top #( .AWid(AWid), .DWid(DWid) )
FIFO_U1
  ( .Dout(), .Din()
  , .ReadIn(), .WriteIn()
  , .Full(), .Empty()
  , .Clocker(), .Reseter(Reset)
  );
//
DesDecoder #( .DWid(DWid) )
DesDecoder_U1
  ( .ParOut(), .ParValid()
  , .ParClk(), .SerClk()
  , .SerIn(), .SerValid(), .Reset(Reset)
  );
//
SerialRx
SerialRx_U1
  ( .SerClk(), .SerData()
  , .SerLinkIn(), .ParClk()
  , .Reset(Reset)
  );
//
endmodule // Deserializer
```

Then, compose the module wiring declarations by including them (illegally) in the instance port maps, using ANSI format. For example, in the code above, you might next change the above `FIFO_Top .Din()` to `output[Dwid-1:0] .Din(DecodeToFIFO)`, and the `DesDecoder .ParOut()` to `output [DWid-1:0] .ParOut(DecodeToFIFO)`.

Do this any way you want, but developing some sort of systematic way to define module ports and declarations top-down is an important skill in VLSI design.

In other words, start by declaring modules where instances will be wired. For example, to instantiate the FIFO in `Deserializer.v`, start by cut-and-pasting this into your `Deserializer.v` file:

```
module FIFO_Top #(parameter AWid = 5  //  FIFO depth = 2**AWid.
                  ,          DWid = 32 //  Default width.
                  )
         ( output[DWid-1:0] Dout(wire[DWid-1:0] FIFO_Out)
         , input[DWid-1:0] Din(wire[DWid-1:0] DecodeToFIFO)
         , output Full(wire FIFOFull), Empty(wire FIFOEmpty)
           ...
         );
```

The port map wiring shown is illegal, because it includes width declarations copied directly from the ports. However, this is just an intermediate step.

Next, copy these combined instance-wirings plus declarations into their separate, proper `.v` files in the `Lab22` subdirectories. After copying, in the submodules the instance wirings may be removed and the ports renamed or resized as necessary.

Finally, alter the code in the `Deserializer` module to change the module port declarations to simple instance pin-outs, leaving this in your `Deserializer.v` file:

```
FIFO_Top #( .AWid(AWid), .DWid(DWid) )
FIFO_Top_U1                       // Instance name.
  ( .Dout(FIFO_Out)               // pin-out (= port map).
  , .Din(DecodeToFIFO)
  , .Full(FIFOFull), .Empty(F_IFOEmpty)
       ...
  );
```

Returning to the submodules, add reset inputs to all submodule declarations, as well as any useful read, write, full or empty communication with the FIFO instance. Err on the side of adding ports; you always can remove unnecessary ports later.

Be sure to use parameters from the `Deserializer` declaration in all instance width declarations; pass the main submodules only `DWid` and/or the FIFO depth.

Finally, with only the FIFO and deserialization decoder actually defined, create a `DeserializerTst.vcs` file and load your mostly-unimplemented

`Deserializer.v` structural design into the simulator, just to compile and check connectivity. Correct connection problems before continuing. When in doubt, add another port.

Step 8. The parallel output buffer (`ParBuf`) in `Deserializer`. For now, we shall implement the output register in Fig. 18.16 simply by declaring a `reg` named `ParBuf` in the top-level module, `Deserializer`.

Although our PLL generates a 1 MHz parallel-data clock from the serial stream, our packet format requires 64 bits per 32-bit word, so our serial line can deliver parallel-data words only at about 500 kb/s.

So, we shall assume an external 1/2 MHz clock `ParOutClk` to clock data into this buffer this way,

```
always@(posedge ParOutClk, posedge Reset)
  begin : OutputBuffer
  if (Reset==1'b1)
       ParBuf <= 'b0; // To be wired to the ParOut port.
  else ParBuf <= FIFO_Out;
  end
```

The `ParOutClk` may be generated in the testbench module. This code says that when the FIFO is reset, only zeroes can be read from it. We also should provide a `ParValid` flag; this can be set in the same always block with `ParBuf`.

Step 9. The deserialization decoder. Change to the `DesDecoder` subdirectory. The contents of your `DesDecoder` module should be copied from `Lab16/Lab16_Ans`, with the module I/O changes already made above (in this lab).

Make a second, new copy of the `Lab16_Ans/DesDecoder.v` file, renaming it to `DesDecoderTst.v`, and remove the design module; keep the testbench at the bottom of the file. In the new testbench file, change the timescale to `1ns/1ps`, remove the `DC` compiler directives, and change the parameter references to `DWid`.

Also, in the new `DesDecoderTst.v` file, if the serial data to be shifted to the `DesDecoder` instance is shifted LSB first, reverse the order, for consistency, so that the data MSB goes out first. Remove all delay expressions from the testbench, except those in the main `initial` block and in the serial clock generator.

Now, modify `DesDecoder.v` as follows:

- In case we might want a command from here to the FIFO, add a 1-bit output port named `WriteFIFO`.
- Remove all #*delay* expressions and rephrase or remove all comments concerning these delays.
- To put this module better under procedural control, change the runtime so that instead of four concurrent `always` blocks, there are just two: The `ClkGen` block, which should be sensitive to change in `SerClk` or `Reset`, and a new, second `always` block as follows:

```
always@(negedge SerClock, posedge Reset)
 begin
 Shift1;
 Decode4;
 Unload32;
 end
```

The sensitivity to the negative edge of `SerClock` is to avoid a race with the serial data shift.

- The `Unload32` task, as written for `Lab16`, includes an unnecessary edge-sensitive event control; replace this with a simple `if` test for `ParClk==1'b0`.
- The `ClkGen` task should be cleaned up a little more: Instead of having an `if` in the non`Reset` condition controlling nothing but serial clock gating, put everything, including the other `if`, under control of `if (SerValid==YES)`.

After these changes, make up a `DesDecoderTst.vcs` file and simulate `DesDecoder` alone, just to be sure the new `DesDecoder` testbench connections are correct. Because of the complexity of this module, some further debugging may be useful at the unit level; but, this module should be almost usable without changes other than ones to permit synthesis, which will be made later in the lab.

Step 10. The `DeserializerTst` testbench. This may be based on the one from the `DesDecoder` of `Lab16`, although, if so, it will have to be modified to make the complete `Deserializer` work. The main problem is that there can be no serial clock in the `Deserializer`, whereas there was a testbench serial clock in the `DesDecoder` of `Lab16`. In the present lab, the `Deserializer` has to extract the parallel clock from the serial input stream, and then use the extracted clock to generate a serial clock synchronized with that same input serial stream.

So, we shall have to go easy on the `Deserializer`, at least at first. We shall use the testbench to feed `Deserializer` a serial stream at a clock rate only very slightly different from the base rate of the PLL. This will make extraction of the clock easy enough for us to validate the rest of the design. Later, we can see how far we can go in providing a serial stream farther off the deserializer's PLL base clock frequency.

With all this, the testbench ***should*** create a padded serial packet just as it did in `Lab16`; however, let us set the testbench serial clock half-period at 15.6 ns, which is only slightly off the `PLL` base rate of 500/32 ns = 15.625 ns = 15 ns after verilog truncation in integer division. The farther away from 15 ns we go, the more serial data the `Deserializer` will have to drop because of loss of synchronization, at least at this stage in the design.

Use the testbench to generate a separate 1/2 MHz clock to attempt reads from the `Deserializer` FIFO output register. Don't worry now about synchronization of `Deserializer` output reads with valid data in the `ParBuf` register. Then, instantiate `Deserializer` in this testbench and simulate it. It should be possible,

as shown in Fig. 18.19 and 18.20, to obtain some good data for the FIFO, although there will be many losses of synchronization.

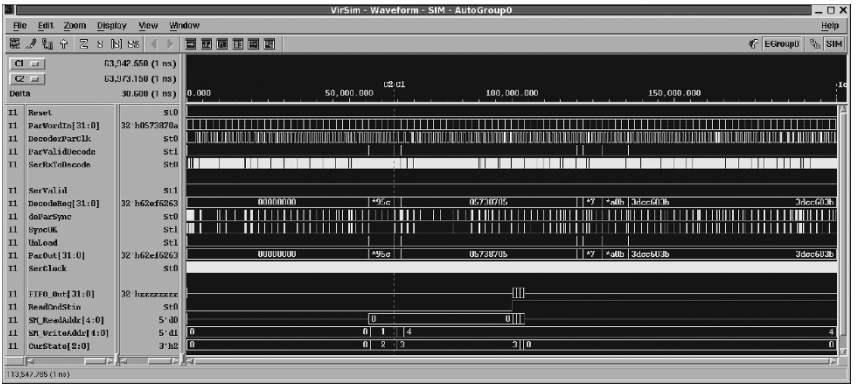


Fig. 18.19 Overview of source simulation of first working `Deserializer`

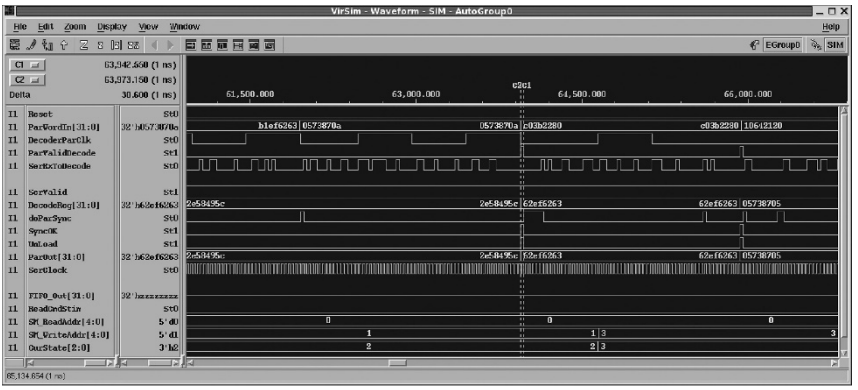


Fig. 18.20 Close-up of a parallel-bus unload of the first working `Deserializer`

Step 11. Tying up loose ends. When you have the `Deserializer` source verilog design more or less working (it should be generating a parallel clock and at least occasionally correctly parallelizing the serial data), refine the design as follows:

A. Remove the `DesDecoder`'s `WriteFIFO` output; this was completely superfluous. The serial-parallel input functionality will decide when to write to the FIFO, and it will be up to the external system to stop sending serial data if the FIFO should become full. Instead, in `Deserializer`, connect the `DesDecoder` instance's `ParValid` output to the `FIFO_Top` instance's `WriteIn` port.

B. Simulate. Your FIFO has no read requested, so it should fill up. If you used your own implementation, you may have to debug the FIFO to do this.

Run the simulation for enough time to fill the FIFO, and then to empty it at least once. To empty it, you will have to issue a read command in the testbench after the FIFO is full: Assert `FIFOReadCmd` to do this. See Fig. 18.21.

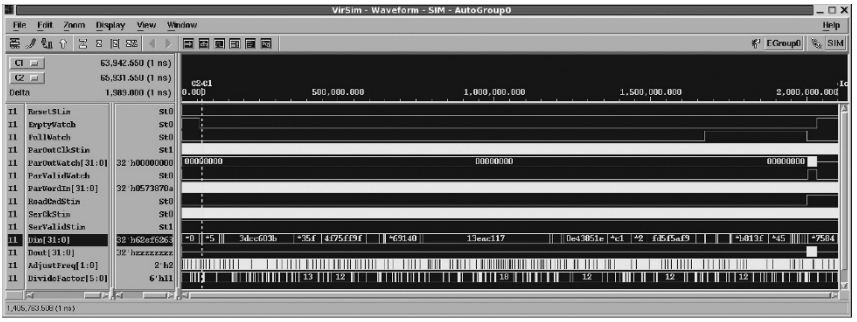


Fig. 18.21 Overview source simulation of a somewhat improved `Deserializer`

C. Check the simulated transferred parallel data against the `Deserializer` testbench input for correctness, allowing for dropped serial words, if any. One implementation yielded the result in Fig. 18.22.

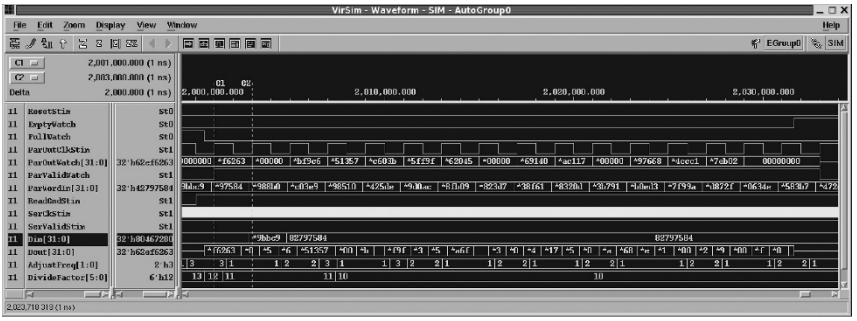


Fig. 18.22 The first word out of this improved `Deserializer` was `32'h62ef_6263` on `Din` at $t = 63{,}942{,}550$ ns; however, it was copied to the `ParWordIn` bus, and stored in the FIFO, as `32'hb1ef_6263`. Clearly, there is more work to be done here

Do not worry now about corner cases, but your FIFO should store data so that the value written to at least two different given addresses is the value later read out. Recalling Step 8 above, typical code in `Deserializer` to read into the parallel output buffer would be,

```
always@(posedge ParOutClk) // 1/2 MHz clock domain of the receiving system.
  if (Reset==1'b1  ||  F_Empty==1'b1)
        begin
        ParBuf      <= 'b0;   // Zero the output buffer.
        ParValidReg <= 1'b0;  // Flag its contents as invalid.
        end
  else begin // Copy data from the FIFO:
        ParBuf      <= (FIFOReadCmd==1'b1)? FIFO_Out : 'b0;
        // Flag the buffer value validity:
        ParValidReg <= FIFOReadCmd && (~FIFOEmpty);
        end
```

Two possible problems we shall address later: (a) Were FIFO addresses `0x00` and `0x1f` written and read? (b) Can the FIFO be read from and written to at the same time, so that the FIFO doesn't quickly fill up or go empty?

Step 12. Synthesis. Attempt to synthesize the entire `Deserializer`. At this point, the goal is just to get some kind of netlist written out. You may have to modify certain parts of the design or conceal them from the synthesizer in `` `ifdef DC `` blocks. The `DesDecoder` may require several changes.

Recall that the `PLL` now is synthesizable, but that the FIFO is not. The `FIFO` part of the netlist will be full of missing connections and dangling output pins.

To reduce the time, use a synthesis script which includes no design rule or speed constraint; just optimize for minimum area.

The resulting hierarchical netlist should total over 35,000 transistor-equivalents and may seem to be intact; however, it will not be functional, so don't bother trying to simulate it.

After this, use the same synthesis script again; but, just before the command to run the compile, insert a command to flatten the design. The synthesizer now will be able to remove unconnected logic across hierarchy boundaries, reducing the netlist size further.

So, we have a reasonable idea of the gate-size of our `Deserializer`, and we can see the extent to which flattening might reduce this size.

18.3.1 Lab Postmortem

How can one constrain synthesis for minimum area vs. minimum delay?

What is the difference between synthesis constraints and design rules?

Think about local delay or area minima and the multidimensional nature of the constraints.

What is incremental optimization?

18.3.2 Additional Study

(Optional) Review parameter and instantiation features in Thomas and Moorby (2002) sections 5.1 and 5.2 (ignore `defparam`).

Chapter 19
Week 10 Class 1

19.1 The Concurrent Deserializer

In this chapter, we'll improve our deserializer implementation and shall assemble a complete serial receiver with imperfect (incomplete) functionality.

The main change will be modification of the FIFO to permit concurrent (dual-port) read and write. Fig. 19.1 gives our working-sketch schematic of the top level.

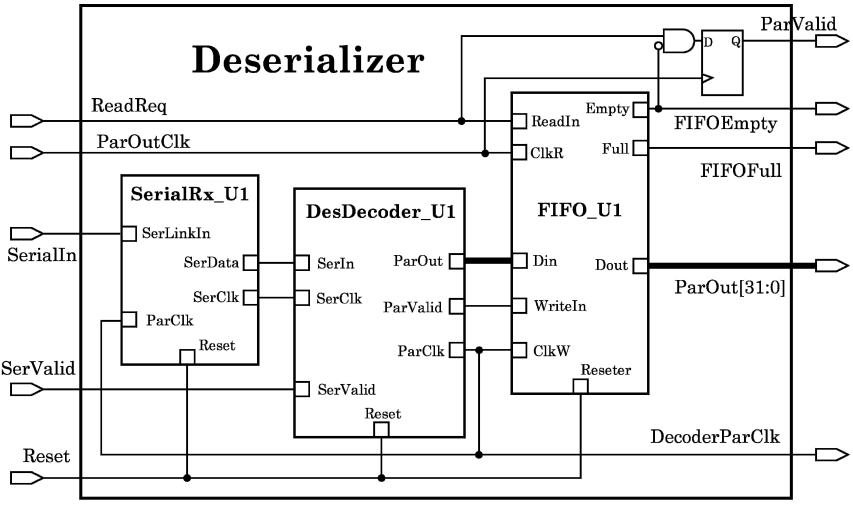


Fig. 19.1 Schematic of the concurrent Deserializer (with FIFO in two clock domains)

To implement this part of our design, we shall modify our RAM to be dual-ported; also, we shall change the FIFO state machine (FIFO controller) so it finds the difference between number of reads and writes on the current clock cycle and changes state according to the preponderance of memory usage on that clock. We read or write on one edge; we determine addresses and state transitions on the other. We'll also fix various things to make the `Deserializer` synthesizable – although perhaps not entirely functional.

19.1.1 Dual-porting the Memory

This just requires enabling the memory to allow simultaneous (= same clock) read and write; of course, these will be to different addresses, as determined by the FIFO controller, not the memory. The FIFO may be in two different clock domains. We shall call the new memory, "`DPMem1kx32`".

To upgrade our memory, we shall do this:

- Wire our chip enable (`Mem_Enable`) in `FIFO_Top` to a constant '`1`' so that the register file always is enabled. This is because of our use of the memory, not because of the dual-port functionality.
- Install separate read & write address ports for the memory.
- Supply separate read & write clock ports for the memory.
- Make the memory read and write procedures independent.

19.1.2 Dual-clocking the FIFO State Machine

This is more complicated than dual-porting the memory; it may be summarized this way:

- Supply independent read & write clocks (shared with the FIFO RAM).
- Permit nonexclusive read and write commands to RAM.
- Derive a single state-transition FIFO controller clock from the input read and write clocks. This is because the FIFO read and write ports in general will be in different clock domains. Furthermore, it implies that an edge in either domain might be effective as a state-transition clock.
- Safeguard against more than one read address or more than one write address per state-clock in a command to the RAM.
- Determine state transitions on each clock solely from the resultant effect of all read and write requests made during that clock cycle.

The rest is explained in the lab instructions.

19.1.3 Upgrading the FIFO for Synthesis

The primary problems with synthesis of the FIFO as modified above for dual-port operation will be:

- The design still may include delay expressions; these are useless for synthesis.

  $\Rightarrow$ We shall remove all delay expressions.

- The `incrRead` and `incrWrite` tasks include edge-sensitive event controls but are called within a change-sensitive combinational block.

⇒ We shall modify these tasks in two steps, as we do the lab: (*a*) First, we shall prevent these tasks from running more than once per state-machine clock cycle, but we shall leave the change *vs*. edge event control problem alone; (*b*) then, we shall remove the task event controls and break up the task functionality to put some of it in an external `always` block. The final result will synthesize correctly.

- The state machine clock generator includes an `always` block with a change-sensitive sensitivity list containing variables not used in the block; this causes dangling inputs to the synthesized (useless) clock-generator logic. The usual synthesizer latch-inference problem.

  ⇒ We shall derive the state machine clock directly from the mutually-independent FIFO read and write clocks.

- The counters are sequential elements in addition to the state register. Thus, the logic in this particular machine never can be divided neatly into sequential logic for the state register and combinational logic everywhere else. The combinational block also causes incorrect synthesis of latches.

  ⇒ We shall replace the state-machine combinational block with a sequential block clocked on the opposite edge from the state-transition block.

19.1.4 Upgrading the Deserialization Decoder for Synthesis

- We shall simplify the `Shift1 always` block by removing its temporary registers.
- We shall rewrite the `ClkGen` task as a rising-edge sensitive `always` block; this will prevent latch inference. Also, we shall remove the `SerClock` gate from `ClkGen` and put it in a combinational block.
- We shall replace all clocked blocking assignments, which are relicts of the old `DesDecoder` tasks, with nonblocking assignments.
- We shall prolong the assertion of `ParValid` by adding a small counter (`ParValidTimer`) to the `Unload32 always` block. The duration of `ParValid` will be guaranteed to be at least 8 serial-clock cycles.

19.2 Concurrent Deserializer II Lab 23

Work in the `Lab23` directory.

Lab Procedure

Step 1. Duplicating the `Lab22 Deserializer`. Start by creating a new subdirectory, `Deserializer`, in the `Lab23` directory. Copy the entire `Lab22/Lab22_Ans/Lab22_Ans_Step12` contents into the new `Deserializer` subdirectory.

After the copy, delete the copied synthesized netlists and netlist log files.

This `Deserializer` should include a synthesizable PLL, a single-port RAM which prevents FIFO simultaneous read and write, and a FIFO which, as it happens, will not synthesize correctly. You may wish to recreate the symlink to the verilog simulation-model library (*LibraryName*`_v2001.v`) in the `Deserializer` directory, if it is not already valid. In Windows, a copy should be made.

Briefly simulate the copied `Deserializer`, using the copied `Lab22` testbench, to ensure a correct and complete copy. The PLL synchronization will not be especially good, but the system should work.

Step 2. Deserializer corner cases and FIFO debugging. Change to the new `Deserializer/FIFO` subdirectory. You should have there the FIFO design, a FIFO testbench, and a simulation `.vcs` file.

Run the FIFO simulation, modifying it if necessary, so that long intervals of writes alternate with long intervals of reads. Verify that corner-case FIFO memory addresses `0x00` and `0x1f` both were read and written; don't bother about the data values, if any. Do not worry now about transition addresses `0x01` or `0x1e`. If the `0x00` and `0x1f` addresses were not accessed both for read and write, fix the `FIFOStateM` module so that they are. Resimulate the FIFO again briefly, modifying, if necessary, the testbench so that both read and write are requested at the same time. Then, exit the simulator.

We shall be changing the FIFO design in this lab. To prepare for a thorough redesign, start by removing all #*delay* expressions from all FIFO design modules. Also, to distinguish the new FIFO from the old, rename all the files to remove the underscores, and change the module names accordingly. The result in the `Deserializer/FIFO` directory should be,

```
default.cfg
FIFOTop.v
FIFOTopTst.v
FIFOTopTst.vcs
FIFOStateM.v
Mem1kx32.v
```

Recall that all address parameters were renamed to `AWid` as part of the `Lab22` exercise. The `default.cfg` file is a VCS side file and is not essential to the design.

Step 3. Enabling memory simultaneous read and write. In the `Deserializer` simulation, you might have seen that write requests for the incoming serial stream were ignored when the FIFO was full; write requests also were ignored when there also was a read request from the parallel-bus `Deserializer` output. If you did not notice this last, resimulate the `FIFO` briefly.

The original `Mem1kx32` which we adapted to use with our FIFO has only one address bus and thus can not execute a FIFO read and write simultaneously. The

single address made the RAM easier to implement and debug, but now it prevents good FIFO operation.

It is the FIFO control logic that first of all enforces the precedence so that a read prevents a memory write. To prepare to remove this control, in `FIFOTop.v`, delete the declaration and assignment to the `Mem_Enable` net. Wire the `ChipEna` port of the `Mem1kx32` instance in `FIFOTop` to a `1'b1`, which will enable the memory permanently. There is only one RAM in the FIFO, so we don't need a chip enable control, anyway.

Although now enabled at the chip level, the memory still can't perform useful read and write at the same time because it has only one address port, and because the memory internal logic itself makes read and write mutually exclusive by `if-else` (in my version of `Mem1kx32`). Furthermore, in `FIFOTop`, there is a continuous assignment which prioritizes read address over write address in determining the one possible memory address; this can't be removed, given our current, one-address memory.

Clearly, we have to redesign the memory.

In the following, keep in mind that the `Deserializer` will generate a FIFO write request whenever `ParValidDecode` is asserted by the `DesDecoder`.

Step 4. Dual-porting the FIFO memory. The original module ports are shown in Fig. 19.2.

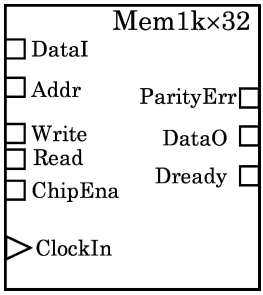


Fig. 19.2 I/O's of the original, single-port `Mem1kx32`

Start by renaming the `Mem1kx32.v` to `DPMem1kx32.v` (`DP` = "Dual-Port"). Then, change the module name to match the file name. Modify `FIFOTop.v` and the `.vcs` file accordingly.

Next, edit `DPMem1kx32` to add the second address port: Rename the existing address input port to `AddrR`, and add a second port named `AddrW`. Both should have widths determined by the `AWid` parameter value.

We will want to clock data in and out with separate clocks, so likewise rename the current, single clock to `ClkR` and add a second clock input port named `ClkW`. See the result in Fig. 19.3.

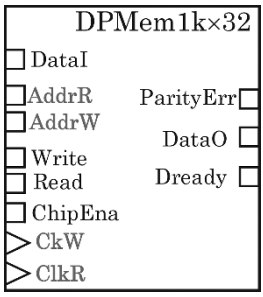


Fig. 19.3 `Mem1kx32` converted to a dual-port memory

Internally, rename the old `ChipClock` to `ClockR`, and declare a new wire named `ClockW`.

Although we have disabled `ChipEna` in the `FIFO`, we still want the control inside the memory. So, put both clocks under control of the `ChipEna` input by wiring them to internal nets and muxing them to `1'b0` this way:

