Comprehensive Verilog

Training Workbook

May 2001

RESTRICTED RIGHTS LEGEND 03/97

Contractor/manufacturer is:
DOULOS Ltd.
Church Hatch
22 Market Place
Ringwood
Hampshire BH24 1AW
UK

A complete list of trademark names appears in a separate “Trademark Information” document.

Trademark Information

Mentor Graphics Trademarks

The following names are trademarks, registered trademarks, and service marks of Mentor Graphics Corporation:
3D Design™, A World of Learning(SM), ABIST™, Arithmetic BIST™, AccuPARTner™, AccuParts™, AccuSim®, ADEPT™, ADVance™ MS, ADVance™ RFIC, AMPLE™, Analog Analyst™, Analog Station™, AppNotes(SM), ARTgrid™, ArtRouter™, ARTshape™, ASICPlan™, ASICVector Interfaces™, Aspire™ Assess2000(SM), AutoActive®, AutoCells™, AutoDissolve™, AutoFilter™, AutoFlow™, AutoLib™, AutoLinear™, AutoLink™, AutoLogic™, AutoLogic BLOCKS™, AutoLogic FPGA™, AutoLogic VHDL®, AutomotiveLib™, AutoPAR®, AutoTherm®, AutoTherm Duo™, AutoThermMCM™, Autowire Station™, AXEL™, AXEL Symbol Genie™, Bist Architect™, BIST Compiler(SM), BIST-In-Place(SM), BIST-Ready(SM), Block Station®, Board Architect™, Board Designer™, Board Layout™, Board Link™, Board Process Library™, Board Station®, Board Station Consumer™, BOLD Administrator™, BOLD Browser™, BOLD Composer™, BSDArchitect™, BSPBuilder™, Buy on Demand™, Cable Analyzer™, Cable Station™, CAECO Designer™, CAEFORM™, Calibre®, Calibre DRC™, Calibre DRC-H™, Calibre LVS™, Calibre LVS-H™, Calibre OPCpro™, Calibre ORC™, Calibre PRINTimage™, Calibre PSMgate™, Calibre WORKbench™, Calibre RVE™, Calibre MGC™, CAM Station™, Capture Station®, CAPITAL H™, CAPITAL H the complete desktop engineer®, Cell Builder™, Cell Station®, CellFloor™, CellGraph™, CellPlace™, CellPower™, CellRoute™, Centricity™, CEOC™, CheckMate™, CheckPlot™, CHEOS™, Chip Station®, ChipGraph™, ChipLister™, Circuit PathFinder™, Co-Verification Environment™, Co-Lsim™, CodeVision™, CommLib™, Concurrent Board Process(SM), Concurrent Design Environment™, Connectivity Dataport™, Continuum™, Continuum Power Analyst™, CoreAlliance™, CoreBIST™, Core Builder™, Core Factory™, CTIntegrator™, DataCentric Model™, Datapath™, Data Solvent™, dBUG™, DC Analyzer™, Design Architect®, Design Architect Elite™, Design Capture™, Design Only™, Design Manager™, Design Station®, DesignView™, DesignWide™, DesktopASIC™, Destination PCB®, DFTAdvisor™, DFTArchitect™, DFTCompiler™, DFTInsight™, DirectConnect(SM), Documentation Station™, DSS (Decision Support System)™, Eldo™, ePartners™, EParts®, E3LCable™, EDGE (Engineering Design Guide for Excellence)(SM), Empowering Solutions™, Engineer's Desktop™, EngineerView™, ENRead™, ENWrite™, ESim™, Exemplar Logic®, Expedition™, Expert2000(SM), Explorer CAECO Layout™, Explorer CheckMate™, Explorer Datapath™, Explorer Lsim™, Explorer Lsim-C™, Explorer Lsim-S™, Explorer Ltime™, Explorer Schematic™, Explorer VHDLsim™, ExpressI/O™, FabLink™, Falcon®, Falcon Framework®, FastScan™, FastStart™, FastTrack Consulting(SM), First-Pass Design Success™, First-pass success(SM), FlexSim™, FlexTest™, FDL (Flow Definition Language)™, FlowTabs™, FlowXpert™, FORMA™, FormalPro™, FPGA Advantage™, FPGAdvisor™, FPGA BoardLink™, FPGA Builder™, FPGASim™, FPGA Station®, FrameConnect™, Galileo®, Gate Station®, GateGraph™, GatePlace™, GateRoute™, GDT®, GDT Core®, GDT Designer™, GDT Developer™, GENIE™, GenWare™, Geom Genie™, HDL2Graphics™, HDL Architect™, HDL Architect Station™, HDL Author™, HDL Designer™, HDL Designer Series™, HDL Detective™, HDL Inventor™, HDL Pilot™, HDL Processor™, HDL Sim™, Hardware Modeling Library™, HIC rules™, Hierarchy Injection™, HotPlot®, Hybrid Designer™, Hybrid Station®, IC Design Station™, IC Designer™, IC Layout Station™, IC Station®, ICbasic™, ICblocks™, ICcheck™, ICcompact™, ICdevice™, ICextract™, ICGen™, ICgraph™, ICLink™, IClister™, ICplan™, ICRT Controller Lcompiler™, ICrules™, ICtrace™, ICverify™, ICview™, ICX™, ICX Custom Model™, ICX Custom Modeling™, ICX Project Modeling™, ICX Standard Library™, IDEA Series™, Idea Station®, INFORM®, IFX™, Inexia™, Integrated Product Development®, Integration Tool Kit™, INTELLITEST®, Interactive Layout™, IntraStep(SM), Inventra™, Inventra Soft Cores™, IP Engine ™, IP Evaluation Kit™, IP Factory™, IP -PCB™, IP QuickUse™, IPSim™, IS_Analyzer™, IS_Floorplanner™, IS_MultiBoard™, IS_Optimizer™, IS_Synthesizer™, ISD Creation(SM), ITK™, It's More than Just Tools(SM), Knowledge Center(SM), Knowledge-Sourcing(SM), LAYOUT™, LNL™, LBIST™, LBIST Architect™, Lc™, Lcore™, Leaf Cell Toolkit™, Led™, LED LAYOUT™, Leonardo®, LeonardoInsight™, LeonardoSpectrum™, LIBRARIAN™, Library Builder™, Logic Analyzer on a Chip(SM), Logic Builder™, Logical Cable™, LogicLib™, *logio*™, Lsim™, Lsim DSM™, Lsim-Gate™, Lsim Net™, Lsim Power Analyst™, Lsim-Review™, Lsim-Switch™, Lsim-XL™, Mach PA™, Mach TA™, Manufacture View™, Manufacturing Advisor™, Manufacturing Cable™, MaskCompose™, MaskPE®, MBIST™, MBISTArchitect™, MCM Designer™, MCM Station®, MDV™, MegaFunction™, Memory Builder™, Memory Builder Conductor™, Memory Builder Mozart™, Memory Designer™, Memory Model Builder™, Mentor™, Mentor Graphics®, Mentor Graphics Support CD(SM), Mentor Graphics SupportBulletin(SM), Mentor Graphics SupportCenter(SM), Mentor Graphics SupportFax(SM), Mentor Graphics SupportNet-Email(SM), Mentor Graphics SupportNet-FTP(SM), Mentor Graphics SupportNet-Telnet(SM), MicroPlan™, MicroRoute™, Microtec®, Mixed-Signal Pro™, ModelEditor™, Model*Sim*®, Model*Sim* LNL™, Model*Sim* VHDL™, Model*Sim* VLOG™, Model*Sim* SE™, ModelStation®, Model Technology™, ModelViewer™, ModelViewer*Plus*™, *MODGEN*™, Monet®, Mslab™, Msview™, MS Analyzer™, MS Architect™, MS-Express™, MSIMON™, MTPI(SM), Nanokernel®, NetCheck™, NETED™, OpenDoor(SM), Opsim™, OutNet™, P&RIntegrator™, PACKAGE™, PARADE™, ParallelRoute-Autocells™, ParallelRoute-MicroRoute™, PathLink™, Parts SpeciaList™, PCB-Gen™, PCB-Generator™, PCB IGES™, PCB Mechanical Interface™, PDLSim™, Personal Learning Program™, Physical Cable™, Physical Test Manager:SITE™, PLA Lcompiler™, PLDSynthesis™, PLDSynthesis II™, Power Analyst™, PowerAnalyst Station™, Pre-Silicon™, ProjectXpert™, ProtoBoard™, ProtoView™, QNet™, QualityIBIS™, QuickCheck™, QuickFault™, QuickGrade™, QuickHDL™, QuickHDL Express™, QuickHDL Pro™, QuickPart Builder™, QuickPart Tables™, QuickParts™, QuickPath™, QuickSimII™, QuickStart™, QuickUse™, QuickVHDL®, RAM Lcompiler™, RC-Delay™, RC-Reduction™, RapidExpert™, REAL Time Solutions!™, Registrar™, Reinstatement 2000(SM), Reliability Advisor™, Reliability Manager™, REMEDI™, Renoir™, RF Architect™, RF Gateway™, RISE™, ROM Lcompiler™, RTL X-Press™, Satellite PCB Station™, ScalableModels™, Scaleable Verification™, SCAP™, Scan-Sequential™, Scepter™, Scepter DFF™, Schematic View Compiler, SVC™, Schemgen™, SDF™ (Software Data Formatter), SDL2000 Lcompiler™, Seamless®, Seamless Co-Designer™, Seamless CVE™, Seamless Express™, Selective Promotion™, SignaMask OPC™, Signal Vision™, Signature Synthesis™, Simulation Manager™, SimExpress™, SimPilot™, SimView™, SiteLine2000(SM), SmartMask™, SmartParts™, SmartRouter™, SmartScripts™, Smartshape™, SNX™, SneakPath Analyzer™, SOS Initiative™, Source Explorer™, Spectra®, SpiceNet™, SST Velocity®, Standard Power Model Format (SPMF)™, Structure Recovery™, Super C™, Super IC Station™, Support Services BaseLine(SM), Support Services ClassLine(SM), Support Services Latitudes(SM), Support Services OpenLine(SM), Support Services PrivateLine(SM), Support Services SiteLine(SM), Support Services TechLine(SM), Support Services RemoteLine(SM), Symbol Genie™, Symbolscript™, SYMED™, SynthesisWizard™, System Architect™, System Design Station™, System Modeling Blocks™, Systems on Board Initiative™, Target Manager™, Tau™, TeraCell™, TeraPlace™, TeraPlace-GF™, TechNotes™, Test Station®, Test Structure Builder™, The Ultimate Site For HDL Simulation™, TimeCloser™, Timing Builder™, TNX™, ToolBuilder™, TrueTiming™, Vlog™, V-Express™, V-Net™, VHDLnet™, VHDLwrite™, Verinex™, ViewCreator™, ViewWare®, Virtual Library™, Virtual Target™, Virtual Test Manager:TOP™, VR-Process(SM), VRTX®, VRTXmc™, VRTXoc™, VRTXsa™, VRTX32®, Waveform DataPort™, We Make TMN Easy™, WorkXpert™, xCalibre™, xCalibrate™, Xconfig™, XlibCreator™, Xpert™, Xpert API™, XpertBuilder™, Xpert Dialogs™, Xpert Profiler™, XRAY®, XRAY MasterWorks®, XSH®, Xtrace®, Xtrace Daemon™, Xtrace Protocol™, Zeelan®, Zero Tolerance Verification™, Zlibs™