```
assign ClockR = (ChipEna==1'b1)? ClkR : 1'b0;
assign ClockW = (ChipEna==1'b1)? ClkW : 1'b0;
```

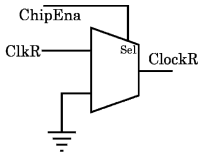


Fig. 19.4 `ClkR` clock gate logic

The logic for the clock gating by the two preceding continuous assignments is the same as in the `ClkR` schematic of Fig. 19.4. Also, keep the gating of `Dready` and `DataO` by `ChipEna`.

We still have read and write logically dependent. To remove this dependency, put the read functionality and the write functionality into two separate `always` blocks as shown below. The continuous assignment gating is enough to control all activity, so we don't need a `ChipEna` either in the read or the write block. Because memory reads will be clocked externally from the `DPMem1kx32` output latches, reconvergence from the internally gated clocks will not be a concern.

There is no special reason to initialize memory in the hardware, but to have fail-safe parity checking, we should generate an error on any bit '`x`' or '`z`'. This means initialization of all memory locations on `Reset` to prevent false parity errors during simulation. `Reset` should be applied both to the read and the write blocks, but the memory initialization should be included only with the write code. Be sure to avoid, ***always***, assigning to any variable in more than one `always` block; the risk of a race condition can not be overstated.

Thus, the separated read and write `always` blocks should be done this way:

```
...
always@(posedge ClockR, posedge Reset)
  begin : Reader
  if (Reset==1'b1)
       (init Parityr, Dreadyr, DataOr)
  else if (Read==1'b1)
            (do parity check and read)
  end // Reader.
//
always@(posedge ClockW, posedge Reset)
  begin : Writer
  if (Reset==1'b1) // Zero the memory:
       for (i=0; i<=MemHi; i=i+1) Storage[i] <= 'b0;
  else if (Write==1'b1)
             (do write)
  end // Writer.
...
```

As shown, name the newly separated blocks `Reader` and `Writer`. Because `Reader` block has to do a parity check, it should initialize the parity error output to `0`, as in the old `Mem1kx32` model. `Writer` probably should use its reset branch to do nothing but zero out the memory storage on `Reset`. The `Writer` block should not assign 'z' when it is inactive (this would be OK for a shared read-write bidirectional bus).

Finally, it's about time we adopted a new `reg` naming convention. The synthesizer always renames design `reg` types, if they are preserved, to `*_reg` in the netlist; so, using our current convention of `*Reg` just creates `*Reg_reg` names in the netlist, which is redundant. So, in the `DPMem1x32` and all other FIFO modules, change all of our declared `reg` names which end in `*Reg` to `*r`.

After these changes, modify `FIFOTop` to remove the `SM_MemAddr` declaration and its assignment statement. Connect each of the two new `DPMem1x32` address input pins directly to its proper state machine output address pin. Also, connect the one available clock to both of the `DPMem1kx32` clock input pins. Then, verify that the FIFO simulates the same way with the new `DPMem1kx32` instance as it did with the original `Mem1kx32`.

After this, we can do no more in `FIFOTop` or `DPMem1kx32`. We have to modify the state machine design, which itself includes code assuming single-port memory operation.

Step 5. Modifying the FIFO state machine I. This is going to be more complicated than the preceding, so we shall do it in two stages.

First, change to the Deserializer directory and change all references from FIFO_Top to the new FIFOTop, and all references from Mem1kx32 to DPMem1kx32. Simulate the Deserializer. You will notice that, during the memory reading, ParValid goes high, indicating a write from the incoming serial line. However, no write occurs, because the FIFO state machine has been designed to avoid "race conditions" by means of an if-else giving every read priority over write.

However, we now have separate read and write addressing of the memory, which should prevent data corruption. So, let us separate the state machine's read and write functions, allowing them to occur concurrently.

Change to the FIFO subdirectory. In FIFOStateM, rename the Clk port to ClkR and add a second clock input port ClkW. After this, in the same module, modify both of the address increment tasks with a latching semaphore which will protect the address from being changed more than once per clock cycle. For example, the read-address task should be modified as shown next:

```
reg LatchR, LatchW; // Reset these to 1'b0 in SM comb. block.
...
task incrRead; // Unsynthesizable!
  begin
  if (LatchR==1'b0)
    begin
    LatchR = 1'b1;
    @(posedge ClkR)
      ReadCount = ReadCount + 1;
    LatchR = 1'b0;
    end
  end
endtask
```

Revise both of the tasks, incrRead and incrWrite, in FIFOStateM to be like the one above.

We have a theoretical issue to address: A state machine can not be guaranteed deterministic unless it has a unique clock. But, now we have two clocks into FIFOStateM, so we must derive a single clock to determine the machine's current state. This can be done by means of a simple, change-sensitive always block:

```
always@(ClkR, ClkW, Reset)
  if (Reset==1'b1)
       StateClock = 1'b0;
  else StateClock = !StateClock;
```

This construct is not synthesizable (ClkR and ClkW go nowhere), and we shall revise it later in the lab. For now, it will do as a quick simulation check; so, declare

a reg named StateClock and add the above to FIFOStateM. Change the state transition clock name from Clk to StateClock. Also, change the state machine combinational block's enable to StateClock==1'b0. In the incrRead and incrWrite tasks, replace each posedge event control with one depending on StateClock.

Next, in the state machine combinational block, in the empty and full state code, remove all mutual dependence of read and write. Specifically, in the empty state, there is no reason for the if to test for a read request; in the full state, the if should test only for a read request.

Leave the a_empty and a_full state code alone for now.

In the normal state code, the counters are checked, so state transitions can be determined correctly on a simultaneous read and write in any order; therefore, in the normal state, just separate the read and write conditions each into its own if by removing all dependence on the other request. Be sure to remove all statements which disable the opposite operation; for example, remove "WriteCmdr = 1'b0;" from the memory read statement.

Also, extract all transition logic from the two if's, and collect it below the second if. Because the requests both may be processed now on one clock, we have to wait for the last one to lapse before deciding on the transition to the next state.

The resulting transition logic should look about like this:

```
(read & write command logic)
// Set the default:
 NextState = normal;
 //
 // Check for a_full (R == W+1):
 tmpCount = WriteCount+1;
 if (ReadCount==tmpCount)
      NextState = a_full;
 //
 // Check for a_empty (W == R+1):
 tmpCount = ReadCount+1;
 if (WriteCount==tmpCount)
      NextState = a_empty;
```

We now have independent read and write in the normal state. We are not finished yet, but you may wish to simulate briefly to check for gross errors. With the changes above, the FIFO register-file corner case addresses 0x00 and 0x1f again should be found to be read and written, as should the neighboring 0x01 and 0x1e.

Step 6. Routing the new read and write clocks. We need two clock inputs for the FIFO. In FIFOTop.v, rename the Clocker input to ClkR and add another input named ClkW. For the FIFO in the deserializer, ClkR will be for the receiving domain, and ClkW will be the parallel-data clock from the DesDecoder. In FIFOTop, connect the two clocks directly to the DPMem1kx32 and the FIFOStateM instances.

After this, in `Deserializer.v`, consistent with our FIFO naming convention, rename `ParValidReg` to `ParValidr` – in fact, now would be a good time to change *all* remaining design names from `*Reg` to `*r`.

Also, rename the `FIFOReadCmd` input port to `ReadReq`; we want to be sure to differentiate a request for a read (originating external to the FIFO) from a read command to the register file (within the FIFO). Modify the `DeserializerTst` testbench for the new port name.

In `Deserializer`, change the `FIFOTop` port map so that `.Clocker` is renamed to `.ClkW`, and a new `.ClkR` port is mapped to `ParOutClk`.

After this, simulate `Deserializer` briefly. The result should be changed, but the result still should be reasonable, if only intermittently good. The FIFO I/O ports now match those shown in the schematic of Fig. 19.1, but we haven't finished yet.

Step 7. Modifying the FIFO state machine II. We should begin by modifying the `a_empty` and `a_full` states so they will work properly for independent read and write requests. Looking at the `FIFOStateM` combinational block, the problem is that (*a*) as soon as one operation is triggered, the machine disables the opposite one; (*b*) the machine prioritizes one operation while mutually-excluding the other; and, (*c*) the machine does not check the counter values in these states and thus can not determine whether a read or write command, or both, has been issued on the current clock.

So, open `FIFOStateM.v` and, for `a_empty` and `a_full`, remove the disabling assignment to `ReadCmdr` in the writing blocks and to `WriteCmdr` in the reading blocks. The rationale here is the same as previously, for the `normal` state, but leave the transition rules alone for the moment.

For `a_empty`, disentangle the transition logic from the memory command logic and collect the transition logic together at the end of the `a_empty` block, so it can be made to depend on the counter values rather than the memory command decision(s).

To ensure correct counter wrap-around, use an `AWid`-bit wide `tempCount reg` as shown here:

```
// In this state, we know W == R+1; ...
// Memory command logic:
if (WriteReq==1'b1) ...
if (ReadReq==1'b1) ...
//
// Transition logic:
// Set default:
NextState = a_empty;
//
// Check for change:
tempCount = ReadCount+2; // Destination determines wrap-around.
if (WriteCount==tempCount)
     NextState = normal;
else if (WriteCount==ReadCount)
     NextState = empty;
```

Do similarly for the a_full logic.

At this point, let's look again at the command logic in the combinational block. Consider the normal state. In that state, the command logic should be something like this:

```
// On a write:
if (WriteReq==1'b1)
  begin
  WriteCmdr = 1'b1;
  incrWrite; // Call task, which blocks on posedge StateClock.
  end
// On a read:
if (ReadReq==1'b1)
  begin
  ReadCmdr = 1'b1;
  incrRead; // Call task, which blocks on posedge StateClock.
  end
(transition logic ...)
```

With the code shown above, a write request would call incrWrite, which would block further execution until the next positive edge of the clock. This would cause loss of a read request on the low phase of the current clock.

A typical simulation solution to prevent this loss would be to put both the read and write requests together in a fork-join block; then, a simultaneous read and write both would be processed together, and both would block on the positive edge, which, when it occurs, would unblock the command processing and allow the transition logic to update next_state and the address counters for the next positive edge. The result would be as shown next:

```
fork // unsynthesizable! But do it for now.
// On a write:
if (WriteReq==1'b1)
  begin
  WriteCmdr  = 1'b1;
  incrWrite;
  end
// On a read:
if (ReadReq==1'b1)
  begin
  ReadCmdr  = 1'b1;
  incrRead;
  end
join
(transition logic ...)
```

Make these changes in your code for the a_empty, normal, and a_full command logic.

In our modified design, we already have made sure that both address-increment tasks will exit simultaneously on concurrent read and write commands; we did this

by using only one posedge clock in the tasks, StateClock. From a previous Step, your tasks should have been rewritten this way:

```
task incrRead; // Still incorrectly synthesized (but not done yet).
  begin
  if (LatchR==1'b0)
    begin
    LatchR = 1'b1;
    @(posedge  StateClock) // Read must not block longer than write.
      ReadCount = ReadCount + 1;
    LatchR = 1'b0;
    end
  end
endtask
```

The synthesizer doesn't use fork-join blocks (netlists are concurrent anyway) and rejects them; so, each "fork" and each "join" in the combinational block should be surrounded by a DC macro test to prevent the synthesizer from reading it.

One last refinement: Let's change the state declarations to sized localparams. And, let's rename the states from *statename* to *statename*S, a final 'S' to make clear that this is a state identifier and not a common English word. For example, "localparam[2:0] emptyS = 3'b000;", etc. These name changes can be done by a quick search-and-replace.

Simulate your Deserializer again. In this simulation, be sure at some point first to disable reads until the FIFO goes full; then, read it long enough so that it goes empty. The design should be much closer to being entirely correct than when we started this lab.

Step 8. Changing the FIFO state machine for correct synthesis. The current design will synthesize to a netlist, but the netlist still will be incorrect. The new PLL will be fine, but the FIFO remains a synthesis problem for two reasons:

- The StateClock generator is an oscillator implemented as an always block with a change-sensitive event control containing nets not expressed in the block. This risks that the synthesizer will allow these nets to dangle.
- The address-updating tasks include embedded edge-sensitive event controls. Although edge-sensitive, the tasks are not called on that edge; and, so, the embedded controls imply complex latches in the state machine combinational logic, which will not be synthesized correctly (as we know).

We shall resolve these problems in this Step.

Step 8A. Testbench for source and netlist. Before anything else, let's be sure our FIFO testbench is adequate for unmodified use either with our FIFO source verilog or with a synthesized FIFO netlist.

At the start of the tests, we want pulsed write and read requests, as in previous FIFO testbench versions. After this, we want a run of back-to-back writes which fills the FIFO, followed by a similar run of reads which forces it empty. Finally, we

want a run of mixed writes and reads which includes at least a few simultaneous read and write requests.

Modify your FIFO testbench to achieve these goals. Use two identical clock generators, one clocking the `FIFOTop ClkR` pin, and the other `ClkW`. If time is short in the lab, consider just copying all or part of the `FIFOTopTst.v` testbench in `Lab23_Ans/Lab23_AnsStep08A`. Don't bother about details of design functionality at this point.

Step 8B. Synthesizable `StateClock` generator. In `FIFOStateM`, we can replace the `always` block with one which is sensitive only to the two input clocks; this will be synthesizable. For example,

```
reg StateClockRaw;
wire StateClock; // See code example below.
//
always@(ClkR, ClkW)
  StateClockRaw = !(ClkR && ClkW);
```

An *xor* (`^`) would be more symmetrical than the *and* (`&&`), but it would not work if `ClkR` and `ClkW` were closely in phase; so, we use a *nand* expression. An *and* also would be fine, but in CMOS technology, *nand* usually is simpler and faster than *and*.

The clock is named `StateClockRaw`, and not `StateClock`, for the following reason:

A possible new problem is that there is no HDL delay here any more, and therefore no more inertial-delay filtering. This means that as the read and write clocks drift in phase independently, the simulator will be allowed to produce `StateClockRaw` glitches of arbitrarily short duration – and there will be no clear relationship to the performance in the synthesized netlist. This actually has been a simulation problem here, all along, ever since we deleted all the delay expressions in previous lab Steps.

The solution is to connect the `StateClockRaw` to a library component delay cell which will filter out the narrowest glitches; the delay cell then can be used to drive the `StateClock`. The same glitch filtering then will occur during simulation both in the source and in the synthesized netlist. For a delay cell, we might as well use the same library cell as we used in `Lab22` for the synthesizable `PLL`.

We can preserve the delay cell from removal during synthesizer optimization by means of an embedded *TcL* script near the verilog instantiation. The added code may be written this way:

```
// Glitch filter:
DEL005 SM_DeGlitcher1 ( .Z(StateClock), .I(StateClockRaw) );
//
//synopsys dc_tcl_script_begin
// set_dont_touch SM_DeGlitcher1
// set_dont_touch StateClock*
//synopsys dc_tcl_script_end
```

After commenting out the StateClock generator of previous Steps and deriving it from the read and write clocks as above, you will have to add a verilog simulation model of the delay cell to your .vcs file list to simulate the FIFO. Do this by referencing the (modified) library file, *LibraryName*_v2001.v, copied or linked in your Lab23 directory. Precede the file name with -v in your .vcs file, so that the simulator will search for and compile the one model, instead of compiling the whole library.

Don't bother synthesizing yet; but, try simulating the FIFO briefly. You may have to modify the testbench a little to get the FIFO to run all the way to full and to empty at least once during the simulation. Don't worry about simulation details; we shall be changing the FIFO, and thus perturbing the functionality, again.

Step 8C. Synthesizable FIFO address-generating tasks. We have to reexamine and change every state in our FIFO state machine combinational block.

We know that the event controls inside the old tasks will not synthesize correctly. So, let us separate the event controls from the task logic into new always blocks. Instead of embedding an edge-sensitive event control in each task, we can use independently StateClock'ed always blocks, clocked on the positive edge, to update the address counters. An edge-sensitive always block can update a counter conditionally and be synthesized correctly.

With counter updates isolated this way, the tasks can be rewritten fully change-sensitive, so that they might synthesize correctly in the state machine combinational block.

Our new tasks (shown below) will latch address state and issue read or write commands to the FIFO register file; the always blocks synchronously will change the addresses used by the register file.

With this implementation, the new always blocks also should be made to perform the fullS and emptyS address initializations on reset. Thus, all manipulation of register-file addresses can be encapsulated in the new posedge always block logic.

Now let's get down to implementation details:

Let's adopt the convention that a task called with an input argument of 1'b1 will request an address increment, unless one is scheduled already on the current clock. A task called with a 1'b0 input argument will deassert the command controlled by that task. One way to write the required always blocks and their tasks would be this way:

First, we declare the following reg's for use by the tasks and by the next-state logic:

```
reg[AWid-1:0] ReadAr, WriteAr       // Address counter regs.
            , OldReadAr, OldWriteAr // Saved posedge values.
            , tempAr;               // For address-wrap compares.
```

The ReadAr and WriteAr also should be used on the right sides of the usual continuous assignments to the corresponding FIFOStateM output ports, ReadAddr and WriteAddr. The Old* are to retain state across calls, because of loss of the sequencing capability we had with the discarded @ edge-sensitive event controls.

After this, we may declare each new `always` block and its respective task. The tasks generally have to treat FIFO empty and full states specially.

For a register-file read, the `always` block:

```
always@(posedge StateClock, posedge Reset)
  begin : IncrReadBlock
  if (Reset==1'b1)
       ReadAr <= 'b0;
  else begin
       if (CurState==emptyS)
            ReadAr <= 'b0;
       else if (ReadCmdr==1'b1)
              ReadAr <= ReadAr + 1;
       end
  end
```

For a register-file read, the new task:

```
task incrRead(input ActionR);
  begin
  if (ActionR==1'b1)
       begin
       if (CurState==emptyS)
            begin
            ReadCmdr  = 1'b0;
            OldReadAr = 'b0;
            end
       else begin
            if (OldReadAr==ReadAr) // Schedule an incr.
                 ReadCmdr = 1'b1;
            else begin // No incr; already changed:
                 ReadCmdr  = 1'b0;
                 OldReadAr = ReadAr;
                 end
            end
       end
  else begin // ActionR is a reset:
       ReadCmdr  = 1'b0;
       OldReadAr = 'b0;
       end
  end
endtask
```

If the FIFO is empty, its first value can be written anywhere, so the `emptyS` starting value of `ReadAr` in principle need not be specified. However, the counter arithmetic we have adopted for state transitions requires that both counters start from a known value in the empty state. Therefore, the transition to `emptyS` must initialize both pointers to the same value, which might as well be `0`. The write pointer must be initialized in its own `always` block, if we want to conform with the synthesis requirement that no `reg` may be assigned in more than one `always` block.

When the FIFO is full, its data occupies all storage locations, and the address of the first datum to be read is predetermined. Therefore, the *writing* `always` block, the only one allowed to modify the value of `WriteAr`, must be the one to initialize `WriteAr` to bring it from its undefined state to its one, correct value on the first read in `fullS`. That value equals the value of `ReadAr`, which, of course, will be incremented on the next `posedge` of the `StateClock`.

So, for a register-file write, the new `always` block:

```
always@(posedge StateClock, posedge Reset)
  begin : IncrWriteBlock
  if (Reset==1'b1)
       WriteAr <= 'b0;
  else begin
       case (CurState)
       emptyS: WriteAr <= 'b0;    // Set equal to read addr.
        fullS: WriteAr <= ReadAr; // Set equal to first valid addr.
       endcase
       if (CurState!=fullS && WriteCmdr==1'b1)
         WriteAr <= WriteAr + 1;
       end
  end
```

And, finally, for a register-file write, the new task:

```
task incrWrite(input ActionW);
  begin
  if (ActionW==1'b1)
       begin
       if (CurState==fullS)
              begin
              WriteCmdr  = 1'b0;
              OldWriteAr = ReadAr;
              end
         else begin
              if (OldWriteAr==WriteAr) // Schedule an incr.
                   WriteCmdr = 1'b1;
              else begin // No incr; already changed:
                   WriteCmdr  = 1'b0;
                   OldWriteAr = WriteAr;
                   end
              end
       end
  else begin // ActionW is a reset.
       WriteCmdr  = 1'b0;
       OldWriteAr =  'b0;
       end
  end
endtask
```

All task calls in the FIFOStateM combinational block now have to be modified to pass a 1'b1 or 1'b0. A call to both of incrRead and incrWrite should be added to the reset block in emptyS and each passed 1'b0; all other task calls should be passed 1'b1. LatchR and LatchW should be removed from the design.

All reference in the combinational block to ReadCount, WriteCount, or tmpCount should be replaced by ReadAr, WriteAr, or tempAr, which latter also should be used in the continuous assignments to FIFOStateM output ports; none of these new identifiers should appear in the combinational block except in transition logic expressions.

We still have changes to make in the combinational block. First, why not delete all the fork-joins? Consider them a temporary simulation kludge, and remove them all, with their `DC controls.

Because we are eliminating fork-joins, we have to simulate a simultaneous read and write request somehow. The individual task calls would be enough; but, we also have to deassert read and write commands on clocks not requesting either. We can do all this explicitly by using a case statement in each branch of the combinational block, as shown below.

Our new tasks don't block on simulation-scheduled events any more, so we can call them individually if there is just one of read or write requested, or we can call them both (procedurally). Here is how to do this in any of a_emptyS, a_fullS, or normalS:

```
...
case ({ReadReq,WriteReq})
2'b01: begin
       incrWrite(1'b1); // On a write.
       ReadCmdr = 1'b0;
       end
2'b10: begin
       incrRead(1'b1); // On a read.
       WriteCmdr = 1'b0;
       end
2'b11: // On a read and write:
       begin
       incrRead(1'b1);
       incrWrite(1'b1);
       end
default: begin // No request pending:
         ReadCmdr = 1'b0;
         WriteCmdr = 1'b0;
         end
endcase
...
```

In emptyS or fullS, there is only one operation possible, so we call just the one task for those; but, we also should deassert the command when it is not requested.

Finally, we have to break the rules to get the FIFO to synthesize. Normally, a verilog state machine is designed with a clocked block to determine state transitions and a combinational block to operate the transition logic and the rest of the machine. This usually works best. But, a FIFO is a strange animal.

Notice that we already have to use clocked logic to update the address counters, and that this is inconsistent with everything but state register updates in a purely combinational block. Attempting synthesis with only the preceding changes still will result in pathological latch synthesis and incorrect functionality. We must prevent latches; we can do this by clocking what up to now was our "combinational" block.

To avoid race conditions, modify the combinational block to be sensitive to the `negedge` of `StateClock`. This, incidentally, will allow us to associate an asynchronous reset to the erstwhile combinational block. Your new "combinational" block now should resemble something like this:

```
...
always@(negedge StateClock, posedge Reset)
  begin
  if (Reset==1'b1)
     begin
     // Reset conditions:
     NextState = emptyS;
     incrRead(1'b0); // 0 -> reset counter.
     incrWrite(1'b0); // 0 -> reset counter.
     end
  else
  case (CurState)
  emptyS:// (The other previously combinational logic goes here)
  ...
  endcase
...
```

After these changes, simulate the FIFO again until it works well with your FIFO testbench.

Optional: If time permits, ***synthesize a netlist*** from this design and simulate the resulting netlist. For correct synthesis, some setup and hold problems will have to be prevented; to see how to do this, examine the `FIFOTop.sct` file provided on the CD-ROM in `Lab23/Lab23_Ans/Lab23_AnsStep08C/Deserializer/FIFO`. This synthesis will take some time, perhaps a half-hour.

To reduce the time, let the synthesizer run until it has been in the "Area Recovery Phase" for about 5 minutes; then, press {control-*C*} *once* and use the text-based menu (which will appear after a little while) to abort optimization. The synthesis script will run to completion, leaving you with a suboptimal but functionally correct netlist.

The FIFO netlist should simulate more or less as well as the source FIFO. There may remain timing issues, but don't bother perfecting your FIFO testbench now; go on to the rest of the lab.

Step 8D. (***Optional***) Verifying that the `Deserializer` from Step 8C does not yet synthesize correctly. If time permits, try the synthesis described in this Step; otherwise, just read the description here of the result.

Without any attempt to simulate the `Deserializer` from the top, change to the `Lab23/Deserializer` directory, run the synthesizer using the `Lab23/Deserializer.sct` script provided, and examine the waveforms resulting from an attempt to simulate the resulting netlist.

The netlist simulation will fail. You should be able to see that `ParValid` does not control FIFO writes correctly. The parallel-bus clock from the sending domain consists of approximately 160-ps wide pulses, repeated at about 2 ns intervals. The problem therefore is in the `DesDecoder`, which generates this parallel clock. Even though the main `DesDecoder` synthesis problem, edge-triggering task calls, was fixed, and the source verilog simulated correctly, the synthesized netlist is not correct.

Step 9. Synthesizing the `DesDecoder`. Change to the `DesDecoder` subdirectory and read through the source code in `DesDecoder.v`. The `ClkGen` task definitely is a problem: It is called in a selectively change-sensitive `always` block and thus implies nonstandard latching. As usual, such a latch is too complex to expect the synthesizer to be able correctly to fulfill the design intent.

We have to rewrite the parallel-clock generator. We can start by cleaning up the rest of the `DesDecoder` module: Rename all `*Reg` to `*r`, if not done already; then, remove any remaining commented-out task remnants.

The clock-generator problem can be solved by rewriting the `ClkGen` task as an `always` block named `ClkGen` and sensitive to the edge of `SerClock` opposite that of the other `always` blocks in this module; this means `ClkGen` should be sensitive to the positive edge. For example,