Third-Party Trademarks

The following names are trademarks, registered trademarks, and service marks of other companies that appear in Mentor Graphics product publications:

Adobe, the Adobe logo, Acrobat, the Acrobat logo, Exchange, and PostScript are registered trademarks of Adobe Systems Incorporated.

Allegro, Composer, Concept, GED, Veritime, Dracula, GDSII, Verilog, Verilog XL, NC-Verilog, Silicon Ensemble, Analog Artist, OCEAN, Virtuoso, and Leapfrog are trademarks and registered trademarks of Cadence Design Systems, Inc.

Altera is a registered trademark of Altera Corp.

AM188, AMD, AMD-K6, and AMD Athlon Processor are trademarks of Advanced Micro Devices, Inc.

Apple and Laserwriter are registered trademarks of Apple Computer, Inc.

ARIES is a registered trademark of Aries Technology.

ARM is a trademark of Advanced RISC Machines Limited.

ASAP, Aspire, C-FAS, CMPI, Eldo-FAS, EldoHDL, Eldo-Opt, Eldo-UDM, EldoVHDL, Eldo-XL, Elga, Elib, Elib-Plus, ESim, Fidel, Fideldo, GENIE, GENLIB, HDL-A, MDT, MGS-MEMT, MixVHDL, Model Generator Series (MGS), Opsim, SimLink, SimPilot, SpecEditor, Success, SystemEldo, VHDeLDO and Xelga are registered trademarks of ANACAD Electrical Engineering Software, a unit of Mentor Graphics Corporation.

AVR is a registered trademark of Atmel Corporation.

CAE+Plus and ArchGen are registered trademarks of CAE Plus, Inc.

CalComp is a registered trademark of CalComp, Inc.

Canon is a registered trademark of Canon, Inc. BJ-130, BJ-130e, BJ-330, and Bubble Jet are trademarks of Canon, Inc.

Centronics is a registered trademark of Centronics Data Computer Corporation.

ColdFire and M-Core are registered trademarks of Motorola, Inc.

Design Planner, HLD Systems, Logic DP, Physical DP, and Pillar are trademarks or registered trademarks of High Level Design Systems.

Ethernet is a registered trademark of Xerox Corporation.

Foresight and Foresight Co-Designer are trademarks of Nu Thena Systems, Inc.

FrameMaker, Frame Technology, and Frame are registered trademarks of Frame Technology Corporation. FrameWriter, FrameViewer, FrameMath, and the Frame logo are trademarks of Frame Technology Corporation.

FLEXlm is a trademark of Globetrotter Software, Inc.

GenCAD is a trademark of Mitron Corporation.

Hewlett-Packard (HP), LaserJet, MDS, HP-UX, PA-RISC, APOLLO, DOMAIN and HPare registered trademarks of Hewlett-Packard Company.

HCL-eXceed and HCL-eXceed/W are registered trademark of Hummingbird Communications. Ltd.

HSPICE is a registered trademark of Meta-Software, Inc.

Installshield is a registered trademark and servicemark of InstallShield Corporation.

IBM, PowerPC, and RISC Systems/6000 are trademarks of International Business Machines Corporation.

I-DEAS Harness Design is a registered trademark of Meta-Software, Inc.

IKON is a trademark of IKON Corporation.

IKOS and Voyager are registered trademarks of IKOS Systems, Inc.

Imagen, QMS, QMS-PS 820, Innovator, and Real Time Rasterization are registered trademarks of QMS Corporation. imPRESS and UltraScript are trademarks of QMS Corporation.

Infineon and C165 are trademarks of Infineon Technologies AG.

Intel, i960, i386, and i486 are registered trademarks of Intel Corporation.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc.

Linux is a registered trademark of Linus Torvalds.

LM-family and SmartModel are registered trademarks of Logic Modeling Corporation. Speed-Model and Speed Modeling are trademarks of Logic Modeling Corporation.

MACH, XP, and Zycad are trademarks of Zycad Corporation.

MemoryModeler MemMaker are trademarks of Denali Software, Inc.

MIPS is a trademark of MIPS Technologies, Inc.

Motif and OSF/Motif are trademarks of Open Software Foundation, Inc.

MS-DOS and Windows are trademarks of Microsoft Corporation.

MULTI is a registered trademark of Green Hills Software, Inc.

NEC and NEC EWS4800 are trademarks of NEC Corp.

Netscape is a trademark of Netscape Communications Corporation.

OakDSPCore is a registered trademark for DSP Group, Inc.

OSF/Motif is a trademark of the Open Software Foundation, Inc.

PKZIP is a registered trademark fo PKWARE, Inc.

PADS-Perform is a registered trademark of PADS Software, Inc.

Quantic is a registered trademark of Quantic Laboratories, Inc.

QUASAR is a trademark of ASM Lithography Holding N.V.

Red Hat is a registered trademark of Red Hat Software, Inc.

SCO, the SCO logo and The Santa Cruz Operation are trademarks or registerred trademarks of the Santa Cruz Operations, Inc. in the USA and other countries.

Signalscan is a trademark of Design Acceleration, Inc.

SimWave product is a registered trademark of Systems Science, Inc.

SPARC is a registered trademark, and SPARCstation is a trademark, of SPARC International, Inc.

Sun Microsystems, Sun Workstation, and NeWS are registered trademarks of Sun Microsystems, Inc. Sun, Sun-2, Sun-3, Sun-4, OpenWindows, SunOS, SunView, NFS, and NSE are trademarks of Sun Microsystems, Inc.

SuperH is a trademark of Hitachi, Ltd.

Synopsys, Design Compiler, Library Compiler, and Chronologic VCS are trademarks registered trademark of Synopsys, Inc.

Teamwork is a registered trademark of Cadre Technologies, Inc.

Times and Helvetica are registered trademarks of Linotype AG.

TimingDesigner, QuickBench, and Chronology, are registered trademarks of Chronology Corp.

Transmission Line Calculator (TLC), Crosstalk Toolkit (XTK), Crosstalk Field Solver (XFX), Pre-Route Delay Quantifier (PDQ), and Mentor Graphics Board Station Translator (MBX) are trademarks of Quad Design.

Tri-State, Tri-State Logic, tri-state, and tri-state logic are registered trademarks of National Semiconductor Corporation.

UNIX and OPEN LOOK are registered trademarks of UNIX System Laboratories, Inc.

Versatec is a trademark of Xerox Engineering Systems, Inc.

ViewDraw, Powerview, Motive, and Viewlogic are registered trademarks of Viewlogic Systems, Inc.

Visula is a registered trademark of Zuken-Redac.

VxSim, VxWorks and Wind River Systems are trademarks or registered trademarks of Wind River Systems, Inc.

Z80 is a registered trademark of Zilog, Inc.

XVision is a registered trademark of Visionware Limited, a subsidiary of the Santa Cruz Operation, Inc.

X Window System is a trademark of MIT (Massachusetts Institute of Technology).

Other brand or product names that appear in Mentor Graphics product publications are trademarks or registered trademarks of their respective holders. Updated 01/15/01.

TABLE OF CONTENTS

Module 1

Module 2

TABLE OF CONTENTS (cont.)

Module 3

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

Module 7

Module 8

TABLE OF CONTENTS (cont.)

Module 9

TABLE OF CONTENTS (cont.)

Module 10

Module 11

TABLE OF CONTENTS (cont.)

TABLE OF CONTENTS (cont.)