```
always@(posedge SerClock, posedge Reset)
  begin : ClkGen
  if (Reset==1'b1) // Respond to external reset.
      begin
      ParClkr <= 1'b0;
      Count32 <= 'b0;
      end
  else begin // If not a reset:
       if (SerValid==YES)
         begin
         // Resynchronize this one:
         if (doParSync==YES)
              begin
              ParClkr <= 1'b0; // Put low immediately.
              Count32 <=  'b0;
              end
         else begin
              Count32 <= Count32 + 1;
              if (Count32==5'h0) ParClkr <= ~ParClkr;
              end
         end // if SerValid.
       end // not a reset.
  end // ClkGen.
```

Edge sensitivity will eliminate complex latches, but further improvement is required. Start by removing the `SerClk` gate from the `ClkGen` block and putting it in a conditional continuous assignment,

```
assign SerClock = (SerValid==YES)? SerClk : 1'b0;
```

After this, if the `Shift1` still uses a temporary shift register, make it more compact by changing it to shift the `FrameSR` directly. The following should work well:

```
always@(negedge SerClock, posedge Reset)
  begin : Shift1
  // Respond to external reset:
  if (Reset==YES)
       FrameSR <= 'b0;
  else begin
       FrameSR    <= FrameSR<<1;
       FrameSR[0] <= SerIn;
       end
  end
```

Let's get a little fancy here, like someone designing configurable IP. In the simulation, you may have noticed the narrow, glitch-like pulses produced on the `ParValid` output. This net should be asserted for longer durations, several `SerClock` cycles, at least.

We cannot effectively program a pulse digitally, but, we can introduce a small, fast SerClock cycle-counting timer which can deassert ParValid after any convenient count. Add this to the beginning of the DesDecoder module:

```
localparam ParValidMinCnt = 8; // Minimum number of SerClocks
                               //    to hold ParValid asserted.
localparam ParValidTWid = // Width of ParValidTimer reg.
                   (    2   > ParValidMinCnt )? 1
                 : ( (1<<2) > ParValidMinCnt )? 2
                 : ( (1<<3) > ParValidMinCnt )? 3
                 : ( (1<<4) > ParValidMinCnt )? 4
                 : ( (1<<5) > ParValidMinCnt )? 5
                 : ( (1<<6) > ParValidMinCnt )? 6
                 : 7; // Thus, width is declared automatically.
//
reg[ParValidTWid-1:0] ParValidTimer;
// ...
```

This approach will make it possible to adjust the width of the ParValid assertion, should system considerations require it.

Then, the Unload32 block may be rewritten along these lines:

```
always@(negedge SerClock, posedge Reset)
  begin : Unload32
  if (Reset==YES)
       begin
       ParValidr <=  NO; // Lower the flag.
       ParOutr   <= 'b0; // Zero the output.
       ParValidTimer <= 'b0;
       end
  else begin
       if (UnLoad==YES)
            begin
            ParOutr       <= Decoder; // Move the data.
            ParValidr     <= YES; // Set the flag.
            ParValidTimer <= 'b0;
            end
       else begin
            if (ParValidTimer<ParValidMinCnt)
              ParValidTimer <= ParValidTimer + 1;
            if (ParValidTimer==ParValidMinCnt && ParClk==1'b0)
              ParValidr <= NO; // Terminates assertion.
            end
       end // UnloadParData.
  end
```

After simulating the rewritten DesDecoder, synthesize it and verify that the netlist simulates the same way as did the verilog source. Do not synthesize or simulate the entire Deserializer yet.

Step 10. (***Optional***) Synthesizing and simulating the `Deserializer` netlist. We complete this lab by demonstrating that the `Deserializer` from Step 9 now can be synthesized to a netlist which functions correctly at least occasionally.

Under proper constraints, with a 32-word FIFO, the `Deserializer` will synthesize as-is in about 15 hours to some 75,000 transistor-equivalents. However, this netlist probably will not simulate correctly. If you wish to run this synthesis, there is a `.sct` file available in the answer directory for your reference. It might be a good idea to start this synthesis just before leaving for the day.

A preliminary synthesis already has been done for you at this stage, producing a bad netlist in about an hour (ten or fifteen minutes with incorrect constraints – see the Step 8C box for a trick to get a quick and dirty netlist in 5 minutes or so). The result is in a `BadNetlist` subdirectory in your answer directory for Step 10.

There probably will be some debugging to do before the `Deserializer` will simulate reasonably even before synthesis. The main simulation problem probably will be synchronization of the serial clocks: The frequencies must stay within about 1/32 (close to 3%) before a complete 64-bit arriving packet will be aligned correctly and decoded; therefore, the testbench sending-domain serial clock should be set fairly close to the free-running `Deserializer` PLL base frequency. I have used a half-period delay of 15.5 ns successfully.
Some things to consider:

- The `DesDecoder` synthesizable parallel clock generator now is called on just one edge, instead of on any change; therefore, it will run at an average of half of the previous speed. The receiving-domain clock frequency must be checked.
- The `DesDecoder` may call `doParSync` too often, thus perhaps causing the extracted parallel clock to fail because of frequent erroneous resynchronization on zero data rather than on `PAD0`.
- The PLL `` `VFO_MaxDelta `` possibly should be different for source simulation than for synthesis of a correctly-simulating netlist.
- The VFO operating limits in `VFO.v` probably should be defined simply as `` `DivideFactor ± `VFO_MaxDelta ``, if not already so
- The number of delay stages (`` `NumElems ``) in `VFO` should be set properly for correct source and netlist simulation. I have found 5 to be a good value here.
- In the top-level testbench, the total duration of the simulation and the sending-domain serial clock speed also may have to be tuned.

It is not important to have the `Deserializer` completed in this lab; there will be time for tuning later in the course. However, all the major submodules should simulate well by now on the unit level (`PLL`, `DesDecoder`, and `FIFO`), both as verilog source and as synthesized verilog netlist.

Success with the entire `Deserializer` system at this point would be indicated by just a couple of sporadic unloads of data from the FIFO onto the output parallel bus. Playing with the `DesDecoder` or the `PLL` probably will make more difference here than anything else. An example of the waveforms is given in Figs. 19.5, 19.6 & 19.7.

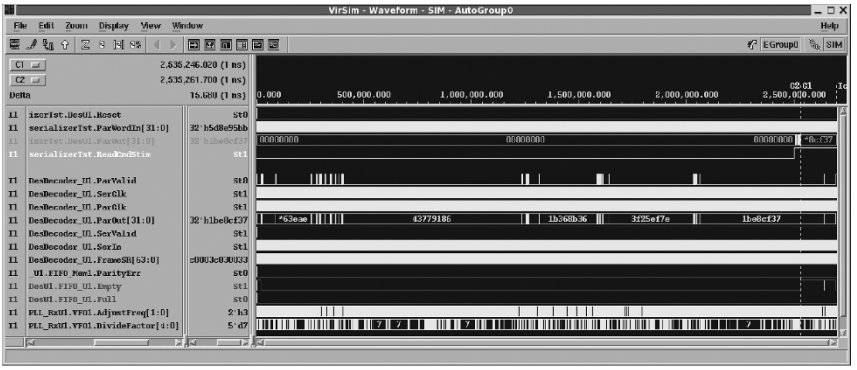


Fig. 19.5 First successful simulation result for synthesized `Lab23 Deserializer` netlist (not SDF back-annotated). The 32-word FIFO clearly goes from empty to a normal state; and, upon receiving-domain `ReadCmdStim`, the stored data are read out onto the parallel bus

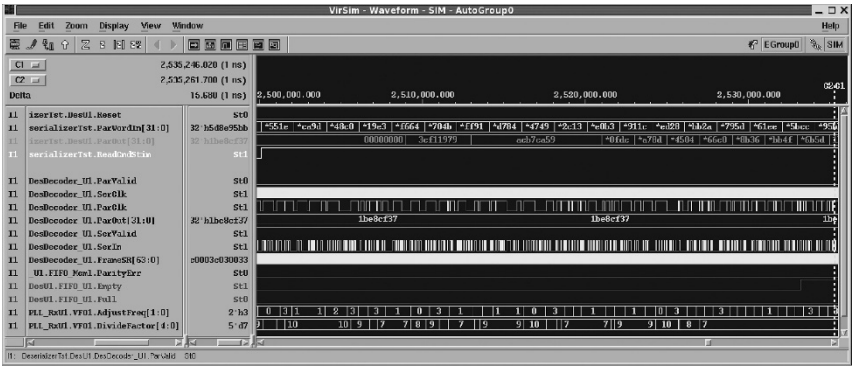


Fig. 19.6 Closeup of the stored data which are being read out from the `Deserializer` FIFO

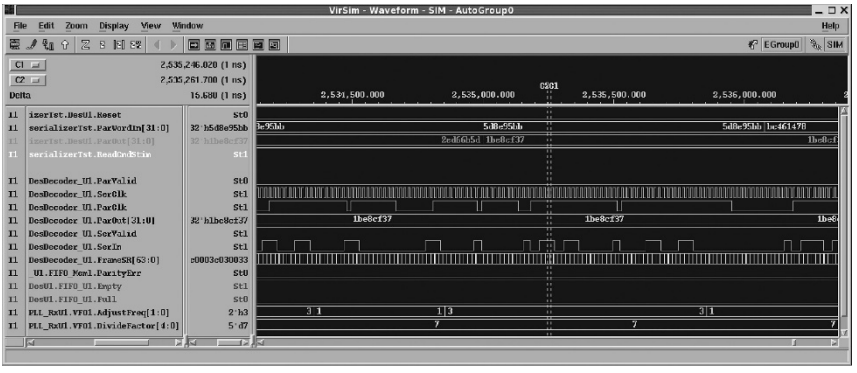


Fig. 19.7 Closeup of the arriving serial data during netlist simulation

19.2.1 Lab Postmortem

Think about the possible kinds of verilog debugging technique.

19.2.2 Additional Study

Optional; Review `fork-join` functionality in Thomas and Moorby (2002) section 4.9.

Chapter 20
Week 10 Class 2

20.1 The Serializer and The SerDes

Figure 20.1 is an update of the SerDes block diagram which was presented at the start of *Week 9 Class 2*. Notice the slight change in the location of the clock divide-by-two, which now is in the Serialization Encoder:

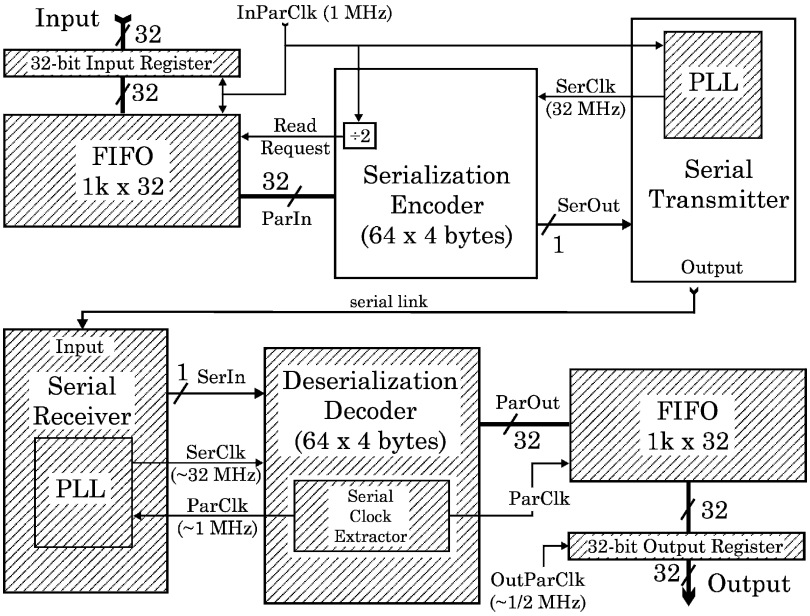


Fig. 20.1 Current status of the `SerDes` project. Hatched areas have been implemented

Although the serializer looks almost as complex as the deserializer, it is much simpler. For one thing, there is a well-defined 1 MHz clock with no need for extraction from a stream. The serializer PLL only has to do a phase-locked 32x frequency multiplication to provide the serial output clock. Also, the serializer's FIFO is in a single clock domain.

The Serializer will use exactly the same PLL and FIFO modules as the Deserializer; however, we shall have to create new SerEncoder and SerialTx modules for the sending functionality.

20.1.1 The SerEncoder Module

This module, implementing the Serialization Encoder, reads from the serializer's FIFO to create serial packets.

It shifts out each packet serially to the SerialTx.

20.1.2 The SerialTx Module

This module, implementing the Serial Transmitter, is what transmits the data on the serial line.

Because it contains the sending-side PLL, it is used to clock the SerEncoder shift register.

20.1.3 The SerDes

With a working Serializer, assembling the complete serdes is trivial: One merely needs instantiate a Serializer and a Deserializer in a containing SerDes module, connect them with a wire, and supply a testbench for the SerDes. This lab will complete our class project, although we shall use the design later when we study design-for-test.

20.2 SerDes Lab 24

Do this work in the Lab24 directory.

Lab Procedure

Step 1. Reuse the previous design. Start by creating a new SerDes subdirectory in the Lab24 directory with a Deserializer subdirectory under it.

Make a new, complete copy of the provided solution from Lab23/Lab23_Ans/Lab23_AnsStep10/Deserializer into the new Lab24/SerDes/Deserializer directory. If you wish to debug your own Lab23 work now, you may use your own files, but the details of the instructions below assume you are using the provided Lab23 answers.

There is no reason to copy over the `Lab23` unit-test or synthesis files at this time; if useful, they may be copied from the `Lab23` subdirectories later.

Step 2. Set up the `SerDes` files. Because the PLL and FIFO designs will be the same throughout the serdes (but different instances may be parameterized differently), some reorganization is appropriate.

The directory structure should be changed as shown in Fig. 20.2.

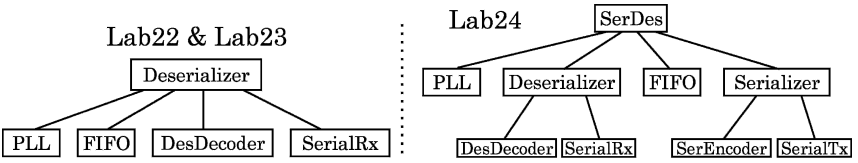


Fig. 20.2 File reorganization for Lab 24

Right next to the `Deserializer` subdirectory, create a new `Serializer` subdirectory in `SerDes`. Also, move the deserializer `PLL` and `FIFO` subdirectories up one level, so that `Serializer`, `Deserializer`, `FIFO`, and `PLL` all are in the same `SerDes` directory.

Move `Deserial.inc` into the `SerDes` directory, renaming it `SerDes.inc`.

Make up a simulator file list in the `SerDes` directory; name it `SerDes.vcs`, and verify the new arrangement by using `SerDes.vcs` to load the new design hierarchy into the simulator.

Step 3. Prepare the `Serializer` module. Under `Serializer`, create a new `SerialTx` subdirectory, and install a `SerialTx.v` file there, with an empty `SerialTx` module, instantiating the PLL. Use the header from `SerialRx.v` temporarily, if you want.

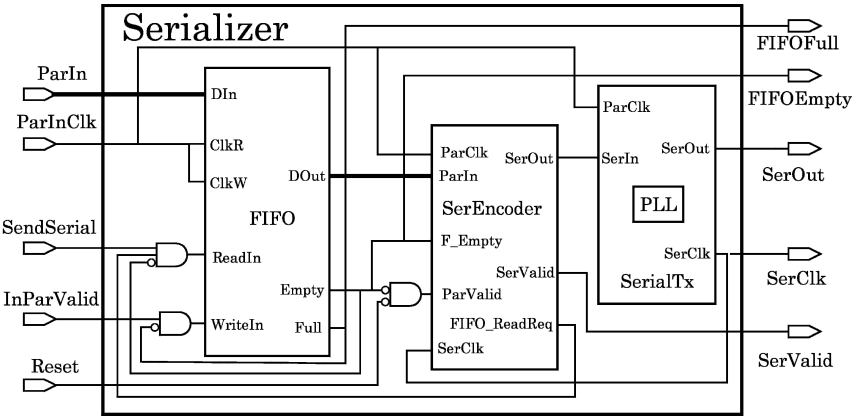


Fig. 20.3 Overall schematic of the completed `Serializer`. Not all `Resets` shown

Likewise create a SerEncoder subdirectory in Serializer; it should contain a SerEncoder module in SerEncoder.v. Copying the DesDecoder header into SerEncoder.v may be a good way to reuse your previous work. A schematic of the design is provided in Fig. 20.3.

The serial packet format will use the same framing on both ends of our SerDes, so, copy the pad localparam assignments and other shared information to a new include file, SerDesFormats.inc, located in the SerDes directory. In Lab23, these were in the DesDecoder module. Copy the comments explaining the format, too, in case debugging may become necessary. However, keep in mind that the specific format of the packets and their padding is parsed in detail from the DesDecoder verilog, and is formed by detailed verilog coding in the SerEncoder, so this new include file has no portability function. The file contents other than comments should be the following:

```
// SerDesFormats.inc:
localparam[0:0] YES  = 1'b1;   // For general readability.
localparam[0:0] NO   = 1'b0;
localparam[7:0] PAD3 = 8'b000_11_000;
localparam[7:0] PAD2 = 8'b000_10_000;
localparam[7:0] PAD1 = 8'b000_01_000;
localparam[7:0] PAD0 = 8'b000_00_000;
```

The value of factoring out the pad format this way is that if the pad format should be changed in the future, it will be changed the same way both for encoding and decoding, thus avoiding possible maintenance problems. However, any pad format change will require detailed rewriting in both the Ser and the Des.

In the Serializer directory, declare a Serializer module in a file named Serializer.v, and instantiate in it FIFOTop, SerEncoder, and SerialTx.

Use the corresponding Deserializer modules as guides; you probably won't have to add new ports or connections, although names and directions will change. Be sure to retain parameters to size the FIFO and the address and data ports; the Deserializer parameters should be used directly for this. Use the same parameter names and defaults for the new serializer as you did for the Deserializer; the values always can be overridden differently, if desired, during instantiation.

Your Serializer ports should be:

Outputs: SerOut, SerValid, FIFOEmpty, FIFOFull, SerClk.
Inputs: ParIn (32 bits), InParValid, ParInClk, SendSerial (the request to send), Reset.

In the Serializer, you will want a control input to assert a request to *read* the (parallel) FIFO output into the serial encoder. Recall that in the Deserializer, you had instead a control input to *read* the FIFO output into the receiving system. By controlling FIFO read at both ends, the FIFO is most properly used as a buffer during continuous communication. Of course, on a FIFO-full or FIFO-empty condition, write may have to be controlled, too; but, we shall leave these problems to the protocol of the system containing our serdes.

The Serializer should compose a FIFO write command to move new data into the FIFO from the parallel bus. Thus, the Serializer controls FIFO read and write requests, in addition to routing the 1-MHz parallel-data input clock (ParInClk) to the FIFO.

The Serializer also should provide a FIFO-valid control (F_Valid) for the ParValid input pin of the SerEncoder, to flag usable parallel data from the FIFO; this can be done very simply by a continuous assignment,

```
assign F_Valid = !F_Empty && !Reset;
```

The other controls just described may be provided this way:

```
assign F_ReadReq  = !F_Empty && SerEncReadReq && SendSerial;
assign F_WriteReq = !F_Full  && InParValid;
```

Here, F* refer to FIFO pins, SerEncReadReq is FIFO_ReadReq from the SerEncoder, and InParValid and SendSerial are Serializer input ports.

After a first cut at the Serializer, create a do-nothing testbench in a separate file named SerializerTst.v in the Serializer directory. Use it to simulate briefly to check your connections and file locations.

Also, create empty placeholder files named SerDes.v and SerDesTst.v in the SerDes directory.

The resulting file locations should be as shown in Fig. 20.4.

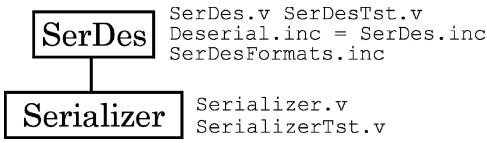


Fig. 20.4 File locations for the Serializer

Step 4. Complete the serial transmitter. This module, SerialTx, should be very simple, like the SerialRx module of the Deserializer. All it need contain is (*a*) a simple continuous assignment statement passing the serial data through from input port to output port and (*b*) a properly connected PLL instance.

Step 5. Complete the serialization encoder. Refer to Fig. 20.3 above, which gives a Serializer block diagram, and to the schematic Fig. 20.5, which gives connectivity details.

We shall design this module for synthesis.

The SerEncoder module has to use the serial and parallel clocks to frame the data and transfer it serially to the SerialTx module. The clocks are independent of this module's functionality and are guaranteed by the SerialTx's PLL submodule to be phase-locked, so there is no need to extract phase or frequency information.

However, the serialization has to be buffered so that it is not interrupted unless the FIFO, which provides the parallel-side input data, has become empty.

Recall that the 2:1 divided clock on the deserializer end of the serdes (`Lab22`) could be provided by the `Deserializer` testbench; this was because the Des was operating at 1 MHz and was providing correct parallel data all by itself; it was up to the receiving domain to use the latched `ParOut` data properly. This is not feasible on the Ser end, because the Ser not only has to operate at 1 MHz, too, but it also has to determine the rate at which parallel data will be copied in and serialized. This means that a 1/2 MHz derived clock has to be part of the `Serializer` design; we shall derive it in the `SerEncoder`.

We shall obtain the 1 MHz clock (`ParClk`) provided to the `Serializer` from an external source (initially, your `SerializerTst` verilog testbench). The `SerEncoder` therefore must contain a block doing a simple frequency-halving of the `ParClk` input and using it to request FIFO reads.

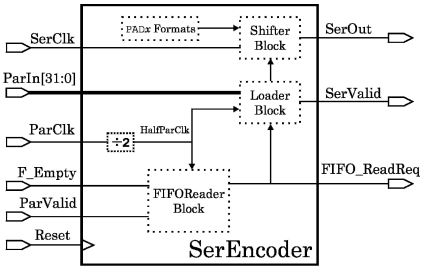


Fig. 20.5 The `SerEncoder`. Dotted outlines indicate blocks of code, not hierarchy

In addition to this, we shall define three other functional blocks within the `SerEncoder` module, as shown in Fig. 20.5:

- `Loader` block: Read on every `posedge` of the half-speed `ParClk` (`HalfParClk`), this `always` block will do two things:
  1. Load the `SerEncoder`'s 32-bit buffer with a new word from the FIFO, or with all `0`'s if the FIFO was empty.
  2. Assert or deassert `SerValid`, depending on whether the FIFO was empty.
- `Shifter` block. This `always` block will be read on every `posedge` of `SerClk`, and exclusively will control the serial data values.

  `Shifter` simply lets its counter wrap at 64 and uses the count on each clock to determine (mux) whether a `PAD` bit, or a data bit from `SerEncoder`'s input buffer, is applied to the serial line out.
- FIFO reader block. This block asserts a FIFO read request on every other `ParClk` to make available a new 32-bit data word for framing. It depends upon the 1/2 MHz clock, `HalfParClk`, derived from `ParClk`. There is no reason not to implement this block as simple combinational logic, for example as,