Appendix A

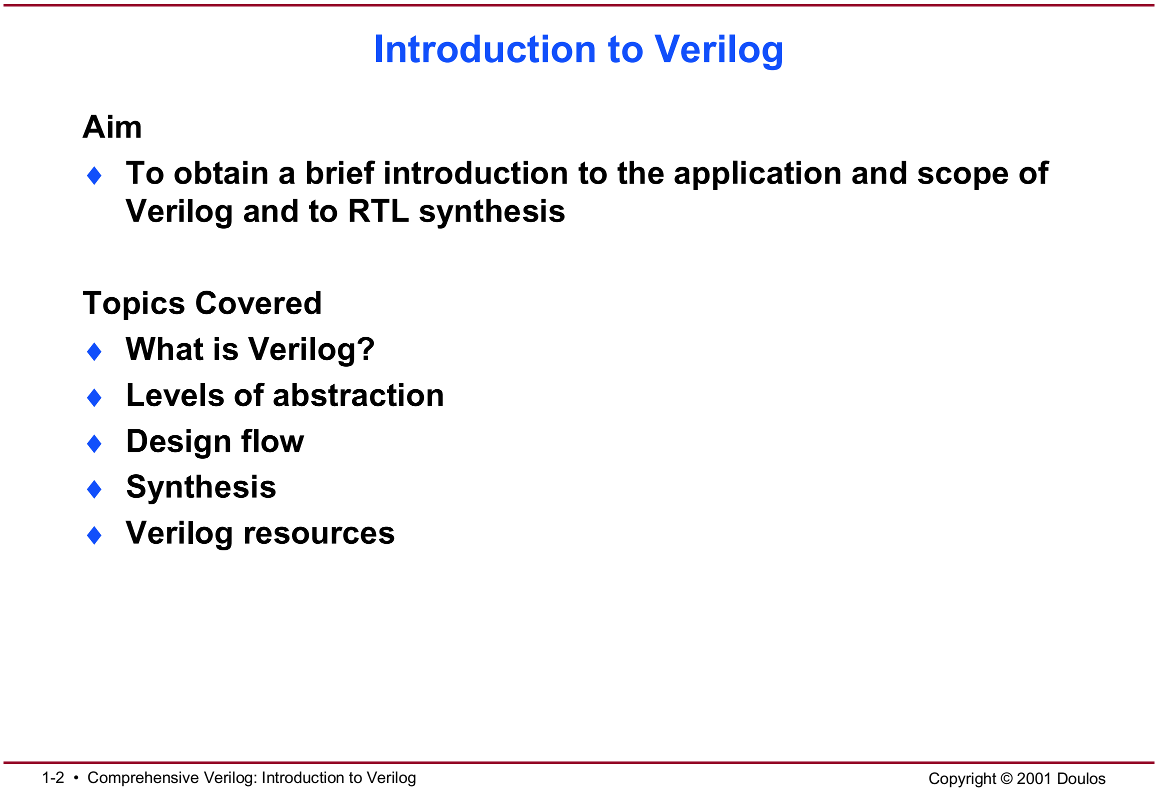
TABLE OF CONTENTS (cont.)

Appendix B

TABLE OF CONTENTS (cont.)

Module 1
Introduction to Verilog

Introduction to Verilog



Introduction to Verilog

Aim

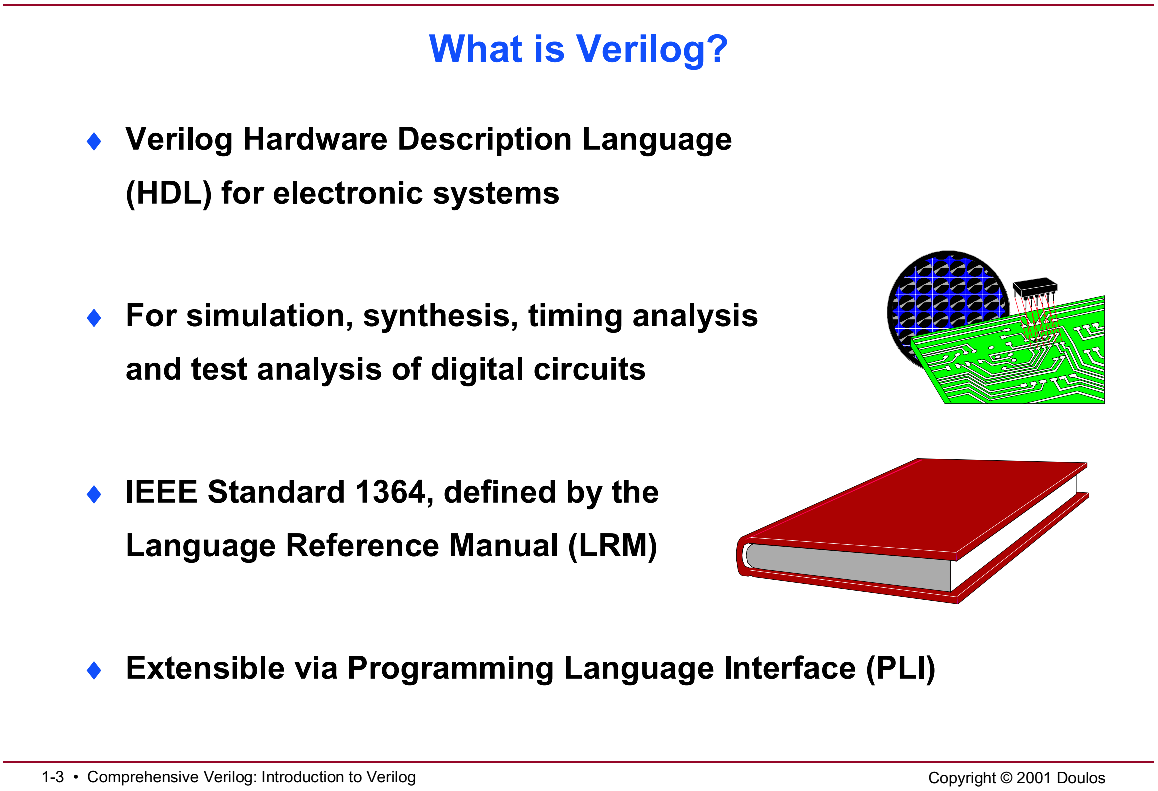
- To obtain a brief introduction to the application and scope of Verilog and to RTL synthesis

Topics Covered

- What is Verilog?
- Levels of abstraction
- Design flow
- Synthesis
- Verilog resources

Notes:

What is Verilog?



What is Verilog?

- Verilog Hardware Description Language (HDL) for electronic systems
- For simulation, synthesis, timing analysis and test analysis of digital circuits
- IEEE Standard 1364, defined by the Language Reference Manual (LRM)
- Extensible via Programming Language Interface (PLI)

1-3 • Comprehensive Verilog: Introduction to Verilog

Notes:

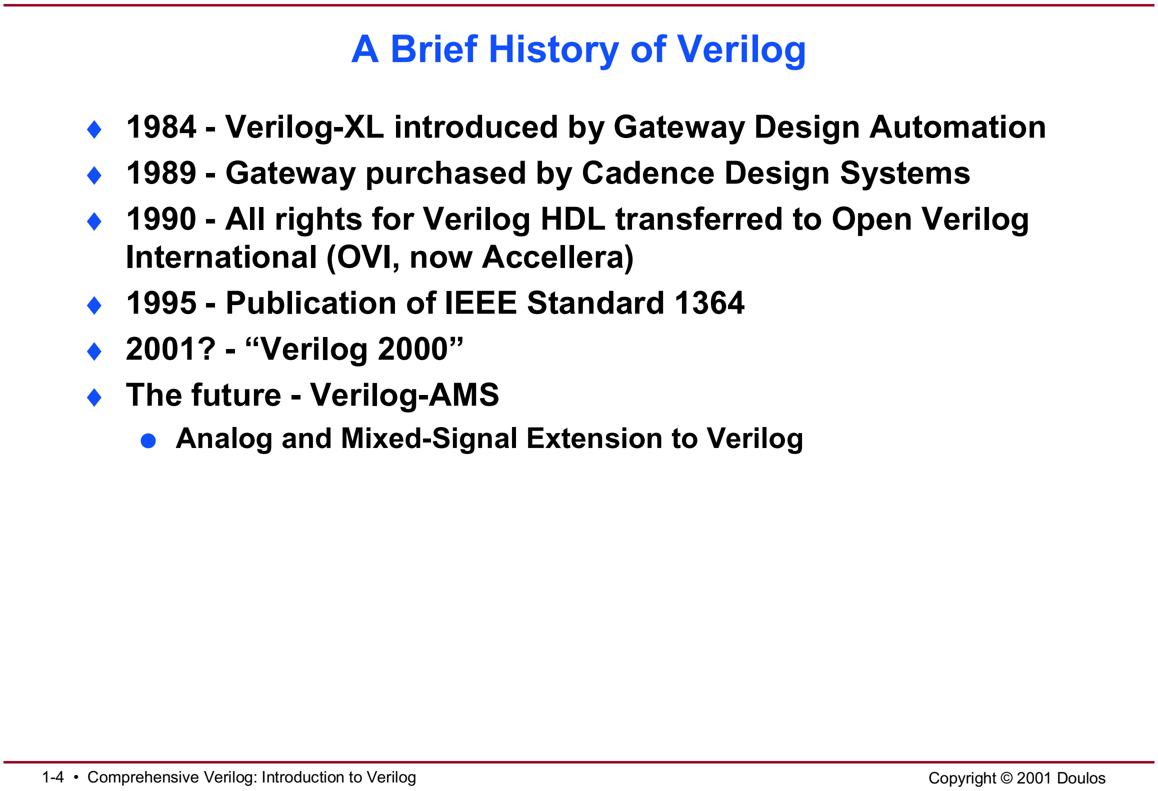
Verilog is a Hardware Description Language; a textual format for describing electronic circuits and systems. Applied to electronic circuit design, Verilog is intended to be used for verification through simulation, for timing analysis, for test analysis (testability analysis and fault grading), and for logic synthesis.

Verilog HDL is an IEEE standard - number 1364. The standard document is known as the Language Reference Manual, or LRM. This is the complete, authoritative definition of the Verilog HDL.

IEEE Std 1364 also defines the Programming Language Interface, or PLI. This is a collection of software routines which permit a bidirectional interface between

Verilog and other languages (usually C). The PLI makes it possible to customize the Verilog language and Verilog tools such as simulators.