```
assign FIFO_ReadReq
       = HalfParClkr && !F_Empty && ParValid && !Reset;
```

The complete block diagram of `SerEncoder` is shown in Fig. 20.5.

Step 6. Use `SerializerTst` to simulate the `Serializer` until it operates correctly. One thing to check is that the FIFO should stop accepting writes when it is full. Also, parallel data should be flagged as invalid when the FIFO is empty. When the FIFO goes empty, the serial data should be flagged invalid when the last valid packet has been shifted out.

Typical `Serializer` simulation results are shown in Figs. 20.6 through 20.8.

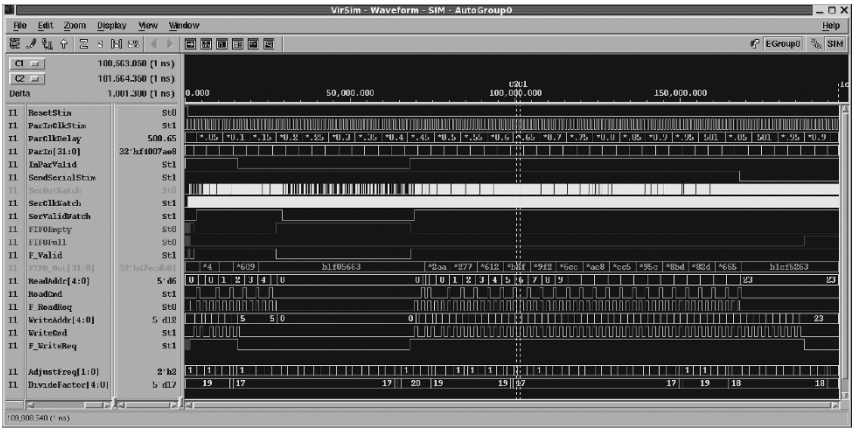


Fig. 20.6 Overview of the first good `Serializer` verilog source simulation. The FIFO clearly goes full and empty under reasonable conditions

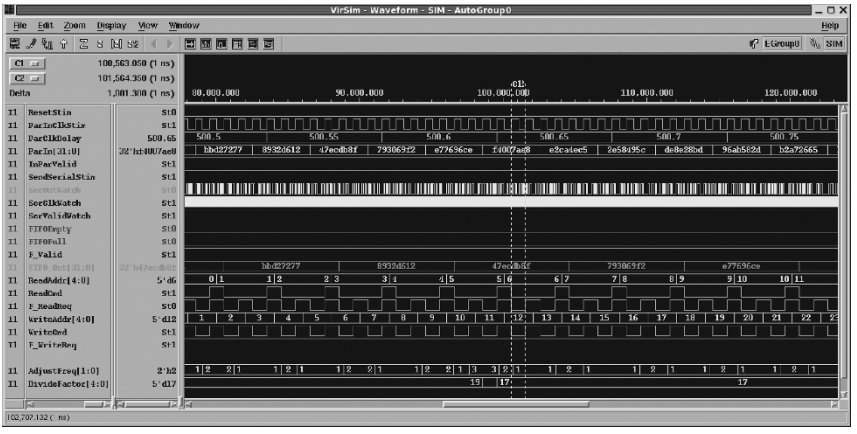


Fig. 20.7 Zoom in on first good `Serializer` source simulation, showing proper handling of the individual parallel-bus words

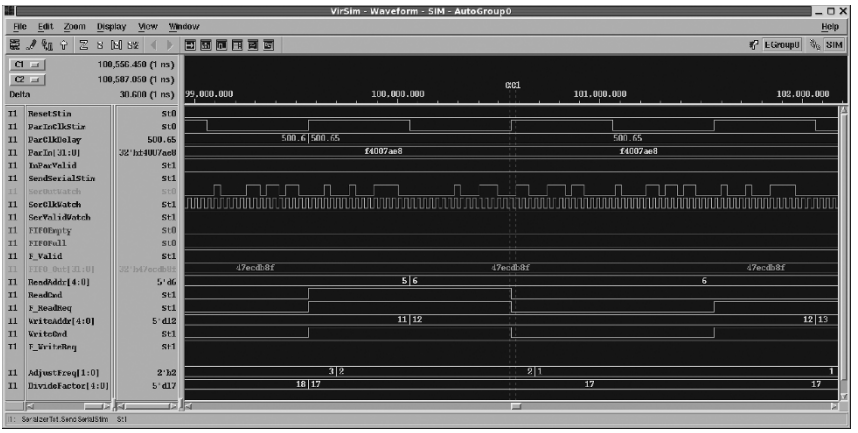


Fig. 20.8 Very high zoom in on the `Serializer` source simulation, showing individual serial-line clock cycles

Step 7. `SerDes` simulation. Connect the `Serializer` serial output to the `Deserializer` serial input by instantiating both in a new `SerDes` module. You probably will have to rename some wires, for example to make collateral outputs from both halves distinct.

Simulate, using at first your `SerializerTst` testbench (renamed to `SerDesTst`). Any serial transfer at all should be considered a complete success at this point. Depending on your FIFO state machine design, you may have to assert a reset twice to initialize everything at the beginning of the run.

Some typical `SerDes` simulation results are shown in Figs. 20.9 and 20.10.

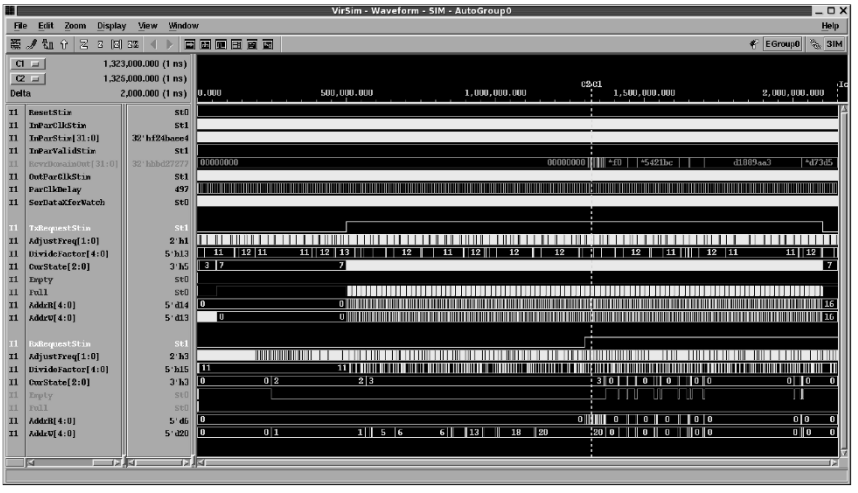


Fig. 20.9 First complete `SerDes` verilog source simulation. The sending and receiving FIFOs seem to be operating reasonably

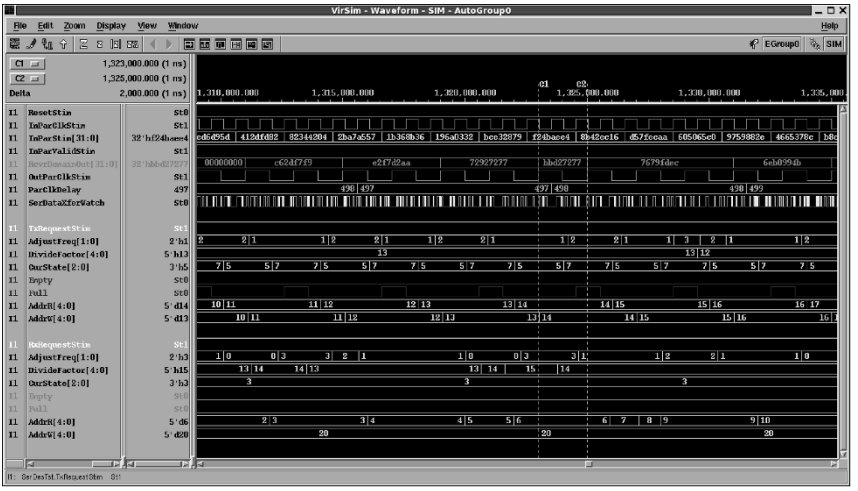


Fig. 20.10 Closer zoom in on the first complete `SerDes` verilog source simulation, showing individual parallel-bus words in the receiving domain

Optional: After your `SerDes` is working, modify the `Deserializer` to have two different parallel data outputs, one clocked out as previously, on the receiving-domain clock, and a new one clocked out on the deserializing PLL embedded clock (`ParClk`), which is in a different domain.

Then, simulate. Notice the skew in the appearance of the 32-bit, clocked-out data. This is an illustration of the digital side of the clock-domain synchronization problem. Of course, a digital simulator can not display the more serious analogue problem of intermediate-value hangups which we discussed in *Week 8 Class 1*.

Step 8. `SerDes` unit-level synthesis. A synthesis of the full `SerDes` system will take too long for classroom scheduling at this point, because system timing refinements may require several full `SerDes` synthesis iterations to get the complete design working.

For this lab session, just verify that each major component of the `SerDes` can be completed on the unit level: Synthesize, under reasonable constraints, the following, each in its own, preexisting subdirectory:

- `PLL`
- `FIFO`
- `Deserializer.DesDecoder`
- `Serializer.SerEncoder`

The `Deserializer.SerialRx` and the `Serializer.SerialTx` are essentially empty wrappers, containing just a `PLL` instance, so there is no reason to synthesize them now: Just do `PLLTop` alone.

You may wish to look at the synthesis scripts in the answer subdirectories. Although the `DesDecoder` is a relatively small design, it requires tight setup and

hold constraints for a perfect netlist, so the synthesizer will take hours (adjusting for the design rules) to finish. Read the comments in the answer script provided to see how to sidestep this long wait.

Likewise, the `FIFO`, a large design for a single module, will take a very long time. Check the comments in the `FIFO` answer synthesis script for advice on obtaining a usable netlist early in the run.

After synthesis of the individual units listed above, use the same simulation testbench as you did for the source, to simulate each synthesized netlist. You should consider reusing simulation testbenches from previous labs as starting points. The unit-level netlists should simulate almost as well as, or better than, the source verilog for them.

Optional: During each unit synthesis, write out an SDF file; then, use the original *TSMC* verilog simulation core library (`tcbn90ghp.v`, not the timing-kludged `*_v2001.v` used so far everywhere) to simulate the netlist using the best possible wireload-model delays. The delays won't make much difference, but the TSMC timing checks may indicate setup or hold problems requiring attention in a design intended for fabrication.

You may leave the synthesized netlists and side files in the individual subdirectories; they will not be used in higher-level simulation or synthesis file lists.

Figures 20.11 through 20.19 show some typical netlist simulation results, using the approximated timing in the provided verilog-2001 library. SDF back-annotation makes for a slight difference in the simulation waveforms.

The `PLL`:

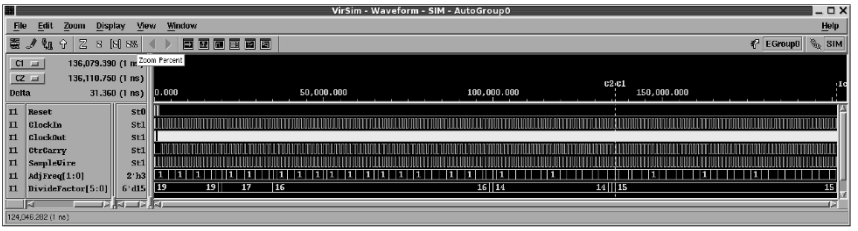


Fig. 20.11 Overall view of synthesized `PLL` netlist simulation

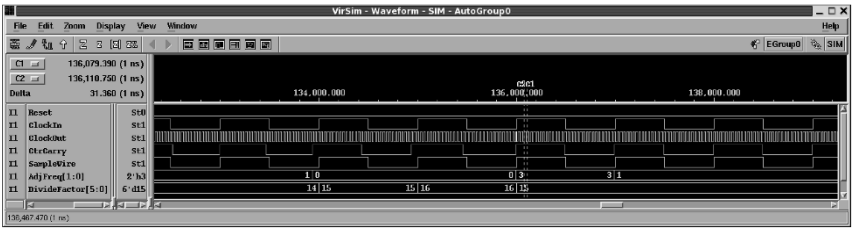


Fig. 20.12 Zoomed-in view of synthesized `PLL` netlist simulation, showing individual serial clock cycles

The `FIFO`:

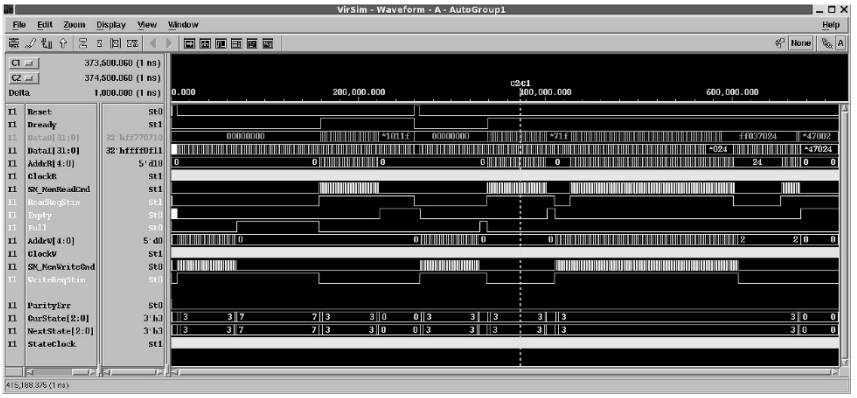


Fig. 20.13 Overall view of synthesized `FIFO` netlist simulation

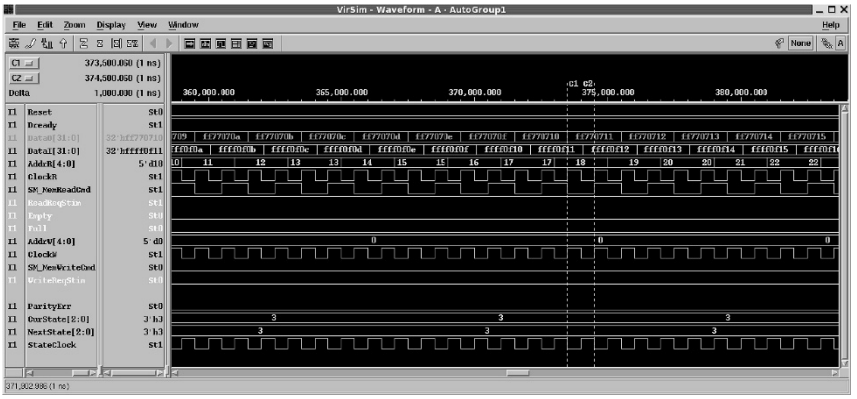


Fig. 20.14 Zoomed-in view of synthesized `FIFO` netlist simulation, showing individual 32-bit word transfers

The `DesDecoder`:

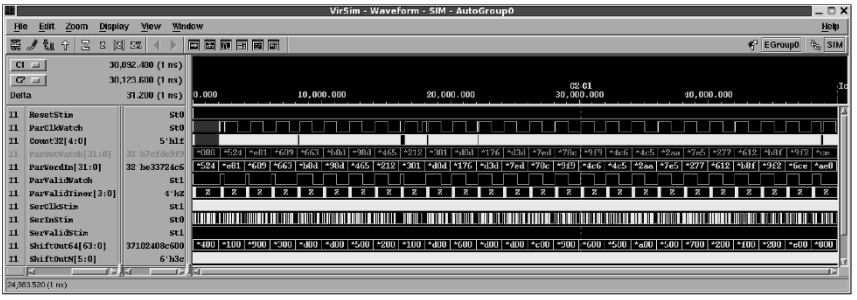


Fig. 20.15 Deserialization decoder (`DesDecoder`) overall view of synthesized netlist simulation

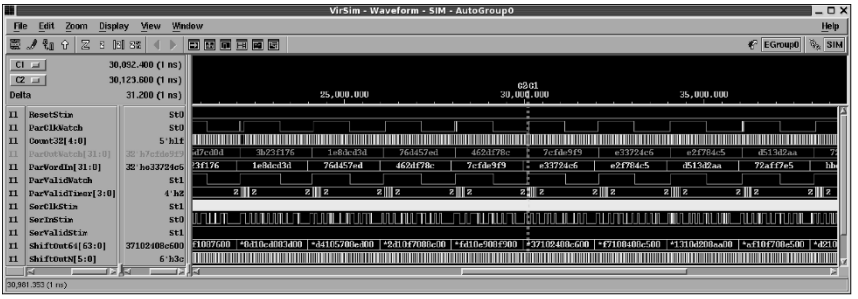


Fig. 20.16 `DesDecoder` netlist simulation, zoomed in to resolve individual parallel-bus words

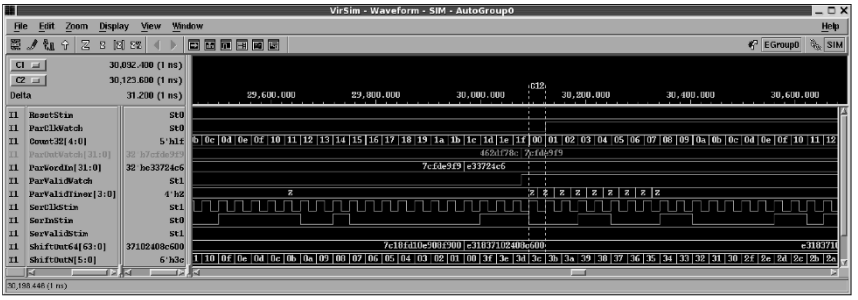


Fig. 20.17 `DesDecoder` netlist simulation, zoomed in to resolve individual serial clock cycles

The `SerEncoder`:

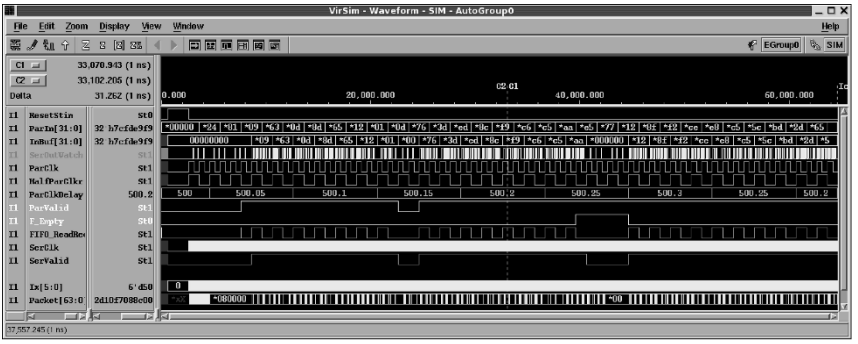


Fig. 20.18 The `SerEncoder`: Overall view of synthesized netlist simulation

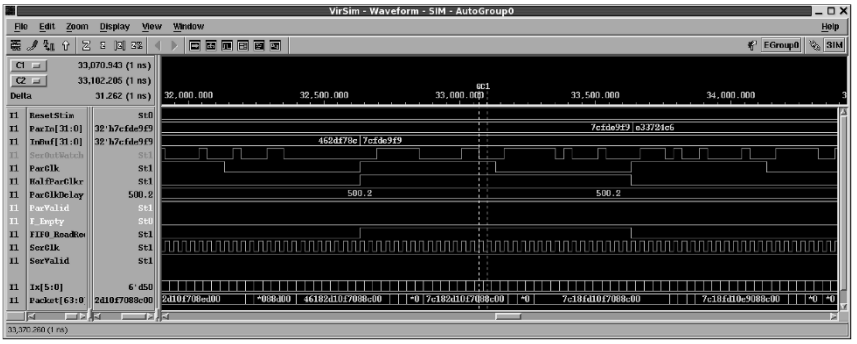


Fig. 20.19 Zoomed-in view of `SerEncoder` netlist simulation, showing individual serial clock cycles

20.2.1 Lab Postmortem

What might be possible incompatibilities of the Ser vs. Des devices?
What about assertions and timing checks for the serdes?

20.2.2 Additional Study

Optional: Compare your shift register with the switch-level MOS shift register in Thomas and Moorby (2002) section 10.1. Why would you want to write a switch-level model if a logic synthesizer was available?

Chapter 21
Week 11 Class 1

21.1 Design for Test (DFT)

21.1.1 Design for Test Introduction

Design for Test (DFT) is more an orientation, or perhaps a methodology, than a design technique. DFT means that the hardware will be testable, which is to say, its functionality will be verifiable, after implementation. Testability usually is planned on the assumption of two major sources of malfunction, design errors and hardware failures.

Design errors can be found during testing only by observing a failure of functionality. Most of these errors will be found and fixed during simulation or synthesis; this means testing both of the design source and of the back-annotated netlist following floorplanning or completed layout. Errors of logic or of timing can be found and corrected before tape-out for mask creation.

Hardware failures, of course, can be discovered only after implementation and production of the hardware. The most serious of these are fabrication quality or physical design errors in which one or more individual gates become stuck in one or another nonfunctional state in every unit produced. This kind of highly localized failure cannot be detected unless it has an observed effect during testing of a design prototype or of the hardware units affected. Less serious are random fabrication errors which affect occasional units in a production run. Defective units must be eliminated before delivery to a customer; but, for this to be done, the defect in each unit must be observed during hardware testing.

Other hardware errors may occur after bonding and packaging, in the form of short- or open-circuit defects; these latter are serious but not difficult to discover, because they are on the boundary of the IC and thus can be observed easily.

There also are "soft" errors caused temporarily by thermal or mechanical stress, unexpected external electric or magnetic fields, or exposure to ionizing radiation. An error is "soft" if it spontaneously recovers itself permanently, or vanishes, after a reset or other change of device state. An intermittent hardware failure at a specific gate does not represent a soft error but rather a hard defect which has degraded that gate. In this course, we shall not be concerned further with soft errors.

J. Williams, *Digital VLSI Design with Verilog*,

21.1.2 Assertions and Constraints

The first line of defense against hardware failure is good software. This means not just good design specifications, but also meticulous attention to warning messages from the simulator, synthesizer, static timing verifier, or other tool. In addition, good software means good design insight into possible problems, by use of assertions and well-chosen simulation unit-test vectors. The functional specifications must be fully validated before attempting to create a hardware representation. Also very important are the timing checks in the verilog or synthesis library which monitor fulfillment of the library-level design constraints during simulation.

We have discussed the software side already; now let us look more closely at the hardware.

21.1.3 Observability

This refers to the capability to measure functioning of logical transformations and data transfers in the hardware. A chip would be 100% observable if every internal state could be measured externally. Ignoring sequences of tests to be applied, this reduces to measurement of every internal storage device Q (flip-flop; latch; memory cell) and every input pin I in the chip. The number of inputs then may be written as N_I and the number of internal storage devices as N_Q. Each test measurement, in a digital device, amounts to one bit of information, which makes for a factor of 2 in the number of states; so, the total number N of possible states to test then must be,

$$N = 2^{\mathtt{N_Q}+\mathtt{N_I}}$$

Equipping a device with a simple thing such as an 8 kB cache adds as much as $2^{8*8^*1024}=$ about $10^{20,000}$ new states to observe. This factor then is multiplied by the number of states calculated without the cache. It is easy to see why testability usually is expressed in terms of the number of pins and registers (a logarithm), rather than number of states.

A small board-level design can be made highly observable by providing electrical test points on the traces on the board. A “bed of nails” automated tester then can bring its probes in contact with these test points, stimulate the board, and observe the effects. Any failure ideally can be detected and localized for manual rework, or discard, of a board found to be defective.

Such an approach is impractical for large digital IC’s, which have to be manufactured protected against electrical influences except through their I/O pads. Even probing the I/O pads without risking damage is mechanically difficult when they may number in the thousands for a modern ball-grid chip package.

Thus, observability has to be designed into the chip itself; it can not be left as an afterthought for someone to worry about after they receive the untested package or wafer in its final form.

21.1.4 Coverage

Coverage is a metric (statistic) which describes the quality of a sequence of test vectors during simulation or hardware testing. Coverage is related to observability. Given the total number of points observable under the test assumptions (I/O's only; or, all registers; or, whatever), coverage describes the fraction of them exercised by the given sequence of test vectors.

Of course, all points are observable in a software simulation. For coverage purposes, the test protocol may include the restriction that only I/O's will be monitored; this permits the software test vectors to be reused by a hardware tester. In such reuse, the simulated results would be compared with the hardware tester measurements in order to validate the hardware functionality.

The idea behind coverage metrics is to verify the absence of failed design pins, ports, or gates; failures of these usually can be described as "stuck-at" faults. If the test vectors can toggle every observable point, then coverage by those vectors is 100%. A ***fault simulator*** is a specialized logic simulator which checks for toggling of observable points.

In hardware testing, a sequence of test vectors may be more or less efficient at achieving a given coverage. Reducing ("collapsing") the number of vectors to achieve a given coverage is a desirable goal in testing, because it reduces the time required for the hardware tester to achieve the coverage. Time is money during manufacture.

Coverage can be used to describe a set of test vectors with reference to all points, not just those observable under the given assumptions; in this case, the maximum achievable hardware coverage usually will be less than 100%.

Be aware through all this, that ECC in the hardware can correct many defects which are missed in testing; however, this only works for stored data, not for control logic. The goal of testing is to detect and eliminate all defects possible.

Coverage Summary:

Coverage is a metric representing thoroughness of testing of observable states:

- Coverage is a measure on a set of test vectors.
- 100% coverage means all observable states have been tested.
- Low coverage means new or better vectors have to be used.
- Coverage usually is considered poor until it reaches 95% or more.

Coverage can refer to software (simulation) vectors:

- Represents lines of code or statements executed.
- Highly order-dependent. Non-Markovian.
- Independent of functional verification.

Coverage can refer to hardware vectors:

- Represents random defects checked.
- Generally only hardware stuck-at fault detection.
- Represents hardware functional verification.

21.1.5 Corner-Case vs. Exhaustive Testing

Hardware testing is a sophisticated field, and we shall touch on just some of its principles here.

Recalling the 8-kB cache computation above, it is completely unrealistic to assume 100% hardware coverage of an entire chip of any practical size. To test all combinations of states possible, looking for random defects and assuming a test vector of length 1024 bits, and a vector application rate of 1 GHz, the 8 kB cache example above would take about $10^{19,980}$ years to complete. This is from 10^7 seconds/year $= 10^{9+7}$ vectors/year $= 10^{9+7+3} = 10^{19}$ bits/year. The age of the known universe is only about 10^{10} years.

Happily, one can assume that not all states must be observed to be sure of the hardware functionality. For example, interaction between adjacent storage cells in a memory array on chip is very possible; so, a defect might be observed in one cell only when adjacent cells were in a certain state. However, such interaction is very unlikely among cells separated even by one other cell. For this reason, applying an alternating `'b010101...` vs. `'b101010...` pattern to a cell array (register) generally reveals any single-bit defect or defect dependent on any two adjacent cells. Combining this with a "walking 1" and "walking 0" test (shifting an isolated '1' and then shifting an isolated '0'), one can assume at a certain level of confidence that all states which would reveal a defect in a hardware register had been observed.

Let's look at the 8 kB cache example again, assuming the memory is organized (in at least two dimensions on the die, remember) so that only adjacent bits in a stored byte can interact. Then, alternating patterns as above, twice applied each, require 4 vectors per byte. Walking '1' and '0' require, say, 8 vectors each, for a total of 20 8-bit vectors to test one byte. Thus, 8 kB would require $8 \times 1024 \times 20$ 8-bit vectors, or a little over 10^6 bits of state. Assuming 1024-bit hardware test vectors, this is only 1.3 k vectors; at 1 MHz, time to test one cache memory this way would take only a millisecond or two. In practice, a millisecond easily would be available to test individual 8 kB cache memory dice on a wafer; so, more elaborate test vectors, for example vectors validating adjacent pairs of bytes, could be accommodated.

Corner case usually refers to the spatial or temporal boundary of a range of values. The latter cache test vector example above in a sense was a corner case, because it was intended to test adjacency, whereas, only a small fraction of all memory bits can be adjacent to those being tested at any one time.

More usually, corner case testing refers to selection of isolated values to test. For example, in a design including a verilog `for` loop that iterated through `i=0` to `i=127`, one would pay special attention to the values `0`, `1`, `126`, and `127` as the software corner cases of the loop. And, watch out for -1 and `128`!

Spatially, physical design problems should be sought especially on the boundaries of voltage islands or clock domains. And, in defining hardware test vectors, vectors of all-'`0`', or all-'`1`' should be applied as externally-defined corner cases.

Temporally, within the tested device, states immediately following chip-enable, chip-disable, read, write, and so forth would be given greater coverage than others. Look for something to go wrong every time you turn a corner!

21.1.6 Boundary Scan

Boundary scan provides for increased observability of the pin-outs of chips on a board; these are the *boundaries* of the chips. Rather than provide numerous direct electrical contact points for a bed-of-nails automated board tester, or for manual probing, dedicated traces on the board are connected to the chip I/O's; and, the board I/O's themselves are used to apply stimuli and observe results. This makes possible observation of inter-chip communication on wires or traces not normally routed to a board I/O.

For boundary scan, each scanned chip is equipped with a test port, the TAP (*T*est *A*ccess *P*ort), for controlling the scan. The test logic of the TAP controller may be very simple, as in the internal scan exercise we did in *Week 2 Class 1*, or it may include a device-specific, programmable state machine which automates some of the test mode shifting to save time or computation by the external hardware tester.

Very much the same as the JTAG internal scan port of *Week 2 Class 1*, the TAP has five pins defined: `TDI` (Test Data In), `TDO` (Test Data Out), `TCK` (Test Clock), `TMS` (Test Mode Select), and `TRST` (Test controller Reset). Except `TMS`, the TAP pins physically may be pins having other chip functions, such as control, address, or data pins, the test functionality being selected by asserting TMS.

A generic boundary-scan is shown in Fig. 21.1. In test mode, the scan latches (or flip-flops) are chained together, stimuli may be shifted in by `TCK`, and outputs may be shifted out. Each scan latch in a cell is connected to input and output muxes so it can be bypassed during normal chip operation. At any time during normal operation, `TMS` and `TCK` may be asserted to store the output states in the latches for subsequent shift-out and inspection.

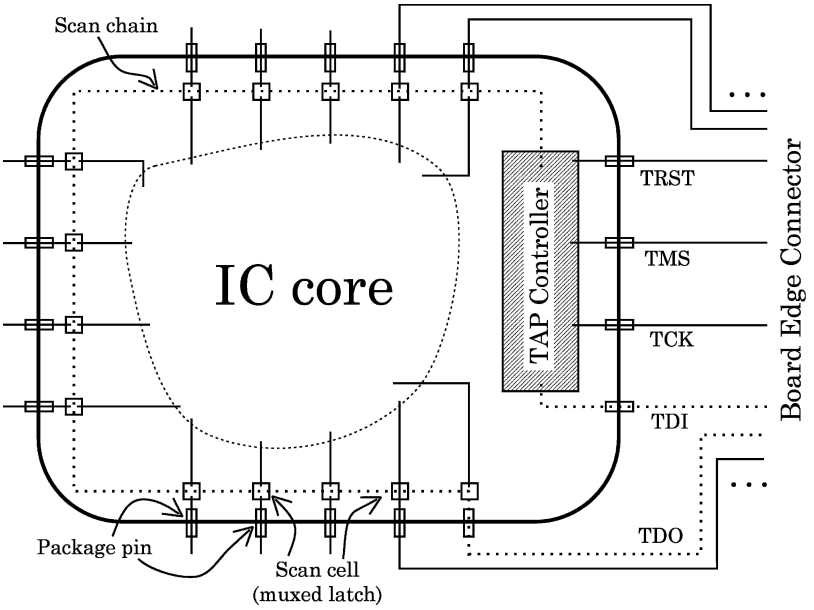


Fig. 21.1 Boundary scan logic on a board-mounted IC. Other IC's on the board are omitted, as are the TCK and TMS distributed to each of the scan cells. The IC is shown in test mode, with the scan chain the dotted line connecting the scan cells. Boundary scan makes points internal to the board, but not internal to the IC, observable

Boundary scan may be combined with internal scan or BIST (or both) in the same chip.

21.1.7 Internal Scan

We have presented the rationale for muxed flip-flop internal scan in *Week 2 Class 1*. Briefly, in internal scan, the registers in the IC are replaced by scan cells which can be configured as one or more chip-wide shift-registers in test mode. Several different ways of designing for internal scan are described in the readings recommended at the end of this chapter.

Sometimes the terms "full scan" and "internal scan" are used interchangeably. However, because it is possible to omit some registers (for example, an entire memory) from an internal-scan chain, internal scan does not imply full scan. Therefore, we prefer to use the two terms differently.

Full internal scan connects every register in the IC in the chain. Because inputs are observable already, this means that full internal scan in principle permits observability of all $2^{N_Q+N_I}$ possible internal states. This observability allows a level of confidence which may be required in certain life-critical systems, or systems in space or undersea vehicles, which can not be maintained or repaired during use. Such systems often are simple enough to make 100% coverage feasible.

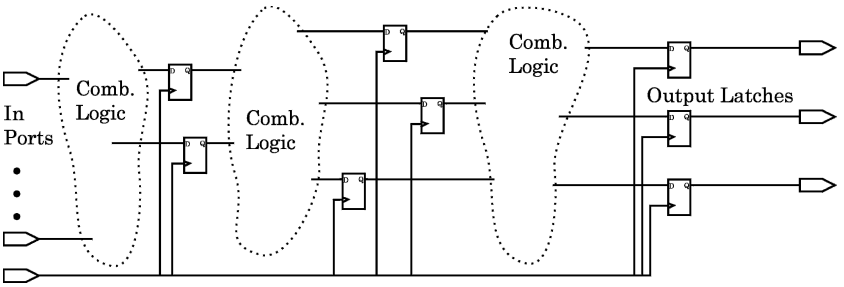


Fig. 21.2 A generic design without internal scan. Outputs are latched, and internal regions of combinational logic are separated by sequential elements. A clock is shown distributed to all the sequential logic

Internal scan does not include pad cells, if they are present in the scanned module. Internal scan logic inserted in the design of Fig. 21.2 is shown in Fig. 21.3.

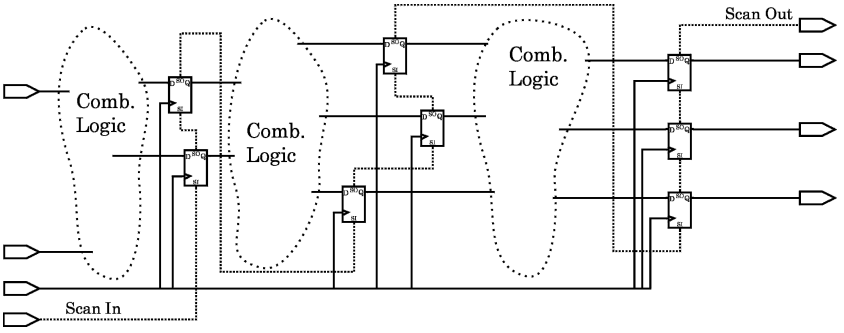


Fig. 21.3 The same design as above, after internal scan insertion. Every sequential element is replaced with a scan cell. The scan data chain is shown (each `SI` to `SO`). Scan clock and mode, as well as other controls, are omitted

Recall that almost all IC's will have outputs latched for synchronization reasons. Outputs, then, will be in the scan chain. Other logic may exist which is not observable; but, with full internal scan, this unobservable logic *can not* affect functionality of the chip and thus usually may be ignored for test purposes. Such logic may be present because of design errors or oversights, or because of design or production work-arounds. A famous example of the latter was the 386-SX microprocessor: Whenever a manufacturing defect was found in the onboard cache of a 386 die, the cache was disabled by tying off a pin, and the chip was packaged and sold at a lower price as a perfectly usable, cacheless, 386-SX. These 386 processors had thousands of nonfunctional gates.

Unobservable logic may affect operating parameters of a chip, because unobservable gates still may draw clock current and may leak the same way as functional gates, causing additional power dissipation merely because they exist on the chip.

It should be mentioned again that internal scan and boundary scan are not mutually exclusive and can be combined in the same chip.

21.1.8 BIST

*B*uilt-*I*n *S*elf-*T*est is an idea not restricted to IC design. Every PC has a ROM-based memory check executed automatically when it is turned on; likewise, when a hard disc is formatted at low level, the software, often in a disc-controller ROM, automatically verifies read and write access to every bit.

One advantage of BIST is that it requires no external test apparatus and only one control input, as well as one result output of some kind. In almost all cases, these additional I/O requirements can be met by assigning multiple functions to preexisting design pins. Even more important, any number of BIST-equipped chips can be tested concurrently during manufacture or operation. So, BIST allows design size and complexity to grow, and to be verified, with little additional hardware production testing time.

Thus, the cost of BIST in terms of chip I/O is negligible. However, the core silicon required for BIST may be substantial, because the self-test controller and program must be stored somewhere; also, usually the BIST must be executed upon the functional logic by means of additional, dedicated internal routing. If the design is complicated and includes many internal registers or modes of operation, the BIST must be very elaborate to generate vectors to achieve even minimal confidence that the hardware is fully functional; this implies significant design overhead. Furthermore, test results must be evaluated on-chip, which requires yet more storage and logic.

The process of BIST insertion is illustrated in Figs. 21.4 and 21.5.

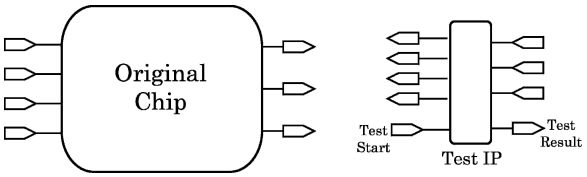


Fig. 21.4 A chip and a BIST test controller IP block

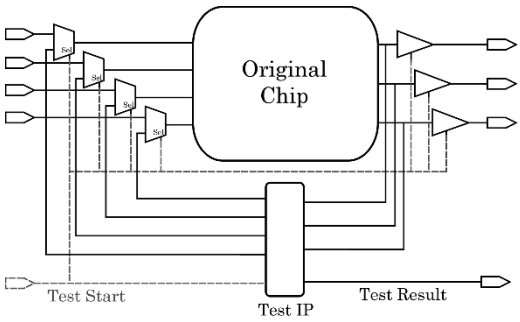


Fig. 21.5 Typical BIST insertion

Because of this, BIST is most efficient for IC's of large size and regular, repetitive structure – in other words, for memory IC's. If the memory is fabricated with spare cells or words, BIST can be used during manufacture to find defective cells and replace them with intact ones, increasing the production yield of the manufacturing process.

BIST usually depends on a standard TAP controller interface. BIST may be combined with scan logic. The BIST may accept a test-mode input, and, as a result put the chip into test mode, scan in preprogrammed patterns, scan out the latched states, and evaluate the result, terminating with a go/no-go signal to the rest of the IC or to the containing system. A random-logic state machine or a ROM may be involved. LFSR's may be used to generate pseudorandom test patterns efficiently.

BIST often is used in systems which require verification without direct access – for example, in the space program or in underseas devices. The currently active Mars rovers are equipped with elaborate BIST, and associated redundancy, for error recovery. The high radiation exposure requires special accommodation to soft errors caused by the cosmic rays against which Mars has no magnetic or atmospheric shielding.

In summary, DFT is a methodology which implies use of a certain collection of design techniques. The methodology includes attention to corner cases and test coverage of points of probable failure, both in software, during design, and in hardware, after implementation. The techniques encompass simulation, fault simulation, insertion of special hardware devices for internal or boundary scan, inclusion of built-in self-test apparatus, and use of I/O's to provide observability of internal functionality.

21.2 Scan and BIST Lab 25

Do these exercises in the `Lab25` directory.

Lab Procedure

Step 1. Internal scan exercise. Create a subdirectory named `IntScan` in the `Lab25` directory. If you have not yet done Step 9 of `Lab05` (*Week 2 Class 1*), copy in the purely-combinational design and do Step 9. Add the FF's, renaming the design to `Intro_TopFF`, insert the scan cells using the synthesizer, examine the result, and proceed to the next Step in this lab.

If you have done this Step already, just proceed to the next Step.

Step 2. Synthesizer boundary scan insertion. Boundary scan is intended for entire chips on a board; like BIST, it is inefficient for a small design. In this lab, we shall use the synthesizer to insert boundary scan in the original `Lab01` design just to see the result. We choose the `Lab01` design because automatic insertion has certain requirements which would be unnecessarily complicated to meet in our larger`SerDes` design.

A. Start by creating a new directory `BScan` in the `Lab25` directory. Copy in the original `Lab01 Intro_Top` design, and simulate it briefly with its `TestBench.v`, to verify connectivity. After this, copy in the verilog pad model wrapper file and the synthesis script file, both of which will be found named in the `Lab25` directory. If you have not recreated the symlinks, you will have to copy the files from the CD-ROM `misc` directory.

B. Add a test port. Automatic boundary scan requires a preexisting TAP, so add one new output, `ScanOut`, to `Intro_Top`, and four new inputs, `ScanIn`, `ScanMode`, `ScanClk`, and `ScanRst`. These are just the usual internal-scan JTAG port names for the TAP.

C. Instantiate pads. For the synthesizer to insert boundary scan, we must have pads in the design. We need three different kinds of pad: input, output, and three-state output. A three-state output pad is required for the TAP `TDO` (= `ScanOut`).

In a verilog design, pads are added inside the top-level module ports, as individual components within the top-level module; the module port names are signal names, not hardware pin contacts, in this context. Of course, a top-level wrapper module normally would be used; but, regardless, putting the pads inside the top design module permits the same testbench to be used for a design before and after pads have been added.

The TSMC pad library that matches our core library can not be used for synthesis, because the pads each are multipurpose and may be controlled by inputs to be input, output, or bidirectional. The models include switch-level elements and are too complicated both to be accurate in simulation and to be synthesizable.

For use in this lab, three pads have been selected from the pad library and provided with verilog wrapper modules for simulation, only. The wrappers may be seen in the verilog file, `tc`*padlibename*`_3PAD.v`, linked in the `Lab25` directory.

The names of the wrappers, only, should be used to instantiate pads for the `Lab25` boundary scan exercises. The names are: `PDC0204CDG_18_Out` for output pads, `PDC0204CDG_18_In` for input pads, and `PDC0204CDG_18_Tri` for the one three-state output pad. The "`0204`" represents a drive strength (`02` = 2 mA), and the "`18`" means 1.8V pad I/O (for 1.0V core logic). The port declarations for these library components are typical for verilog:

```
module PDC0204CDG_18_Out (output PAD, input I);
module PDC0204CDG_18_Tri (output PAD, input I, OEN);
module PDC0204CDG_18_In  (output C, input PAD);
```

The `OEN` output enable pin is asserted low in the library; however, the logic has been inverted to be asserted high for the three-state component in the verilog wrapper.

Remember that the ports of the `Intro_Top` module were named *X*, *Y*, and *Z* (outputs) and *A*, *B*, *C*, and *D* (inputs), and that each port pin in verilog is associated

with an implied wire of the same name. After declaring a few new, obviously-named *to-* and *from-* wires, the resulting pad structure for `Intro_Top` should look something like the following:

```
PDC0204CDG_18_Out Xpad1( .PAD(X),   .I(toX) ); // X is port; toX is wire.
PDC0204CDG_18_Out Ypad1( .PAD(Y),   .I(toY) );
PDC0204CDG_18_Out Zpad1( .PAD(Z),   .I(toZ) );
PDC0204CDG_18_In  padA1( .C(fromA), .PAD(A) );
PDC0204CDG_18_In  padB1( .C(fromB), .PAD(B) );
PDC0204CDG_18_In  padC1( .C(fromC), .PAD(C) );
PDC0204CDG_18_In  padD1( .C(fromD), .PAD(D) );
```

The TAP port has to connect to its pad cells. The core I/Os of the pads should be left dangling, so the synthesizer can determine how to connect to them:

```
PDC0204CDG_18_Tri TDOpad1( .PAD(ScanOut)/*, .I(),.OEN()*/ );
PDC0204CDG_18_In  padTMS1( /*.C(),*/  .PAD(ScanMode) );
PDC0204CDG_18_In  padTDI1( /*.C(),*/  .PAD(ScanIn)   );
PDC0204CDG_18_In  padTCK1( /*.C(),*/  .PAD(ScanClk)  );
PDC0204CDG_18_In  padTRST1( /*.C(),*/ .PAD(ScanRst)  );
```

The pad instances will be treated as any other instances during synthesis and netlist optimization. To protect the pad wiring from being modified during synthesis, add a comment directive to `Intro_Top.v` which flags all pad instances as `dont_touch`. You may use a wildcard, for example, "`*pad*`", for this.

The synthesis `.sct` script will be used to tell the synthesizer which ports (see Step 2B in this lab) we want it to use for the TAP.

D. Synthesize the boundary scan logic. After instantiating and wiring in the pads, you may use the `BScan.sct` synthesis script in the `Lab25` directory to insert the boundary scan cells and TAP controller. After compiling, write out the hierarchy and view it in `design_vision`. Read in the file, `Intro_TopNetlistBSD.v`. You may wish to back-track in the schematic from the `tdo` pad to the boundary scan tap controller, for example.

This exercise merely showed how to handle TAP ports; the controller has not been programmed, so this netlist can not be simulated.

The remainder of this lab is a memory BIST design.

Step 3. Design a BIST for our `DPMem1kx32` RAM.

A. Set up a working location. Create a new subdirectory named `BIST`, and copy into it your `DPMem1kx32.v` RAM model from the `Lab24 FIFO` directory. Name the new copy `Mem.v` and rename the module correspondingly.

B. Generate a benchmark synthesis result for later use; then set up a simulation testbench. This preparation will reacquaint you with the (already simulated) memory functionality and thus speed development and lessen the likelihood of design errors.

First, synthesize the renamed RAM of **A**, in `Mem.v`, sized for 32 words of 32 addressable bits each, optimizing for area only. As usual, impose simple logical-netlist design rules such as maximum fanout, etc. Make a copy of the synthesizer's area report in a text file for later reference. Also be sure you keep a copy of the synthesis constraints you used to obtain this area. ***Do not simulate this netlist*** – we are just interested in the approximate size of it, to compare sizes when the BIST is added.

For simulation of the verilog (source) design, instantiate the renamed RAM of **A** in a new `MemBIST_Tst.v` file, and create a small testbench there that exercises the RAM to demonstrate both write and read. Use a common clock for read and write. Set parameters to size the memory to be 32 addressable bits wide and 32 words deep. Simulate briefly, just to verify basic functionality. Your setup should be the same as in Fig. 21.6.

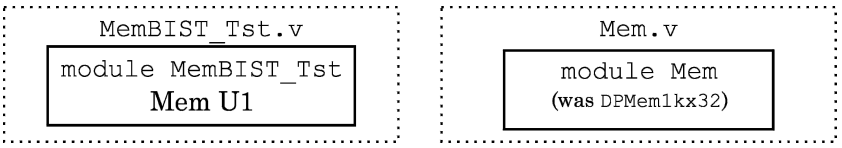


Fig. 21.6 Lab 25, Step 3B. The two verilog files are one testbench file and one memory model file. `Mem` is instantiated in `MemBIST_Tst`

C. Prepare a BIST plan which describes what the built-in self-test should do. Of course, it will be for random hardware defects, only. We shall assume that only isolated defects can occur; this is solely for lab convenience; but, even so, in a real production run, on this assumption, we would catch the majority of hardware fabrication defects.

We must be sure that our tests accommodate the memory parameters of width and depth, so we shall not assign numerical values to width or depth anywhere in the BIST module. Our memory is equipped with parity, so we should monitor parity throughout testing, in case of a single-bit failure during testing – even a soft one.

Here's what our plan for this lab says the BIST should do:

First test pattern: Validate the addressability. Write a different value to each address word; then, read the values out to verify that the memory hardware can store the expected number of different data. This will detect permanently shorted or open address lines, or obvious address decoder failures.

We shall write a value which counts from 0 to the max address, this count being replicated at several offsets in each word, and easily recognizable visually. An example of such a pattern sequence, counting from address 0 to address 31 (`5'h0` to `5'h1f`), would be,

```
32'he0e0_e0e0, 32'he1e1_e1e1, ..., 32'hffff_ffff
```

Second test pattern: Write '`1`' to every bit at all memory addresses, then read out and check the value. Repeat with '`0`'. This will find any bit or bits stuck independently at either level and missed by the previous counting test.

We shall stop with these two patterns, although more efficient, more thorough, or more exotic tests could be devised. For example, alternating checkerboard patterns on adjacent words might be written and read, or walking '1' tests run. Testing of a large RAM can be very elaborate, and thus there is an advantage to programming the test into the RAM chip, so that all RAMs in a manufacturing run, or installed in a system, might be tested at once, with minimal need for external test apparatus.

Step 4. Plan the BIST interface. We shall put the BIST logic in a separate module. The layout almost certainly will require that the memory core be a regular cell array in its own block; so, the BIST logic will have to be kept in a separate partition of the design.

Our BIST will have to address the memory by its address bus, and it will have to read and write on the memory data bus. The clocks can be shared. There will have to be a new input in the test-equipped memory for a test-start signal, and at least one new output to report test status. We shall keep these ports separate and not worry about sharing test functions on preexisting pins. During self-test, the Mem module must be unresponsive to external addressing or read or write requests, and it will have to disable its output drivers.

The best way to accomplish all this is to instantiate the Mem in a wrapper module which also instantiates a module containing the BIST logic. During self-test, the BIST-Mem system then can be isolated from the external world by the wrapper logic.

Step 5. Create the BIST wrapper: Create a new module named MemBIST_Top in a file named MemBIST_Top.v. by making a copy of Mem.v. Give the new MemBIST_Top module the same I/O's as Mem, except for one new input, DoSelfTest, and two new outputs, Testing, and TestOK.

After modifying the MemBIST_Top header for the new I/O's as just described, instantiate Mem in the MemBIST_Top module, and, declaring explicit wires for every I/O, directly connect all Mem I/O's to the corresponding MemBIST_Top ones using continuous assignment statements. Explicit wires in this exercise are important and will simplify interconnection of BIST and Mem later in the lab. There is no area or timing overhead for such wires, because the synthesizer will convert implicit wires to explicit ones anyway.

Your setup should be as in Fig. 21.7.

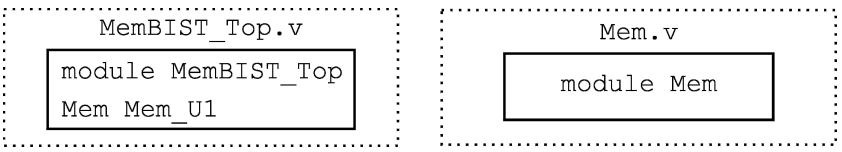


Fig. 21.7 Lab25 Step 5. The MemBIST_Top wrapper has almost the same module ports as Mem. Mem is shown instantiated inside MemBIST_Top

We are done with MemBIST_Tst.v of Step 3. After instantiating Mem in MemBIST_Top as above, and completing the wiring, rename MemBIST_Tst.v to MemBIST_TopTst.v, changing the testbench module name accordingly. Change the Mem instantiation to MemBIST_Top. Then, simulate the new two-module memory (MemBIST_Top containing Mem) briefly to check connectivity before going further.

Step 6. Define the BIST interface. Create a new BIST.v file for the built-in self-test module, BIST, by making a copy of MemBIST_Top.v.

But, before changing anything in the new file, instantiate BIST in the original MemBIST_Top.v by making a copy of the existing Mem instance and editing it as will be described. In this design, because Mem already has a complete interface to guide us, it will be easiest to connect a BIST instance first, and then to edit the copied module file in BIST.v to work out declarations of the required BIST I/O's.

The setup for this Step thus is as shown in Fig. 21.8, and the work will be to modify the BIST instance so we will know how to change BIST.v.

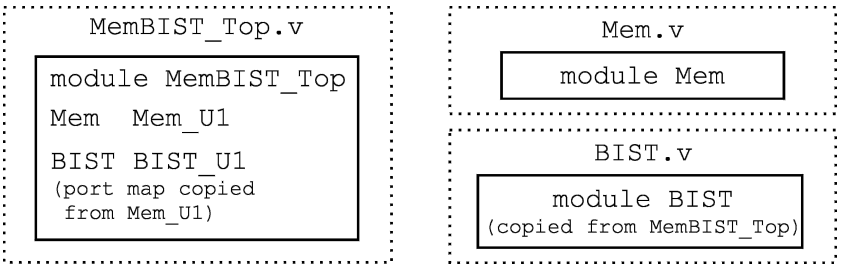


Fig. 21.8 Lab25 Step 6. Using wrapper MemBIST_Top to help determine the new BIST module ports

So, looking at the new BIST instance in MemBIST_Top.v:

First, we shall have to pass the AdrHi and DWid parameters to BIST; so, we should retain these exactly as copied from Mem.

We can eliminate Dready and ChipEna from the BIST I/O's.

Second, we require control of the Mem data input and output, so we should retain these I/O's in BIST. Soon, we shall multiplex these Mem busses so they can be directed either to the MemBIST_Top ports, or to the BIST ports. We shall connect the DataO output port of the Mem instance to the DataI input of the BIST instance, and vice-versa. So, we retain the DataO and DataI ports in the BIST instance.

Third, we should add a BIST address output bus, as well as read request (ReadCmd) and write request (WriteCmd) outputs. These also will have to be multiplexed with Mem inputs. Because none of our planned tests involves simultaneous

read and write, a single BIST address bus can be supplied both to the RAM read and write address input ports.

Fourth, we require a BIST clock input and a hardware Reset input. We shall require that testing be clocked by the RAM read clock (ClkR); so, this is the one we shall supply to BIST.

Finally, we should add the special BIST I/O's (the DoSelfTest input, and the Testing and TestOK outputs), all of which will be routed directly to the MemBIST_Top module ports. We also should monitor the RAM's ParityErr output using a corresponding BIST input during testing.

At this point, after the above described changes were made, the verilog code for your BIST instance in MemBIST_Top should look something like this:

```
wire ClkRw, Resetw, ...; // Assigned from MemBIST_Top module inputs.
...
BIST #( .AdrHi(AdrHi), .DWid(DWid) )
  BIST_U1
    ( .DataO(), .Addr(), .ReadCmd(), .WriteCmd() // outputs.
    , .Testing(), .TestOK()                      // outputs.
    , .DoSelfTest(), .ParityErr(), .DataI()      // inputs.
    , .Clk(ClkRw), .Reset(Resetw)                // inputs.
    );
```

Step 7. Implement the BIST controls in MemBIST_Top. After adding port names as above, complete the wiring in MemBIST_Top to BIST_U1, including the multiplexed data and other busses. You can do the muxes most easily as continuous assignments with conditional operators; you'll have to declare new wires for the BIST instance to do this. If, initially in Step 5 above, you used continuous assignment statements to wire the connections between the MemBIST_Top header and the Mem instance, only the new wire declarations and the conditional expressions will have to be added.

Briefly, when the DoSelfTest input goes high, the edge will initiate the test. Muxes in MemBIST_Top will be put in test mode by the assertion of Testing, which will be kept at '1' while self-test is in progress; TestOK will be asserted after a self-test if the test found no defect (otherwise, TestOK will stay at '0').

After this, go to the new file named BIST.v, which you created above, remove everything but the header and parameters from your new BIST module, change the port names and directions to match the code fragment above, and simulate MemBIST_Top briefly, using MemBIST_TopTst, just to check connectivity.

Step 8. (partly optional) Implement the BIST functionality. Before anything else, the test mode control should be defined. Here is one way to do this:

```
...
reg AllDoner // Flag completion of testing for internal BIST use.
  , Testingr;  // Sets test mode for the BIST.
//
assign Testing = Testingr; // Testing is a BIST output port.
//
always@(posedge Clk, posedge Reset)
  begin : TestSequencer
  if (Reset==1'b1)
       begin
       Testingr <= 1'b0;
       AllDoner <= 1'b0; // Normal level (noncommittal).
       ...
       end
  else // Must be a clock:
       begin : TestClocked
       if (DoSelfTest==1'b0)
            begin // Init, but leave TestOK alone:
            Testingr <= 1'b0;
            AllDoner <= 1'b0;
            ...
            end
       else Testingr <= 1'b1; // Entering test mode for this clock.
   ...     ...
```

All `reg` names end in '`r`'. A rising level on `Testingr` then can be used to start the self-test; the test routines should determine when the testing is finished.

A clocked `always` block can be used as a simple state machine to sequence the BIST through its tests. Our tests in Step 3 above address all of memory for write and then for read. It seems reasonable to implement each test as a separate `always` block containing an address counter and clocked on the `BIST` input `Clk`. The `always` for each test would be run selectively because of a flag set in the test-sequencing block. Actually, each `always` block used to write memory could be different from the one verifying the stored results. The test sequencer could check test status on every clock; when a test is complete, the current `always` block could set a flag reporting results and telling the test sequencer it is done.

Implementation of this BIST makes a very good ***optional project***; but, it is very time-consuming and probably is at least a day's work. Therefore, a complete implementation is provided for you in `BIST_Done.v` in the `Lab25/Lab25_Ans` directory.

To continue this lab, copy `BIST_Done.v` into your `BIST` directory. Copy your own `BIST.v` to a different name to save it. After looking through `BIST_Done.v`, copy or link it to `BIST.v` to replace your empty interface model. Simulate briefly to check connections.

If you want to spend some lab time on the `BIST` module, it might be interesting to rewrite part of the answer provided as a verilog *task*: The test sequencer code for phases 1 through 6 is very repetitive; this suggests replacing it with six calls to a task with a single number as input.

Simulate to validate your changes: A good simulation might exercise the memory a little; then, run a self-test; then, exercise it a little more. See Fig. 21.9.

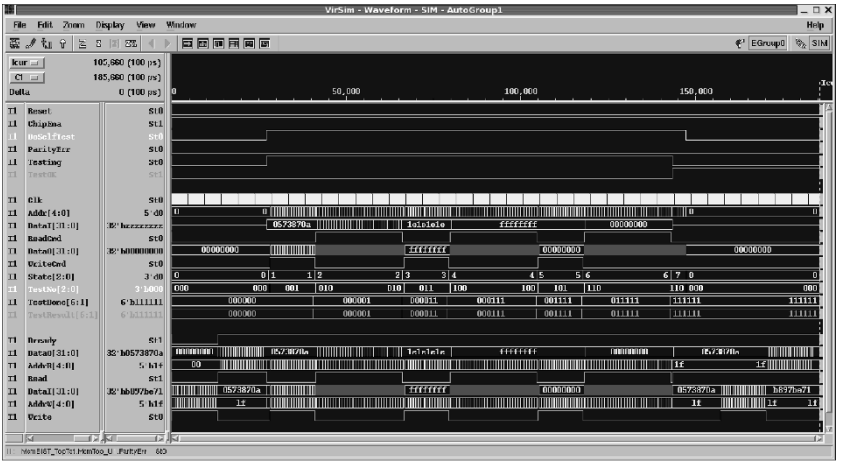


Fig. 21.9 Typical `MemBIST` verilog source simulation

Step 9. Synthesize the completed `MemBIST_Top` design, optimizing it for area with the same constraints as you used in **Step 3 B**. The synthesis will take about 10 minutes with mild constraints. With a quick netlist, don't bother with simulation; the netlist may not simulate because of lack of clock constraints, but its size will be about right. Compare the size of the design with and without the `BIST`.

Optional: After comparing areas, you may add clock definitions and a maximum output delay constraint to your synthesis script, constrain to fix hold violations, and resynthesize (see the answer directory for specific values). This netlist will take a half-hour or more to synthesize, but the result, as in Fig. 21.10, will simulate correctly.

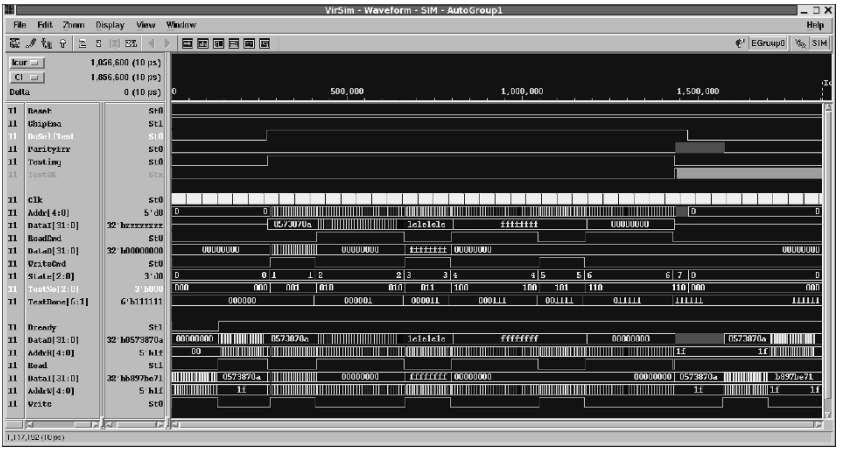


Fig. 21.10 Simulation of the synthesized `MemBIST` verilog netlist, using the same testbench as for the source

21.2.1 Lab postmortem

How big was the `BIST` netlist, compared with the one for our bare `DPMem1kx32` RAM?

Would you expect the use or arrangement of the BIST `tasks` to make any difference in synthesis?

21.3 DFT for a Full-Duplex SerDes

We'll finish up today with a lab on test insertion for our `SerDes`. First, we'll modify our `Lab24 SerDes` design to have full-duplex functionality, as would a *PCI Express* lane. Then we'll add test logic.

21.3.1 Full-Duplex SerDes

A full-duplex lane is just a dual serial line, with one serdes sending in one direction, and a second serdes sending in the other. The two serial lines being independent, this lane can send and receive simultaneously in both directions.

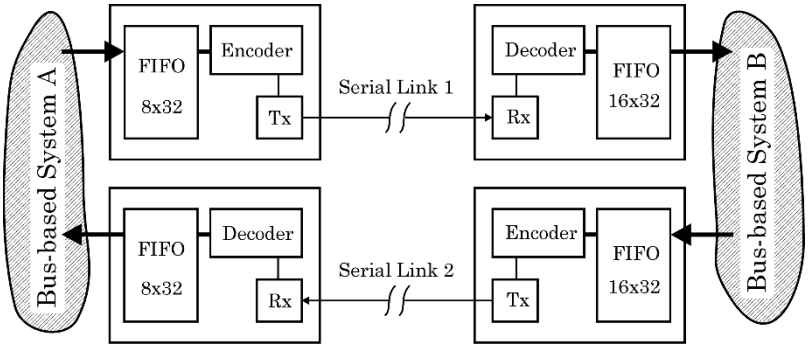


Fig. 21.11 Two instances of the `Lab24 SerDes` assembled into a full-duplex lane. A simulator testbench can represent the two communicating systems, *A* and *B*. The FIFO depths differ for the two systems

For variety, we shall assume that system *A* can provide a FIFO only with 8 words, but that system *B* can provide one 16 words deep. See Fig. 21.11. Such a difference would not be unusual, assuming that our duplex serial lane was between different chips on a board. For example, the communicating chips might be from different suppliers, in different technologies, or fabricated on availability of different kinds of IP.

It is important to understand the organization of the full-duplex lane proposed. Each SerDes device model can be used to span the opposite sending and receiving ends of a single serial line; or, each SerDes could be interpreted as referring to one end (send and receive) of a full-duplex lane. The difference is illustrated in the top and bottom of Fig. 21.12.

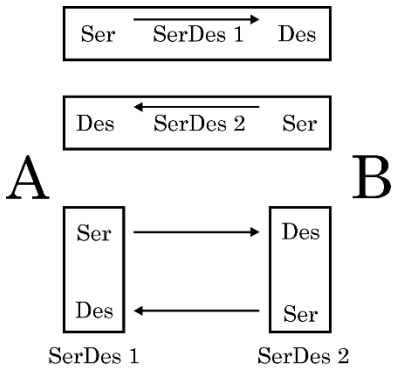


Fig. 21.12 Two ways of using a pair of `SerDes` devices between two systems, *A* and *B*

Our choice is the upper one shown; it permits immediate instantiation of our `SerDes` design. We assume that our two serdes will reside in a system (maybe on a single chip) which will floorplan each Ser some distance away from its Des, so that a serial data transfer between *A* and *B* would be useful.

21.3.2 Adding Test Logic

Before proceeding to the lab, some thought might be given to the following points:

- How should we equip our serdes for testability?
- How good is observability without test logic?
- Where should we add assertions?
- How much internal scan?
- Would boundary scan be useful?

21.4 Tested SerDes Lab 26

Work in a new subdirectory of the `Lab26` directory for each of these exercises. The instructions will say how to name the subdirectories.

Lab Procedure

Step 1. Gather the parts of the full-duplex serdes. Create a directory named `FullDup` in `Lab26`, and create in it a subdirectory named `SerDes`. Make a

copy of the entire contents of the SerDes directory from Lab24/Lab24_Ans/Lab24_Step08 in it. For now, use the provided answer design; there will be opportunity later to return to your own SerDes implementation, if you should want to do so.

What we'll do now is to create place-holder files for our full-duplex serdes by modifying the files for the Lab24 unidirectional SerDes.

Leave SerDes.v in the new SerDes directory, but move the other files up one directory, into the new FullDup directory. These files should include: SerDes.inc, SerDesFormats.inc, SerDes.vcs, and SerDesTst.v. The only things remaining in the SerDes directory should be SerDes.v and the four subdirectories originally there.

Rename the moved design files by changing "SerDes" in their names to "FullDup". For example, you would have in the FullDup directory, FullDup.inc, and FullDupTst.v, among others. Don't change the contents of these files yet.

Step 2. Use a testbench copy as a template for FullDup.v. Copy the file you just renamed to FullDupTst.v, to a new file named FullDup.v in the FullDup directory.

Open FullDup.v in a text editor and delete everything in it except the old SerDes instance, with its parameter map.

Copy-and-paste a second, identical copy of this instance into FullDup.v. Name the upper instance in the file SerDes_U1 and the lower instance SerDes_U2.

Then, open the SerDes.v file (in the SerDes subdirectory), and copy-and paste the SerDes module header declarations into the top of FullDup.v. Change the module name to "FullDup".

FullDup.v, which once was a copy of FullDupTst.v, now should consist of the module port declarations from SerDes.v, and two instances of SerDes each with different instance names. The top module name in FullDup.v should be FullDup.

Next, we shall edit FullDup.v to make minor changes in the SerDes part of the design; then, we shall complete the full-duplex design of FullDup.

Step 3. Resize the SerDes FIFOs. It would be fun arbitrarily to vary both widths and depths of the FIFOs for the two systems, *A* and *B*, shown in Fig. 21.11. However, the frame encoding would have to be altered to change the width, so we shall leave all widths at 32 bits (33 including parity), and just modify the depths. Also, we have used exact counter wrap-arounds in the FIFO state machine, so each FIFO must have a depth in words equal to a power of two.

As shown in Fig. 21.11, instead of 32 words, each *A* FIFO should include 8 words, and each *B* FIFO 16 words. We need separate depth parameters for the *A* and *B* sides of each SerDes.

To accomplish this in the FullDup.v file, remove the old AWid FIFO depth parameter and replace it in the FullDup header with four new ones, one for each FIFO instance in the full duplex design. The result should look something like this:

```
module FullDup #(parameter DWid = 32
                 , RxLogDepthA = 3, TxLogDepthA = 3 // 3 -> 8 words.
                 , RxLogDepthB = 4, TxLogDepthB = 4 // 4 -> 16 words.
                 )
... (port declarations) ...
```

The new parameters should be passed to the `SerDes` instances; but, first we have to decide which instance in the verilog corresponds to which one in Fig. 21.11 above. Let's take the first `SerDes` instance in the file, which we have named `SerDes_U1`, as the top serial line in Fig. 21.11 ("Serial Link 1").

Then, postponing port-mapping details, the parameters should be passed this way:

```
...
SerDes #( .DWid(DWid)
        , .RxLogDepth(RxLogDepthB)
        , .TxLogDepth(TxLogDepthA)
        )
SerDes_U1 ( .ParDataOutRcvr ... // A sender; B receiver.
  ...
SerDes #( .DWid(DWid)
        , .RxLogDepth(RxLogDepthA)
        , .TxLogDepth(TxLogDepthB)
        )
SerDes_U2 ( .ParDataOutRcvr ... // B sender; A receiver.
 ...
```

At this point, the `SerDes` module can not use the new parameters, so we must modify it to accept them.

A `SerDes` module in our `FullDup` design gets two FIFO depth parameter declarations, one for the receiving (Deserializer) FIFO, and one for the sending (Serializer) FIFO; of course, at the `SerDes` level, there is no distinction between '*A*' and '*B*'. In the `SerDes` subdirectory, in `SerDes.v`, pass in the new parameters, and find every occurrence of `AWid` and rename it to `RxLogDepth` or `TxLogDepth` appropriately. Do not rename the `FIFO` parameter, just change the value mapped to it. The `Serializer .AWid` gets the `Tx` parameter; the `Deserializer .AWid` gets the `Rx` parameter. Assign 5 (32 words) to be the module header default for both depth values; this will be overridden to 3 (8 words) or 4 (16 words) during instantiation.

There is no need for further editing of parameters lower in the design; the `Serializer` and `Deserializer` will pass the proper values to their submodules as they did before.

In particular, each `FIFO_Top` instance now properly will pass on the depth values required to define FIFO addressing and number of memory words. Because

of our previous planning, there is no reason to rename the parameter declared in the FIFO_Top module or to change anything in the FIFO or PLL parts of the design.

Step 4. Complete the connection of the full-duplex lane. Back again in FullDup.v, we have to decide what should be our module I/O's, and how they should connect to the SerDes instances.

The SerDes instances are independent; there should not be any communication between them except over the serial line, and this simplifies our decisions. What we shall do next, is decide what to call the nets connecting to the SerDes instances, and what should be the FullDup module I/O's. We shall try to winnow away all but the minimum required FullDup I/O's.

Let's start with a simplification of the system relationships: SerDes_U1 (link #1 in Fig. 21.11) transmits data from *A* to *B*; SerDes_U2 transmits from *B* to *A*. Therefore, on the *A* side, the only SerDes_U1 serializer output port would be the serial line (SerLineXfer); the *A* serializer inputs would be ParDataIn, InParClk, InParValid, Reset, and TxRequest;. On the SerDes_U1 *B* side, the *B* deserializer, the outputs would be ParOutRxClk (in the *B* clock domain) and ParOutTxClk (in the *A* clock domain). The *B* deserializer inputs would be OutParClk, Reset, and RxRequest.

The opposite I/O relations must hold for SerDes_U2.

That takes care of the instance pins. As for the nets, to avoid elementary errors or confusion, we shall start by renaming all of them. Initially, we'll prepend 'A' or 'B'; later, when decisions have been made, we shall rename the nets by moving the prepended letters to the end of the net name, following the example above of "RxLogDepthA", etc.

So, let's begin by adding an 'A' prefix to every SerDes_U1 and SerDes_U2 *A* net, and a 'B' prefix on every other net connected to the SerDes instances. For example, in SerDes_U1, we would have AInParStim, AInParClkStim, *etc*. At this point, the only net lacking an 'A' or 'B' would be ResetStim, which, as shown above, we shall share between *A* and *B*.

After this, we can go up to the module port declarations and make a *complete, duplicate copy of the current FullDup module ports*. Do this by simple copy-and-paste of the entire module header port declarations, just doubling the original number of FullDup I/O's.

Once the FullDup module ports are duplicated, doubling the number of I/O's in the FullDup header, prepend an 'A' to every net name in one copy and a 'B' to every net name in the other.

We could declare a few new nets and stop here, but let's simplify things a bit:

Simplification **A**. Eliminate one FullDup module input port by declaring just one common Reset in the module header to be routed (later) to both SerDes instances.

Simplification **B**. We want to transfer data between *A* and *B*, which are bussed systems; we don't really require the serial lines to be visible outside of FullDup.

So, declare wires for the serial lines named SerLine1 and SerLine2, and substitute them for the nets on the SerLineXfer pins of SerDes_U1 and SerDes_U2, respectively. These output nets will remain otherwise unused, and this allows us to delete the two corresponding module output ports in the header of FullDup.

Simplification **C**. We may assume that *A* and *B* correspond to distinct clock domains. Therefore, the *A* clock for clocking in data to SerDes_U1 should be the same clock as the *A* clock used to clock out SerDes_U2 data. Declare a single module input named ClockA, and connect it to SerDes_U1.InParClk and SerDes_U2.OutParClk. Then, declare ClockB, connect it correspondingly, and delete the module input ports for all other clocks. This eliminates a total of another two FullDup module ports.

Simplification **D**. It was suggested optionally at the end of Lab24 to create two parallel-bus output ports from the SerDes; data from one would be clocked out by the receiving-domain's clock, and data from the other would be clocked out using the parallel clock embedded in the serial framing. These ports were named ParDataOutTxClk and ParDataOutRxClk. The purpose was to see the way the data delivery differed because of the different clock domains.

Thus, you may have two parallel output ports on each SerDes. We don't want these two ports in FullDup; we only want outputs clocked in the receiving domain. So, if you have them, delete both ParDataOutTxClk ports, even if they were commented out in your Lab24 SerDes instance. This leaves just one parallel-data output port in each SerDes instance and allows us to delete as many as two more FullDup module output ports.

Simplification **E**. Let's look at the RxRequest and TxRequest inputs. They were convenient for debugging our SerDes and demonstrating FIFO functionality; but, now they can be reduced. However, to preserve flexibility in case our reduction plan doesn't work out, let's keep the port connections and declare four *request* wires to assign them explicitly; this will be shown in a code example below. So, we'll tie RxRequest high permanently; we'll control everything with the input ParValid signals at both ends: If ParValid is asserted in either system, it also will assert TxRequest in that system. This allows us to remove four FullDup ports and simplifies the SerDes operational protocol.

Simplification **F**. Wrap-up. Except ClockA, ClockB, and Reset, rename the FullDup module ports so each remaining input begins with "In" and each output with "Out". Also, except Reset, rename all ports so that every *A* port ends with 'A', and every *B* port with 'B'. This moves the 'A' and 'B' prefixes to the end of each name, to become postfixes; the whole renaming sequence was an accounting technique to help keep the port renaming free of confusion or errors.

After all this, your FullDup.v should contain something close to the code shown below:

```
`include "FullDup.inc" // timescale & period delays.
//
module FullDup
     #(parameter DWid = 32                    // 32 bits wide.
     , RxLogDepthA = 3, TxLogDepthA = 3       // 3 ->  8 words deep.
     , RxLogDepthB = 4, TxLogDepthB = 4       // 4 -> 16 words deep.
     )
     ( output[DWid-1:0] OutParDataA, OutParDataB
     , input[DWid-1:0]  InParDataA,  InParDataB
     , input InParValidA, InParValidB
     , ClockA, ClockB, Reset
     );
//
wire SerLine1, SerLine2
   , RxRequestA, RxRequestB, TxRequestA, TxRequestB;
//
assign RxRequestA = 1'b1;
assign RxRequestB = 1'b1;
assign TxRequestA = InParValidA;
assign TxRequestB = InParValidB;
//
SerDes #( .DWid(DWid)
        , .RxLogDepth(RxLogDepthB)
        , .TxLogDepth(TxLogDepthA)
        )
SerDes_U1 // Ports reordered:
   ( .ParOutRxClk(OutParDataB), .SerLineXfer(SerLine1)
   , .RxRequest(RxRequestB), .ParDataIn(InParDataA)
   , .InParValid(InParValidA), .TxRequest(TxRequestA)
   , .InParClk(ClockA), .OutParClk(ClockB), .Reset(Reset)
   );
(similarly for SerDes_U2; but, with SerLine2, and with 'A' & 'B' suffices reversed)
```

Simulate briefly to verify connectivity, and file names and locations. Remember that by default some simulators will attempt to open `` `include `` files on a path relative to their invocation context.

You may reuse your `SerDesTst.v` testbench (which was renamed to `FullDupTst.v` in Step 1) for this after changing names and adding the new FIFO parameters. Or, to expedite the testbench, which should be quite elaborate for this design, you might consider just copying the `FullDupTst.v` testbench file provided for you in the `Lab26_Ans/FullDup_Step4` directory.

Use the simulator to check the hierarchy tree to be sure that both `SerDes` instances, and all `FIFO` instances, are present. Make sure all warnings about bus widths, assignment widths, and so forth are corrected or well-understood before proceeding. Fig. 21.13 and 21.14 show some `FullDup` simulation results.

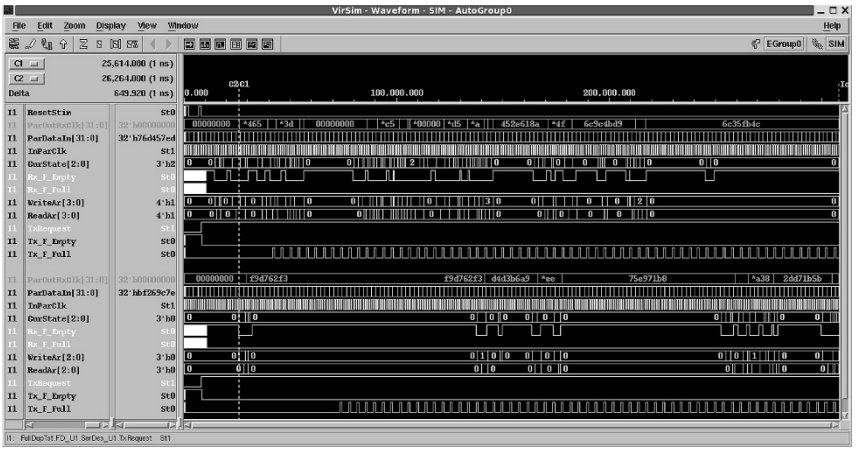


Fig. 21.13 The first full-duplex serdes (`FullDup`) simulation. Serdes `U1` waves are on top; `U2` below. Data are random and different in each direction; the *A* and *B* FIFO's are of different depth

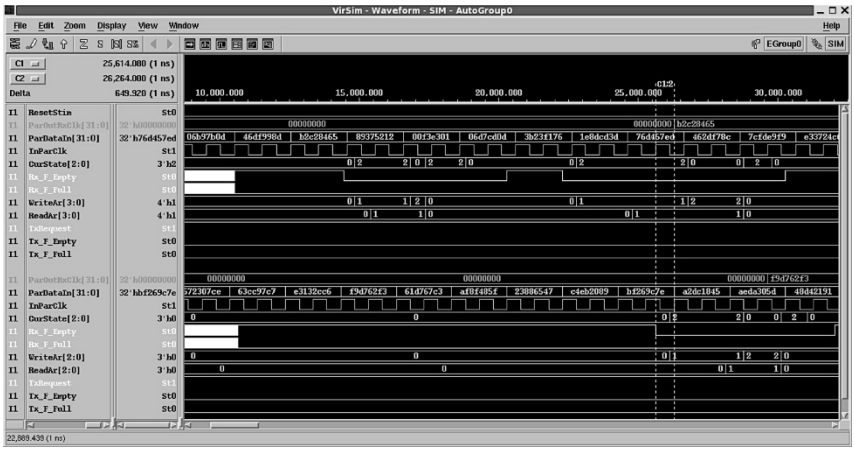


Fig. 21.14 Close-up showing clock skew between the two, independent parallel-bus domains

Step 5. Add assertions. Create a new subdirectory `FullDupChk` in the `Lab26` directory and copy everything in `FullDup` into it. Do this Step in the new directory.

We've ignored assertions and timing checks in our serdes design, except for one parity check assertion in the FIFO memory model. Let's correct this deficiency now.

No FIFO is visible at the `FullDup` level of the design, but we still would like to know when the FIFO's go full or empty. The first appearance of the full and empty flags in the design occurs at the `SerDes` level, in the `Serializer` and `Deserializer` instance port mappings.

Add four assertions to `SerDes`, two in `Serializer` and two in `Deserializer` , that check that the four FIFO flags are false. Each assertion

should print a *warning* message to the simulator console. The assertions may be your own, or they may be modelled after our generic assertion in *Week 4 Class 2* (Lab 11). Use `%m` so that the assertion will print the module instance name in which it was triggered.

Our design will not fail under these warning conditions, but they imply that possibly the containing *A* and *B* systems will have to resend lost data, the loss being determined from the data rather than from `FullDup` hardware. We could use these assertion warnings to decide whether the design word depth of the `FullDup` FIFO's should be changed.

Simulate. There should be at least a few FIFO-empty states to exercise your new assertions.

If you have not done so already, add testbench code to enable transmission (`Tx`) from both `Serializers`; your testbench also should make `ClockA` and `ClockB` independent. You might consider copying the testbench provided in `Lab26_Ans/FullDup_Step4`, to save time in lab. This testbench is fairly sophisticated and includes independent clock-frequency drift in both domains, as well as different random input data for each serdes.

Step 6. Add timing checks to monitor the `DesDecoder` control pulse widths. Do this Step in `FullDupChk`. Using a simulation testbench from `Lab26_Ans`, or one very similar, display the signals in a `DesDecoder`. There is only one `SerDes DesDecoder` module, so both instances will implement your checks. Our simple approach to control in this module has allowed `ParValid`, `ParClk`, `doParSync`, and `SyncOK` to be asserted apparently as pulses of wildly varying width.

Let us study these pulse widths. We wish to determine the narrowest positive pulse that occurs during simulation. To do this, add `$width` timing checks in `DesDecoder` to report a violation whenever the width of a positive pulse on one of these signals is less than the value given by a `specparam`. Declare a different `specparam` for each timing check.

Vary the `specparam` values while repeatedly running simulation. For example, after setting up four `specparams` and four `$width` timing checks, set all the `specparams` but one to 0; set the other one to 100 (=100 ns); this will disable all but one. Then, run the `FullDup` simulation for a relatively long time, say 500k to 1M ns.

If there is no violation reported, increase the nonzero criterion `specparam` value by 50 or 100 ns, or more, and repeat the simulation until violations occur (you will know a violation message when you see one). Use the simulator `[Stop]` button if violations are numerous. Going by the message numbers or guesses, then decrease the criterion and hunt until you have found the maximum criterion value at which no violation occurs. Leave the `specparam` there.

Repeat the procedure for the other three `specparams`, separately or simultaneously, until all are at their maximum no-violation level.

Now, any unusual state causing a shorter pulse will be revealed by these checks. Record your timing-check widths in the table below.

Variable	Maximum no-violation width (ns)
ParValid	
ParClk	
doParSync	
SyncOK	

It pays to be able to find unexpected brief pulses. As the old naval shipyard saying goes,

"Loose glitch sinks chips".

Step 7. For this Step, change out of `FullDupChk` and return to the `FullDup` directory. Make up a synthesis script file, `FullDup.sct`, using as a template, for example, a script in one of the subdirectories at `Lab24/Lab24_Ans`.

With this script, synthesize the design while imposing only zero-area and just one timing constraint: Set a stringent maximum delay limit on all outputs, one which the synthesizer can meet only with a little slack. Do not define a clock. Flatten the design before running `compile` with all defaults (no option); this will make the resulting netlist comparable with the scan-inserted netlist we shall create next. Be sure to log an area report into a text file for later reference.

The purpose of this Step is just to prove quickly that the design verilog is entirely in synthesizable format. Without any clocking, clearly it should not be expected to simulate correctly. However, with clocks and other timing constrained, and with hold violations fixed, this `FullDup` design can be made to synthesize to a correctly simulating netlist. See the CD-ROM answers for various ways of applying workable synthesis constraints. For curiosity's sake, Fig. 21.15 and 21.16 give waveforms for a fully-constrained, ***completed class serdes project***:

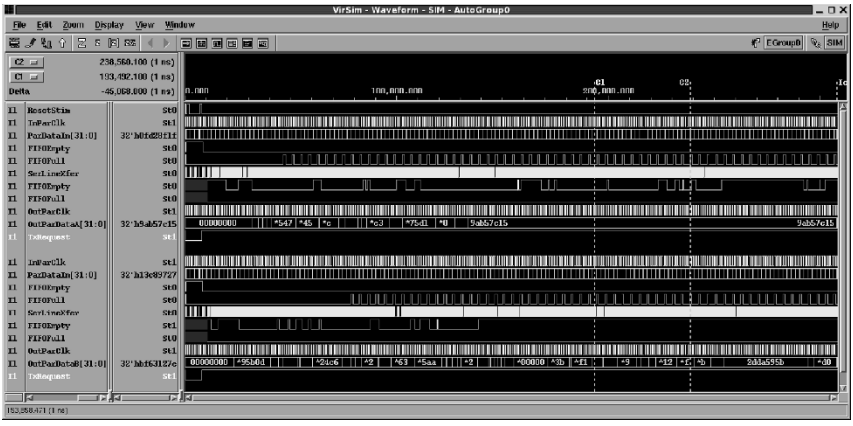


Fig. 21.15 Netlist simulation of the completed full-duplex serdes, synthesized with hold-fixing for all clocks

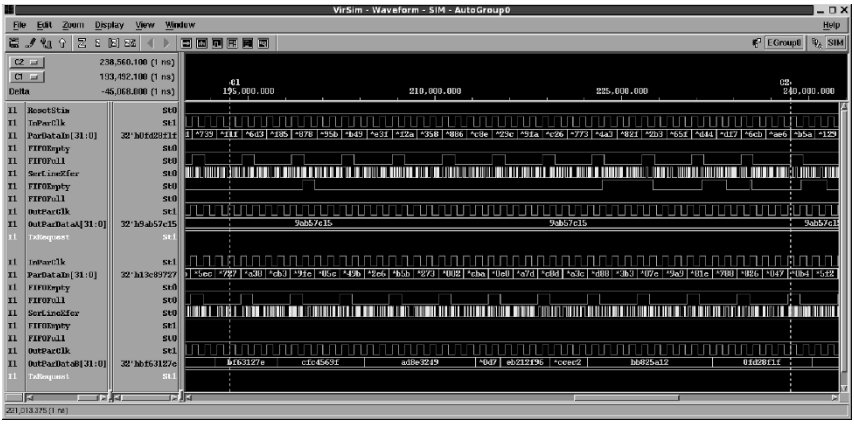


Fig. 21.16 Close-up of the completed full-duplex serdes netlist simulation, demonstrating that the *A* domain input parallel word `0x0fd2_8f1f` (time ∼193 ms) is transferred correctly to the *B* domain output bus (time ∼235 ms)

Step 8. Insert scan logic. Create a new subdirectory named `FullDupScan` and copy the entire contents of the `FullDup` directory into it. Simulate briefly at the new location to verify file locations and connectivity. Do the rest of this lab in `FullDupScan`.

Use the synthesizer to insert internal scan as follows:

A. First, rename your copied synthesis script to something like `FullDup Scan.sct`; edit it to change its output file names (netlist and log) so they will be identifiable as scanned outputs.

In the script file, delete the existing compile command and substitute in its place the scan-insertion commands used in `Lab25`. These may be found at `Lab25_Ans/IntScan/DC_Scanned/Intro_TopFFScan.sct`. You will have to change the `Clk` and `Clr` names and the scan clock period. You also will have to edit the `FullDup` module header in `FullDup.v` to add two new input ports, `tms` and `tdi`, and one output port, `tdo`, for the scan circuitry.

B. Run the synthesizer to insert scan elements. This will take only a little longer than did plain synthesis. Be sure to log an area report in a separate text file so you can compare it with the one in the previous Step.

The same constraints should not be difficult to meet with scan inserted, because previous compilation should have shown they could be met without scan. This might have been guessed by recalling that scan insertion replaces sequential elements with scan elements, and that the extra gate count from (effectively) one mux per flip-flop really can't amount to very much. Without dwelling on library issues, inspection of the library we are using for synthesis indeed shows that the area of a typical scan flip-flop is only about 30% greater than that of an ordinary flip-flop. This exercise primarily was to show the effect of scan on netlist size. Do not bother with simulation at this point; the TAP controller will not be functional because the synthesis design constraints did not include a clock, preventing creation of a working scan test mode.

C. Browse through the scan-inserted netlist file in a text editor to see where the scan elements were substituted.

21.4.1 Lab Postmortem

Where in `FullDup` might other timing checks be added?

What was the area impact of scan insertion in this design?

How hard would it be to provide a hardware test engineer with a key to the scanned design? In other words, define a `FullDup` scan test vector to be a vector the length of the entire `FullDupScan` scan chain. How wide would this vector be? Then, how would one go about providing a key giving the location of each design bit in the test vector?

21.4.2 Additional Study

Read *The NASA ASIC Guide: Assuring ASICs for SPACE*, Chapter Three: Design for Test (DFT). At: `http://klabs.org/DEI/References/design_guidelines/content/guides/nasa_asic_guide/Sect.3.3.html` (updated 2003-12-30).

If you want a little more detail on LFSR pseudorandom number generation, try the Koeter article, *What's an LFSR?*, at `http://focus.ti.com/lit/an/scta036a/scta036a.pdf` (2007-01-29).

(Optional) Retrieve your `BIST.v` empty interface file of `Lab25`, Step 8, and complete the BIST functionality your own way.

Chapter 22
Week 11 Class 2

22.1 SDF Back-Annotation

Today, we'll look into SDF just enough to understand how it works.

22.1.1 Back-Annotation

Back-annotation refers to the modification of a netlist with newly assigned or updated attributes, usually delay times. A tool is run; and, the result goes *back* to the netlist; whence, *back*-annotation. The netlist itself is not modified; the back-annotations instead are stored in a side file.

Although the synthesizer can estimate the effect of trace length on delay time while optimizing a netlist, these delays initially are in the synthesis library's wire-load model, and they do not appear as such in the synthesized netlist. The netlist merely contains component instances with certain timing and drive strengths chosen during optimization to be adequate to meet the timing constraints, given the expected trace lengths. These component choices are fairly inaccurate when compared with actual performance after placement and routing. A good estimate of many of the layed-out delays based solely on the initially synthesized netlist might be off by 10%; a typical estimate would be off by more.

After a netlist has been used to implement a floorplan or placed-and-routed layout, accurate delay times can be associated with the placement of individual component instances. The verilog library propagation delays are ignored and back-annotated delays are used instead. This means that a back-annotation side file should contain a description of the timing of every component instance in the netlist.

22.1.2 SDF Files in Verilog Design Flow

The standard file format for netlist delay back-annotation is called SDF (Standard Delay Format). It is described in IEEE Std 1497; its use with a verilog netlist is described in section 16 of the verilog language standard, IEEE Std 1364. Various

J. Williams, *Digital VLSI Design with Verilog*,

other back-annotation file formats, for example SPEF (Standard Parasitic Exchange Format), are used for a variety of other physical layout purposes not included in the present course. SDF contains delay times; SPEF contains capacitances. In any case, the flow is as in Fig. 22.1.

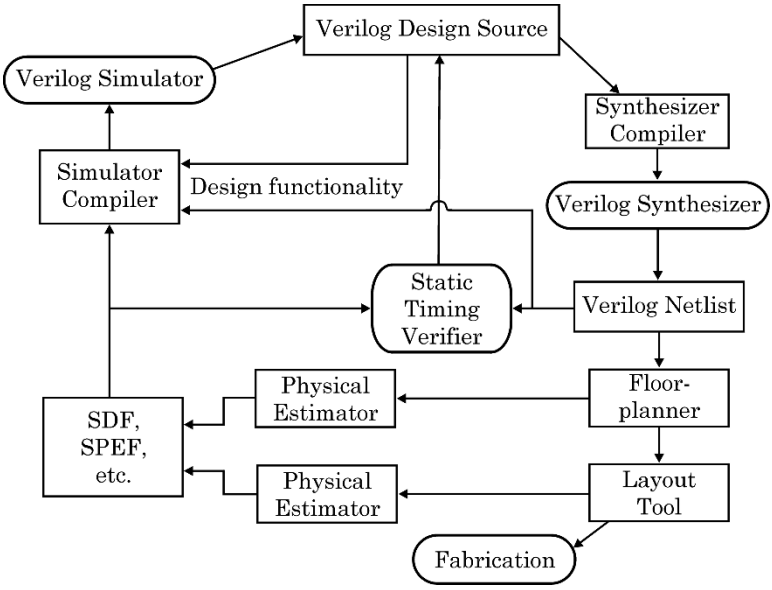


Fig. 22.1 Back-annotation in a typical design flow. The estimated delays or physical parameters are kept in a side file associated by tools with the design netlist

Timing in an SDF or other back-annotation file will override verilog library `specify` block path delays, including conditional delays. The net (interconnect) delays attributable to trace length also can be back-annotated in SDF, although in the verilog, such delays generally would be lumped to component outputs. Verilog path delays are associated with the `IOPATH` keyword in SDF; net delays are associated with the `INTERCONNECT` keyword. The keyword `CELL` is used for any component or module instance. Except for colons in timing triplets, the only SDF punctuation is by parentheses (‘ (‘and’) ’); so, the language looks a bit like lisp or EDIF at first glance.

In an SDF file, the SDF keywords are written in upper case and are not case-sensitive; references to netlist objects are copied literally from the netlist and are case-sensitive.

22.1.3 Verilog Simulation Back-Annotation

This process can be very simple and straight-forward. A verilog netlist is created somehow, for example by synthesis, and an SDF file is obtained somehow for that netlist. Typically, after the netlist has been floorplanned, an SDF file may be written

out by the flooplanning tool. Another way to obtain an SDF file is simply to write one out from our synthesizer after saving the netlist in verilog format.

Once a verilog netlist and corresponding SDF file have been obtained, the verilog system task, `$sdf_annotate` ("*sdf_file_name*"), is inserted into the netlist at whichever level of hierarchy the SDF file was written. When the simulator then is invoked, those modules which were back-annotated (usually, the whole design) will be simulated using the SDF timing instead of the verilog component library timing.

Complexity can arise if timing is to be back-annotated from several SDF files. If the files are nonoverlapping, the `ABSOLUTE` keyword (which appears everywhere in our SDF files in the lab below) may be used to make the last file read overwrite a common delay with its final value. It is possible to use an `INCREMENT` keyword instead; in that case, subsequent delay values are added to previous ones.

In summary:

SDF (Standard Delay Format) is defined in IEEE Std 1497. It is

- A netlist representation of timing triplets.
- Written by the synthesizer, static timing tool, or layout tool.
- Hierarchical; an SDF file may be rooted at any verilog module level.
- Associated with a design netlist by the following verilog system task in a design module `initial` block:
  `$sdf_annotate` ("*file_name*`.sdf`");

22.2 SDF Lab 27

Do this work in the `Lab27` directory.

Lab Procedure

Step 1. Simulate a design. Create three new subdirectories, `orig`, `ba1`, and `ba2`, in `Lab27`. Change to the `orig` directory and copy in the verilog files for our old friend, the `Intro_Top` design from `Lab01`. Use the original files provided, including the original `TestBench.v` file.

In `TestBench.v`, comment out the `` `include `` of `Extras.inc`, invoke the simulator, and simulate the design. Do not exit; but, rather, leave up the waveform window displaying the simulation of the top-level primary inputs and outputs.

If you are using VCS and save your VCS configuration to a file, this file may be used to invoke VCS for the rest of this lab with the same exact window sizes and waveforms displayed.

NOTE: The simulator process ***must be left active***; otherwise, the wave window will not resize or redraw if moved. The easiest way to guarantee this is to do each step in this lab in a new terminal window.

Step 2. Simulate a netlist. After simulation in Step 1, replace your copied `Lab01 Intro_Top.sct` with the modified one provided in your `Lab27` directory. The design rules in the `Lab27` version have been changed so that the gate count in the netlist is minimized. You may wish to use a text editor to compare the original `Intro_Top.sct` with the new one to see how this was done.

After copying in the new `Intro_Top.sct`, synthesize the design with it. The new script will write out the synthesized netlist and an SDF file automatically. The SDF file will be based solely on the synthesis library's wireload model, but its timing will be far more accurate than that of the original verilog design.

Make up a new simulator file list containing `TestBench.v`, the netlist file just created, and the *LibraryName*`_v2001.v` verilog component library file provided. Simulate the netlist and compare the waveforms with those of the original design in Step 1.

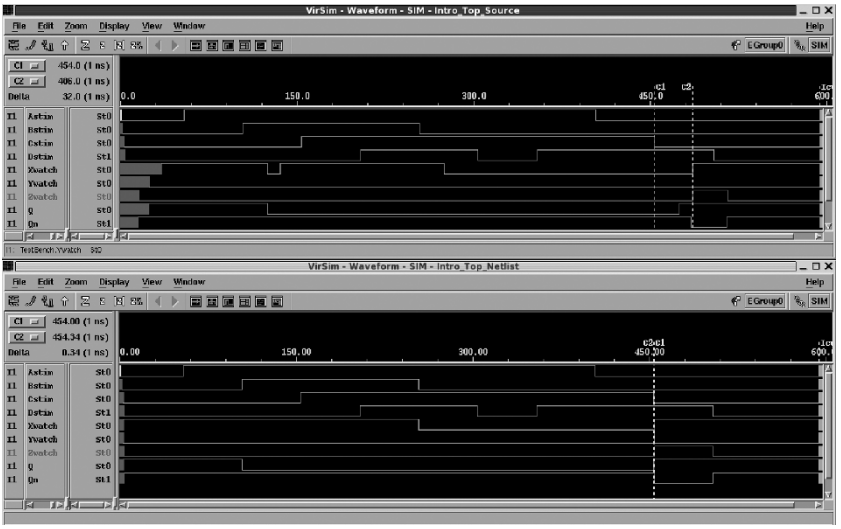


Fig. 22.2 Timing differences between the verilog source (above) and its synthesized netlist

As shown in Fig. 22.2, the netlist-simulation wave shapes should be somewhat different from those of the source verilog, because the actual library gate delays are very short (approximated in the *LibraryName*`_v2001.v` file) compared with the delays in the source verilog. Most noticeably, a glitch in the `Xwatch` output wave will be missing in the netlist simulation.

After examining the simulation timing, leave up the netlist simulation waveform window, but close the wave window, or exit the simulation session, of the original design (Step 1).

Step 3. Simulate the back-annotated netlist. Copy `TestBench.v`, the netlist file, the simulator file list, and the SDF file of Step 2 into the `ba1` directory you created above.

Open the netlist file in a text editor and find the module `Intro_Top`. Add an initial block in that module to reference the SDF file. For example,

```
initial $sdf_annotate(``Intro_TopNetlist.sdf'');
```

Simulate again, and compare the SDF back-annotated waveforms with those of the original netlist of Step 2. All things considered, the shapes should be the same, and the timing of the outputs should be close to the same. There will be small timing differences (see Fig. 22.3), because the original netlist used timing from the verilog component library, *LibraryName*`_v2001.v`, which was hand-created with approximate timing, whereas the back-annotated netlist uses more accurate SDF delays created by the synthesizer from the original *LibraryName*`.lib` (compiled by Synopsys as *LibraryName*`.db`) database.

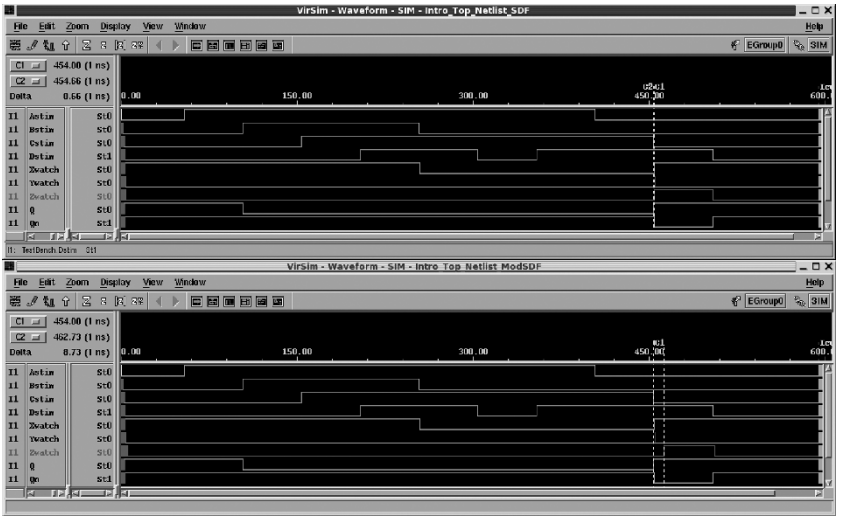


Fig. 22.3 Netlist timing for an approximated verilog library (above) and synthesizer SDF back-annotation

When you have examined the delays, close the simulator waveform display window from Step 2 but leave up the display from this Step.

Step 4. Modify the SDF timing and simulate to see the difference. Copy all files from `ba1` of Step 3 to the `ba2` directory you created previously.

Let's modify the timing of the `Intro_Top.Z` output port to see how this might be done by back-annotation by a floorplanner or layout editor. This is just for instructional purposes: A designer *never* would edit manually an SDF file under normal conditions.

Open the verilog netlist file in a text editor, find the `Intro_Top` module, and locate the driver of the `Z` output port; this probably will be an inverter gate of some kind. Just find the last driver cell of the `Z` output and note the component type *ctype* and the instance name *inst_name* of that directly-connected driver cell.

Then, open the SDF file in a text editor and look for this statement,