A Brief History of Verilog



A Brief History of Verilog

- 1984 - Verilog-XL introduced by Gateway Design Automation
- 1989 - Gateway purchased by Cadence Design Systems
- 1990 - All rights for Verilog HDL transferred to Open Verilog International (OVI, now Accellera)
- 1995 - Publication of IEEE Standard 1364
- 2001? - “Verilog 2000”
- The future - Verilog-AMS
  - Analog and Mixed-Signal Extension to Verilog

1-4 • Comprehensive Verilog: Introduction to Verilog

Notes:

The history of the Verilog HDL goes back to the 1980s, when a company called Gateway Design Automation developed a logic simulator, Verilog-XL, and with it a hardware description language.

Cadence Design Systems acquired Gateway in 1989, and with it the rights to the language and the simulator. In 1990, Cadence put the language (but not the simulator) into the public domain, with the intention that it should become a standard, non-proprietary language.

A non-profit organization, Open Verilog International (OVI) was formed to take the Verilog language through the IEEE standardization procedure. Verilog HDL became IEEE Std. 1364-1995 in December 1995. In 2000, OVI merged with

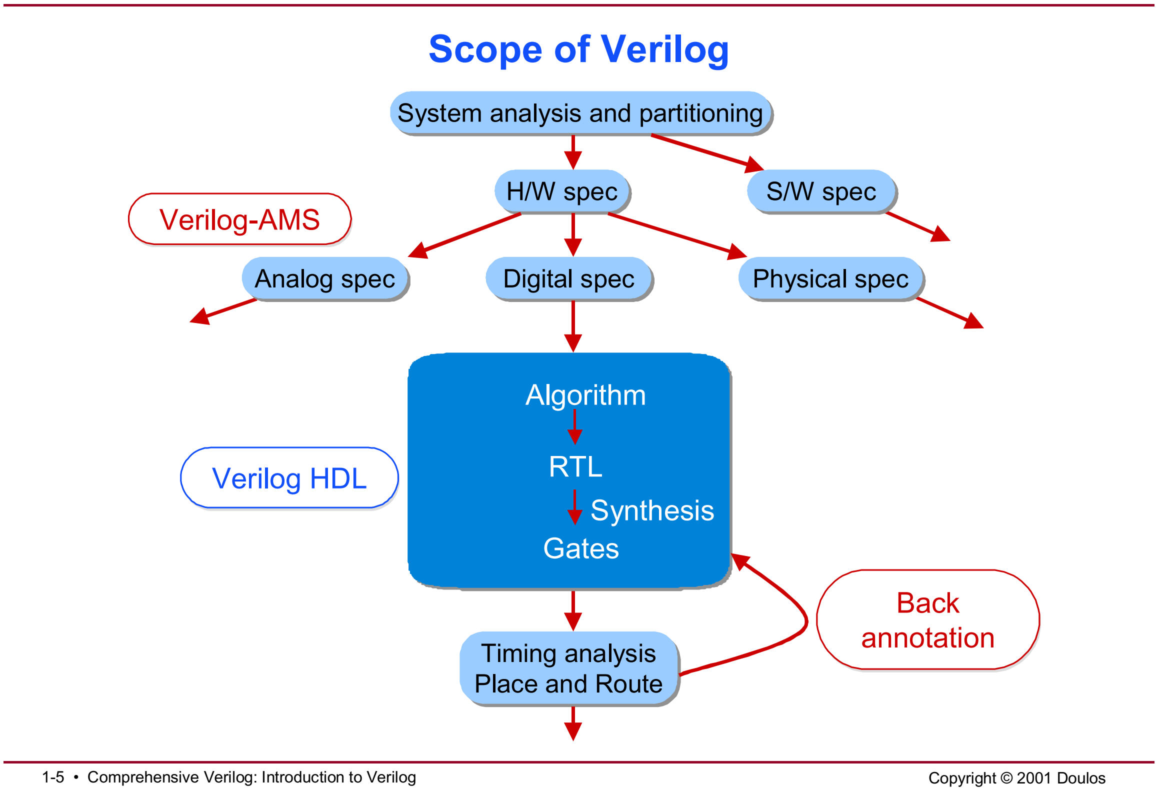
VHDL International to form Accellera, who will have the task of coordinating future Verilog standards.

A draft of a proposed revised standard has been created, provisionally called Verilog 2000 (The name may change - the new standard was not ratified in 2000). This proposes significant changes and enhancements to the Verilog language and the PLI. Verilog 2000 is not covered in this course and it will be a while before simulators and synthesis tools support the new standard.

Work is also proceeding on Verilog-AMS, which will provide analog and mixed-signal extensions to Verilog.

Unless specifically stated otherwise, the terms Verilog and Verilog HDL are used in this course to the language, and not the Verilog-XL simulator.

Scope of Verilog



Scope of Verilog

1-5 • Comprehensive Verilog: Introduction to Verilog

Copyright © 2001 Doulos

Notes:

Verilog HDL is suited to the specification, design and description of digital electronic hardware.