```
(CELL
  (CELLTYPE ``ctype'')
  (INSTANCE inst_name)
```

Below the instance name, there should be a `DELAY` and `IOPATH` statement. It is the `IOPATH` statement which determines the delay on that path in *inst_name* when the netlist is simulated with the SDF back-annotation. The first triplet is rise delay; the second is fall. The correct instance will be at the top of the design represented in the SDF file, and it will not be preceded with a hierarchical path.

You might like to look in the *LibraryName*`_v2001.v` file to see how the hand-entered verilog delay differs from the synthesizer's delay for this component. This difference will account for our simulator display differences. However, in general, an SDF delay will be instance-specific; whereas the library delay can be only component (module) specific. The actual database used by the synthesizer to calculate the SDF delays can be seen in *LibraryName*`.lib`, in your Synopsys library installation directory area.

Anyway, to see how the SDF file controls timing, edit the `IOPATH` rise delay for the `Z` output to be about 20 times the original value, and the fall delay to be about 10 times. You may wish to comment out the original line; use a verilog "`//`" to do this. Then, save the file and repeat the simulation. The effect of the new delays should be fairly obvious: As visible in Fig. 22.4, the final `Z` positive pulse in the simulation should be displaced to a greater time and should become narrower than it was with the unaltered SDF back-annotation.

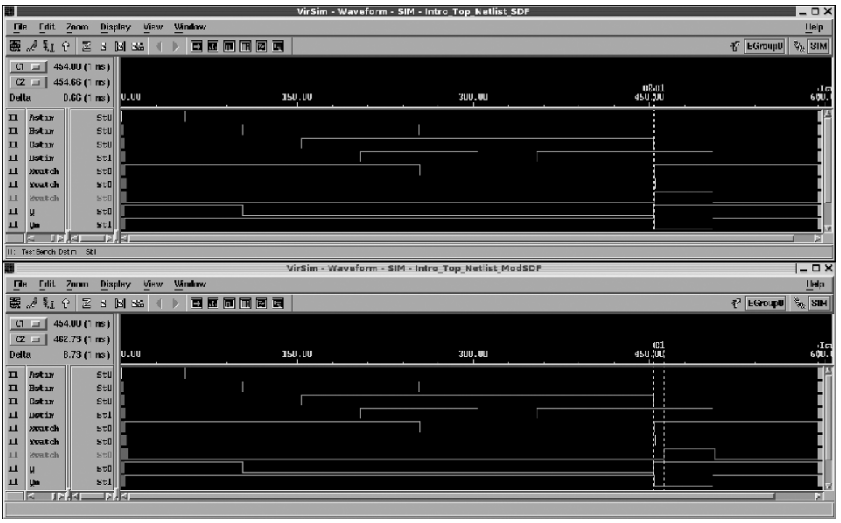


Fig. 22.4 Back-annotated netlist timing: Original SDF (above) vs. hand-modified SDF

Optional. You may consider repeating Step 4 but entering different min vs. typ vs. max values in the SDF file, and then invoking the simulator to use any one of these sets of delays.

22.2.1 Lab Postmortem

What if the `$sdf_annotate` task call is located in the wrong module?

How might just part of a design netlist be back-annotated?

22.2.2 Additional Study

Optional. Recall Step 2 of today's lab. Explain in detail why a brief transition to 0 on the top-level `X` output occurred during simulation of the original design but not for the original synthesized netlist.

Chapter 23
Week 12 Class 1

23.1 Wrap-up: The Verilog Language

23.1.1 Verilog-1995 vs. 2001 (or 2005) Differences

We have continuously exercised one major difference between *verilog-1995* and standards later than *verilog-2001*: The ANSI-*C* module header declaration format. Some other differences new in 2001 were: Implementation of `generate`, multi-dimensional arrays (*verilog-1995* allowed only 1 dimension), the `signed` arithmetic keyword (only `integers` or `reals` were signed in *verilog-1995*), and the `automatic` keyword for recursion in tasks or functions. In addition, there was standardization of the use of SDF, and there were VCD dump file enhancements. There were a host of other, perhaps less important changes, all explained in the Sutherland paper referenced in today's *Additional Study*.

23.1.2 Verilog Synthesizable Subset Review

By now, all this may be well understood. However, here's a brief review:

The synthesizer can not synthesize:

- delay expressions,
- initializations (`initial` blocks),
- timing checks,
- verilog system tasks or functions;

or, generally, anything else which is transient in, or depends on, simulation time. The component delays used by the synthesizer for timing optimization and timing constraint violations come from its own library models and not from `specify` blocks or other verilog constructs.

The synthesizer may reject, or issue a severe warning about, procedural blocks containing even one nonblocking assignment with a delay. However, it will

J. Williams, *Digital VLSI Design with Verilog*,

synthesize correctly any sequence either of blocking or nonblocking assignments which are not associated with delays. Delays in the verilog simply are ignored.

The synthesizer will synthesize `generate` blocks, looping or conditional.

The synthesizer will enforce certain good coding practices by rejecting assignments to the same variable from different `always` blocks, and by rejecting mixed blocking and nonblocking assignments to the same variable or in the same `always` block.

The synthesizer has great difficulty synthesizing correctly any construct which is equivalent functionally to a complicated latch – which is to say, to a complicated level-sensitive or transparent latch. To avoid latch problems, (a) do not omit variables from an `always` block with a change-sensitive event control list; and (b) do not omit logic states from cases assigned in a change-sensitive block, unless the code represents a simple latch with just one output bit.

23.1.3 Constructs Not Exercised in this Course

We have indeed covered the entire verilog language, including some constructs not useful in VLSI design; but, we have omitted a few details which differ unimportantly from ones covered, or which are not frequently implemented in any tool. Here is a complete list:

- All the verilog language ***keywords*** are listed in Appendix B of IEEE Std 1364. Thomas and Moorby (2002) section 8.5, also contains a table of them, a miscellaneous few of which we have ignored in the present work.
- File input/output manipulations. These include `$readmemb` and `$readmemh`. Note that the correct verilog filesystem name divider in Windows is '`/`', not '\'!

  Thomas and Moorby (2002) sections F.4 and F.8 present the available file-related system functions. There also is some relevant discussion of file I/O in Bhasker (2005), section 3.7.4.
- System tasks and functions. For the most part, consistent with the synthesis orientation of the course, we have called attention to these constructs only when necessary. The many builtin tasks and functions we have ignored are reasonably self-explanatory, given our practice in the few we have used. Some good explanation may be found in Bhasker (2005), section 10.3. See a list of them below.
- Many compiler directives (`` `*directive* ``). These also are quite easily understood, for the most part, from their names. See a list of them below.
- Attributes. These are Boolean expressions in a scopeless block delimited by the two tokens, "`(*`" and "`*)`", the same tokens as used in Pascal for comments. For example, `(* dont_touch U1.nand002 *)`. Verilog attributes are tool-specific and are meant to control synthesis or simulation behavior. Current tools use comment directives ("`//synopsys ...`") for this purpose and generally will not recognize verilog attributes.

- `defparam`, `force-release`, `assign-deassign`. As has been explained, these constructs should be avoided in modelling code, although `force-release` sometimes might be useful for special testbench functionality or for debugging.
- Declaration of module `output` ports as **`reg`**. This is permitted in verilog-2001, as a compatibility carryover from verilog-1995. It is not recommended as a design practice, because internal `wires` no longer can be connected to such ports (bringing back the error-proneness of the 1995 module header format), and module delays on such ports are difficult to make visible. Also, the presence of mixed `wire` and `reg` ports requires perpetual attention to mutually exclusive port assignments (`assign` vs. `always`), negating any small savings of time because of omission of internal `reg` declarations paired with continuous assignments.
- The verilog PLI, which is discussed below.

Closely related, omitted topics which are not part of verilog:

- Many synthesizer constraints have not been exercised. The command reference manual for the Synopsys DC synthesizer contains hundreds of commands, thousands of options, and exceeds 2,000 printed pages.
- The Synopsys DC Shell ***dcsh*** constraint syntax of `dc_shell` or `design_vision` command mode. This is very similar to Tcl syntax, but its use is deprecated now by Synopsys.
- The Synopsys Design Constraint language (SDC), which is an adaptation of Tcl and which is licensed free by Synopsys. SDC is meant to provide scripting and constraint portability among tools such as synthesizers, static timing verifiers, power estimators, coverage estimators, and formal verifiers. It can be used with VHDL or any other language such as System Verilog or SystemC, if the tool will accept it.

23.1.4 List of all Verilog System Tasks and Functions

These are documented in the IEEE 1364 Std document, section 17. They fall into ten categories; a complete list is given in the next table. The functionality usually is indicated by the name, but the Std or other help documentation should be consulted for full explanation. Many are intended solely for testbench use.

No tool set known to the author has implemented all the system tasks and functions; however, all tools known to the author have implemented the subset presented in this course. The ones we have exercised are in bold in the table below.

In choosing a new task or function for some special purpose, it is advisable to verify that all tools involved have implemented that choice correctly (or at least in exactly the same way).

Type	Task and Function Names
Display	**$display** $displayb $displayh $displayo **$monitor** $monitorb $monitorh $monitoro $monitoroff **$strobe** $strobeb $strobeh $strobeo $write $writeb $writeh $writeo $monitoron
Time	**$time** $stime $realtime
Sim. Control	**$finish $stop**
File I/O	$fclose $fdisplay $fdisplayb $fdisplayh $fdisplayo $fgetc $fflush $fgets $fmonitor $fmonitorb $fmonitorh $fmonitoro **$readmemb** $swrite $swriteo $sformat $fscanf $fread $fseek $fopen $fstrobe $fstrobeb $fstrobeh $fstrobeo $ungetc $ferror $rewind $fwrite $fwriteb $fwriteh $fwriteo **$readmemh** $swriteb $swriteh **$sdf_annotate** $ssacf $ftell
Conversion	$bitstoreal $itor $signed $realtobits $rtoi $unsigned
Time Scale	$printtimescale $timeformat
Command Line	$test$plusargs $value$plusargs
Math	$clog2 $ln $log10 $exp $sqrt $pow $floor $ceil $sin $cos $tan $asin $acos $atan $atan2 $hypot $sinh $cosh $tanh $asinh $acosh $atanh
Probabilistic	$dist_chi_square $dist_exponential $dist_poisson $dist_uniform $dist_erlang $dist_normal $dist_t **$random**
Queue Control	$q_initialize $q_remove $q_exam $q_add $q_full
VCD	$dumpfile $dumpvars $dumpoff $dumpon $dumpports $dumpportsoff $dumpportson $dumpall $dumpportsall $dumplimit $dumpportslimit $dumpflush $dumpportsflush $comment $end $date $enddefinitions $scope $timescale $upscope $var $version $vcdclose
PLA Modelling	$async$and$array $async$nand$array $async$or$array $async$nor$array $sync$and$array $sync$nand$array $sync$or$array $sync$nor$array $async$and$plane $async$nand$plane $async$aor$plane $async$nor$plane $sync$and$plane $sync$nand$plane $sync$or$plane $sync$nor$plane

23.1.5 List of all Verilog Compiler Directives

A complete list follows below. No tool known to the author has implemented all these directives, but many tools which have not implemented a directive are likely to ignore it or just issue a warning. As suggested before, if feasible, use `` `undef `` at the end of any file in which a macro name has been `` `defined ``.

```
`begin_keywords `celldefine  `default_nettype `define `else
`elsif `endcelldefine  `endif `end_keywords  `ifdef `ifndef
`include `line `nounconnected_drive `pragma `resetall`timescale
`unconnected_drive  `undef.
```

23.1.6 Verilog PLI

PLI stands for "*P*rogramming *L*anguage *I*nterface". The PLI strictly speaking is not part of the verilog language, so we have not discussed it previously in this course. However, the PLI is specified in the same IEEE standard document (Std 1364) which specifies the verilog language.

The PLI language is *C*, and the PLI is a library of routines callable from *C*. The PLI is a feature of the verilog simulator which allows users to design and run their own system tasks and even to create new verilog-related applications to be run by the simulator. Examples of such applications would be fault-simulators, timing calculators, or netlist report-generators. Like the builtin system tasks such as `$display`, the user-defined ones also must be named beginning with '`$`'.

The PLI generally is not especially useful for a designer, but it may be valuable to someone creating tools for a design team. Another use might be to implement certain simulator features not available in the current release by a particular tool vendor.

Examining the messages printed by the VCS simulator when it compiles a new module, one can see that it simply is a specialized compiler compiling and linking an executable *C* program. The VCS GUI is a different, precompiled program which optionally creates an interprocess communications link with this executable when the latter runs; the executable's I/O then is formatted and displayed by the GUI. To save run time and memory, VCS can be invoked in a text mode, with no GUI; the VCS help menu shows how. QuestaSim, Silos, Aldec and other verilog simulators work the same way as VCS, although the GUI and the simulator may be more or less tightly coupled.

In the past, the verilog simulation executable was *interpreted* (like a BASIC program or shell script) rather than being *compiled*. Except for running slower than a compiled executable (but being modifiable faster), the interpreted version worked essentially the same way as the compiled version.

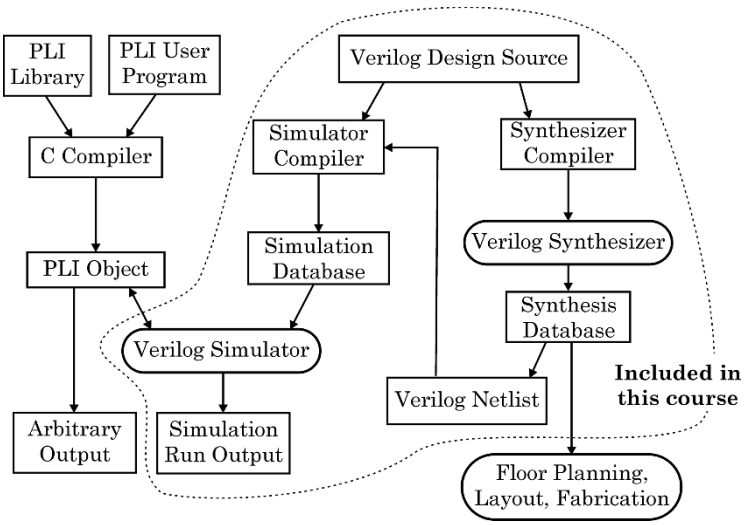


Fig. 23.1 Sketch of PLI relationship to simulation and synthesis flows

The PLI operates on the simulator itself (actually, on its internal data structures and operational runtime code), as shown in Fig. 23.1. Two independently accessible databases representing the verilog design are the verilog design source code, which may be accessed by shell scripts or by visual inspection in a text editor, and the synthesized netlist, which, if written out in verilog format, also may be accessible by shell script or text editor. The simulator internal data structures generally are in a proprietary binary format; but, if a design is simulatable, the internal data must consist of objects representing the design structure, component models (including primitives), nets, and attributes of the corresponding verilog objects. Busses and ports must be represented independently for every bit, and, for netlist simulation, all loops must be represented unrolled. Over and above the objects in the design, the simulator database includes state, delay, and simulation time information.

As of the 2005 Std, the PLI consists solely of the VPI or ***verilog procedural interface*** routines. Past versions of the PLI included two other collections of routines, TF or ***task-function*** routines and ACC or ***access*** routines, which now are absorbed into the VPI. All three collections are overviewed in chapter 13 of Palnitkar (2003), and we won't go into their properties further here.

23.2 Continued Lab Work (Lab 23 or later)

23.2.1 Additional Study

On the enhancements introduced in *verilog-2001*, read Stuart Sutherland's excellent summary, "The IEEE Verilog 1364-2001 Standard: What's New, and Why You

Need it". Based on an *HDLCon 2000* presentation. `http://www.sutherland-hdl.com/ papers/2000-HDLCon-paper_Verilog-2000.pdf` (2005-02-03).

Read the summary of *verilog-2001* enhancements in Thomas and Moorby (2002) Preface pp. xvii–xx.

Optional Readings in Palnitkar (2003)

For file I/O, see sections 9.5.1 and 9.5.5 and the Palnitkar CD example `Memory.v`.

Read section 14.6 on coding for logic synthesis.

Read through all of chapter 14 on logic synthesis.

Read chapter 13 on the PLI.

Chapter 24
Week 12 Class 2

24.1 Deep-Submicron Problems and Verification

24.1.1 Deep Submicron Design Problems

We restrict this discussion to electrical factors of interest to a verilog designer. Deep submicron effects related strictly to fabrication are ignored (ion implantation; details of optical proximity correction; shift to shorter, ultraviolet mask exposure wavelengths; immersion lithography; etc.).

The term "deep submicron" was adopted in the late 1990s to refer to layout pitches below about 250 nm; or, below about 1/4 of a micron. At this scale, trace delays begin to overtake gate delays as primary considerations in design timing. For our purposes here, we include ***nanoscale*** technology in this deep-submicron category (Fig. 24.1).

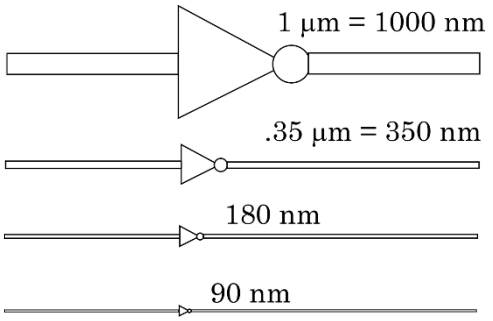


Fig. 24.1 Linear proportional gate schematic sizes and trace widths. Routing represented across a chip region of constant width

Some generalities about the deep-submicron problem: Call the linear pitch size *L*. The major issue is that trace delays along distances *L* begin to contribute significantly to total delays, and anything introducing variation in trace timing becomes a major problem:

- The rate of electric charge transfer to or from a transistor is about proportional to its area, or L^2. But the delay on a trace is more or less proportional to its RC time constant. The resistance R increases linearly as trace width decreases, and capacitance C to ground decreases about the same way, so the time constant of a trace because of this reactivity remains about the same as pitch shrinks. To charge to a given voltage, capacitance C of a trace decreases linearly as one power of L. Thus, the time to charge a trace of a given length increases about proportionally to the one remaining power of L. Individual trace lengths become important for delay calculation.
- As gate sizes shrink, distances on chip begin to increase *relative to gate size* (see Fig. 24.1). So, delays begin to grow because distances connecting gate pins do not shrink as rapidly as L, which is proportional to the distance between traces.
- As a greater and greater number N of gates is fabricated on a given chip area, the number of possible connections (traces) grows as something between $2N$ and N^2. Thus, on the average, as L shrinks (and chip complexity grows), the final fanout required of the average gate grows. For this reason, too, loading caused by traces begins to dominate timing estimation on the average.

In addition to propagation delay timing problems, pitches at and below 90 nm have introduced new problems related to fabrication-layer current leakage and noise. In particular, shrinking distance between traces has increased capacitance between them and therefore has increased the average cross-coupled noise.

Larger designs, lower supply voltages, and smaller pitches imply that libraries must contain a greater variety of different gates, including scan cells. Characterization must be more elaborate because of the smaller potentials and closer spacing of the gates on chip.

Library simulation and synthesis models can not simply store gate delays; instead, delays have to be calculated according to gate connectivity context, using input slew rates, output capacitance loading, current source capability, and chip operating conditions (process quality, voltage, and temperature – "PVT"). A sophisticated scalable polynomial, composite current source, or other complicated model must be used.

Also, random manufacturing process variations tend to be greater, the smaller the pitch. So, electromigration has become an issue for aluminum metal traces, requiring the more complex process of fabrication of copper traces.

As pitches have shrunk, clock speeds have increased; and, with leakage, this has put power dissipation in the forefront of all design problems. To mitigate power loss, and to reduce switching slew rates, voltage supplies on chip have dropped from 5 V or more to around 1 V.

Power loss, which goes more or less as the square of supply voltage, can be reduced by optimizing different chip areas for different operating voltages, so voltage level shifters have become commonplace. All clock and data traces generally must be level-shifted between different voltage domains. Furthermore, power consumption of many battery-operated digital devices has to be minimized by isolating different chip areas, using specialized data isolation cells, for low-speed or sleep modes

of operation. Sleep modes may require retention of state by additional scan-like sequential cells, or by register duplication in always-on retention latches.

In addition to random fabrication variations, the inevitable but predictable diffraction of light at near-wavelength dimensions becomes a major factor in mask design, requiring corrective reshaping of masks (optical proximity corrections) to obtain pattern images adequate to fabricate working devices in the silicon. The growing requirement for diffractive corrections is shown in Fig. 24.2.

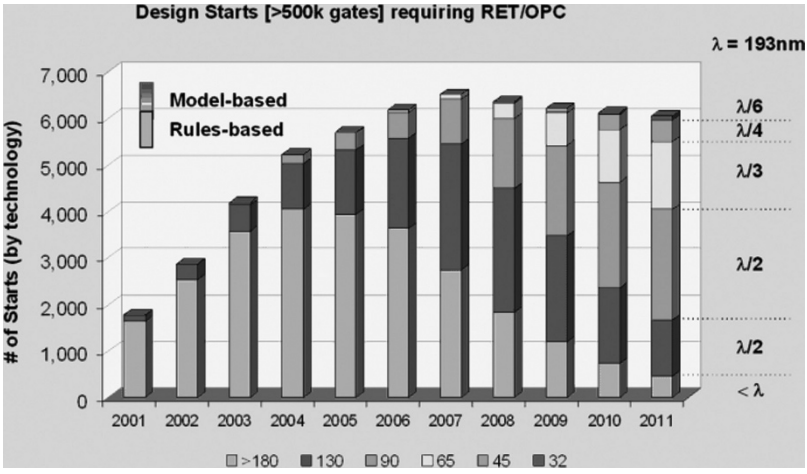


Fig. 24.2 The growth of optical proximity correction in digital design (Reproduced courtesy of W. Staud, Invarium, Inc.)

Diffraction problems are equivalent to quantum-effect problems. For light, the quanta have the wavelength of the masked photons; for fabricated structures, the quanta have the wavelength of the logic-gate electrons. Low-*k* dielectric leakage is directly calculable in terms of quantum tunnelling of the electrons which operate the field-effect transistor components of each on-chip CMOS device.

In a different concern, huge designs have made simple full internal scan chains impractically slow. Deep submicron internal scan has to be hierarchical, with subchains nested and muxed to the TAP port, requiring complicated TAP controller logic to select subchains for the hardware tester.

At modern clock speeds, RF interactions also are becoming important, especially for serial transfer domains, such as those of ethernet, wireless antenna interfaces, and PCI Express.

All these new problems can be solved by proper physical design. While physical design is beyond the scope of this course of study, good design, especially intelligent partitioning, at the verilog source level can ease greatly the work of finding physical solutions to these, and to other, new, deep submicron problems as they arise.

Design pitches will continue to shrink until quantum uncertainty finally brakes progress in this direction. It is unclear what will be the scale at which fundamental

limits prevent further reductions in digital structures, but it probably will be below 10 nm, the linear extent of 50 to 100 average atoms in the crystal structure of a solid.

24.1.2 The Bigger Problem

One should be aware that the deep-submicron problems definitely will increase as pitches shrink. However, based on industry surveys, even at about 130 nm, the primary causes of chip failures resulting in respin still are functional failure (logical design error) or clock-timing design error. According to these data, at 90 nm one would expect that perhaps only 25% of chip failures specifically will be because of deep submicron factors.

> Therefore, the verilog designer can prevent the majority of chip respin failures by entering correct functionality, with accurate assertion limits, and by preparing error-free clocking.

24.1.3 Modern Verification

Palnitkar (2003) and others group assertions with verification tools; however, the present author would prefer to view assertions as part of the design, along with timing checks and synthesis constraints.

The present author views verification as an operation independent of design entry; verification is a practice which finds design errors. Assertions should be entered by the designer and usually are based solely on design considerations (how the design *should* simulate). Of course, as a result of design verification, QA assertions may be added, after the fact, to catch errors by preventing them from recurring silently.

So, we shall not discuss assertions here, with verification.

Verification of a verilog design should be understood as beyond the simple use of a compiler for a simulator or synthesis application. Compilation only checks the syntax of the language in which the design which was entered.

Usually, ***functional verification*** is done during design by use of a simulator. Because the synthesized netlist is optimized by netlist library component timing, and not verilog source code delays, simulation must be repeated, at least on a design-unit level, using a netlist and back-annotated delays.

Although a full chip, nowadays easily over 5 million equivalent gates, can not be debugged functionally on the typical workstation, designers resort to simulation of a netlist of arbitrary size by using distributed computation, hardware simulation acceleration, or hardware emulation in FPGA's.

Timing verification is a major step in signoff of a physical design before tape-out for manufacturing. Any one delay out of specified limits, on a chip with twenty million traces, can cause failure and loss of millions of dollars in redesign and refabrication costs. Potentially, such failures can cause additional, far greater, sales losses because of late entry in the product market window.

Timing verification rarely is done by simulation, because of infeasibility of meaningfully simulating tens of millions of traces, and because of inability to evaluate simulation results even if such exhaustive simulations could be done. Instead, static timing verifiers are used which estimate delays from physical library data and back-annotated netlists. A static verification run can be completed on a modern design in a day.

24.1.4 Formal Verification

An important tool for functional verification is formal verification. A formal verification tool runs statically, thus making possible the complete checking of very large designs.

There are several distinct kinds of formal verification:

- **Equivalence Checking**. This is the most popular type of formal verification; tools implementing equivalence checking of a new netlist against a "golden standard netlist" (functionally known to be correct) are mature and reliable. Typically, a verdict is reached of "equivalent", "functionally different", or "cannot determine equivalence".

  Tools which check a netlist against an RTL model are not so well perfected and often impose coding style requirements in addition to those for synthesis. Of course, any RTL design which can be synthesized can be submitted to a netlist equivalence checker.

  There also exist equivalence checkers for system-level designs against RTL implementations; however, such tools currently are experimental and are not yet widely used.
- **Model Checking**. These tools are analogous to tools which check library component characterization in *ALF* or *Liberty* against a *SPICE* model: A model is defined in a specialized language (for example, *PSL*; or, *P*roperty *S*pecification *L*anguage; IEEE Std 1850), and an implementation in an HDL such as verilog then is checked against the model. Often, such a tool can be considered a requirements checker as well as an implementation checker. Bugs can be found this way, and specific design units can be verified; but, usually, a complete verification of an entire implementation is not practical. Many of these tools can handle simple sequential logic.
- **Formal Proving**. This approach describes an implementation in logical statements and attempts to find inconsistencies. It can be applied fairly easily to blocks of combinational logic, but it can be more complex and difficult than an HDL when sequential logic is involved.

Some tools combine formal verification with partial simulation, allowing the designer to step through formally difficult parts by simulation, and to obtain complete verification of those parts of the design most amenable to formal methods.

24.1.5 Nonlogical Factors on the Chip

We have studied only digital design in this course. Digital design requires correct functionality and correct timing for correct logical operation. However, independent of these two primary factors are those implied by physical interactions among logic gates and by location on the chip. A working chip must have a proper distribution grid for gate power supply (VDD and VSS), and, it must be designed so that gate activity can not interfere with that of other gates:

- **Power Distribution**. As already mentioned, chips can be designed with regions operating at different voltages. The usual reason for this is conservation of power (in battery-operated devices) or limitation of high temperatures caused by power dissipation. Also, adoption of IP blocks in a big chip can be facilitated if the chip can meet special power-supply requirements of the IP.

  Many devices can operate in a "normal" vs. a sleep mode. In the sleep mode, clocking simply may be suspended; however, power also may be reduced or switched off to reduce leakage in unused regions. In many designs, state can be saved on entering sleep mode and restored when normal function is resumed.

  A chip typically is bounded by its I/O pads; just inside these pads, for chips with a single power supply, power and ground are distributed in two rings. A specific voltage region in a multivoltage chip also will be ringed by power and ground supplies. The rings provide noise isolation in addition to supplying power.

  Within any voltage region, the high and low potentials will be supplied in a grid covering the interior of the region. Each row of cells will get vias connecting its power pins to the high and the low supplies. The grid provides multiple current sources and sinks so that brief surges are averaged out, reducing variation at each individual gate.
- **Noise Estimation**. Before being taped out, a chip must be checked by simulation or static verification for noise. No possible gate activity should be capable of coupling unconnected logical states from one gate to another. Coupling simply may be capacitive; however, when changes are rapid, and gate distances and switching transients are fast enough to cause metal traces to maintain significant potential differences along themselves, inductive coupling also must be taken into account. Switching changes also can radiate electromagnetically at trace corners, causing power loss in addition to noise.

  Coupled noise can cause changes in logic level. More common is the problem of alteration of timing: A coupled potential in the direction of a changing level in a nearby trace can speed up the change; in the opposing direction, it can retard

the change. Either of these effects can cause failure of the nearby operation by creating a setup or hold error.

To prevent noise, the design must include shielding (unused power or ground traces between switching logic traces) or must allow enough distance between nearby traces that the coupling is not significant under normal operating conditions. Long parallel runs of possibly interfering logic must be rerouted. The thickness (perpendicular to chip surface) of the traces also may be varied to reduce coupling, because thick traces cross-couple capacitively better than thin ones; however, in many fabrication processes, metal or polysilicon thickness is predetermined and can not be varied locally.

In any event, the noise of contiguous logic, especially traces, must be estimated and controlled if the chip is to be fabricated reliably.

24.1.6 System Verilog

System Verilog is an Accellera standard recently adopted by IEEE. The last Accellera version was 3.1a; this draft is available as a free download from the Accellera web site, but the only authoritative document is from IEEE.

This language is a superset of *verilog-2001* which has been extended to include many *C++* design features as well as a standalone assertion sublanguage.

An important goal of System Verilog is to make porting of code to and from *C++* easier than it has been with *verilog-2001*. Another important goal is to make complex, system-level design and assertion statements directly expressible.

Some features of System Verilog are

- User-defined types (`typedefs` as in *C++*), and VHDL-like type-casting controls.
- Various pointer, reference, dynamic, and queue types.
- Simulator event scheduler extensions.
- Pascal-like nested `module` declarations, VHDL-like `packages` for declarations, and *C++*-like `class` declarations (with single inheritance, only).
- New basic variable type (`logic`) may be assigned procedurally or concurrently; new basic two-state `bit` type.
- Timing-check-like gated event control by `iff`:

```
always@(a iff Ena==1'b1) ...;
```

- `timeunit` and `timeprecision` declarations instead of ‘`timescale`.
- *C*-like `break`, `continue`, and `return` (no block name required).
- Module `final` block; synthesis `always` block variants: `always_latch`, `always_ff`, `always_comb`.
- New `interface` type to make reuse of module I/O declarations easier.
- A self-contained assertion language subset permitting assertions in module scope or in procedural code.

- `covergroup` for functional code coverage.
- New programming interface, the DPI (Direct Programming Interface).

In the present author's opinion, the most important of these enhancements are:

- packages
- `break`, `continue`, and `return`
- interface types
- assertion language.

24.2 Continued Lab Work (Lab 23 or later)

24.2.1 Additional Study

(Optional) Read Thomas and Moorby (2002) section 11.1 for a project in which toggle-testing is used to represent power dissipation as a function of adder structure. You may wish to implement the verilog models suggested.

Optional Readings in Palnitkar (2003)

Read section 15.1 on simulation, emulation, and hardware acceleration as aids to design verification. Also read the summary in section 15.4.

Read section 15.3 on formal verification.

Index