System level

Verilog is not ideally suited for abstract system-level simulation, prior to the hardware-software split. Simulation at this level is usually stochastic, and is concerned with modeling performance, throughput, queuing, and statistical distributions. However, Verilog provides a few system tasks and functions for stochastic modeling, and has been used in this area with some success.

Digital

Verilog is suitable for use today in the digital hardware design process, from specification through high-level functional simulation, manual design, and logic synthesis down to gate-level simulation. Verilog tools provide an integrated design environment in this area.

Verilog can also be used for detailed, implementation-level modelling, that includes switch-level simulation, timing analysis, and fault simulation.

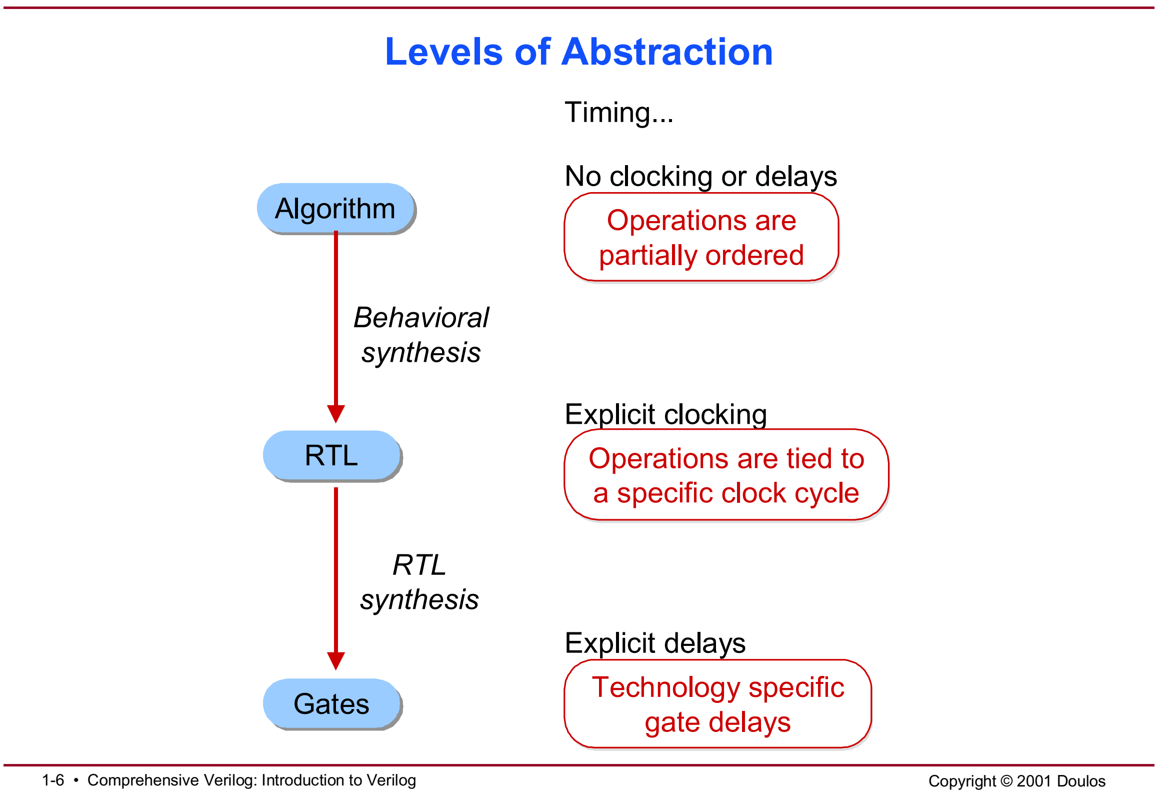
Analog

The proposed Verilog-AMS language enables mixed-signal and behavioral analog modelling and simulation.

Design process

The diagram opposite shows a very simplified view of the electronic system design process that incorporates Verilog. The central portion of the diagram shows the main parts of the design process that will be impacted by Verilog.

Levels of Abstraction



Levels of Abstraction

Timing...

Algorithm

No clocking or delays

Operations are partially ordered

Behavioral synthesis

RTL

Explicit clocking

Operations are tied to a specific clock cycle

RTL synthesis

Gates

Explicit delays

Technology specific gate delays

1-6 • Comprehensive Verilog: Introduction to Verilog

Copyright © 2001 Doulos

Notes:

The diagram above summarizes the high level design flow for an ASIC (i.e. gate array, standard cell array, or FPGA). In a practical design situation, each step shown above may be split into several smaller steps, and may uncover parts of the design flow that iterate as errors.

As a first step, you can use Verilog to model and simulate aspects of the complete system containing one or more ASICs. This may be a fully functional description of the system allowing the ASIC/FPGA specification to be validated prior to starting detailed design. Alternatively, this may be a partial description that abstracts certain properties of the system, such as a performance model to detect system performance bottlenecks.

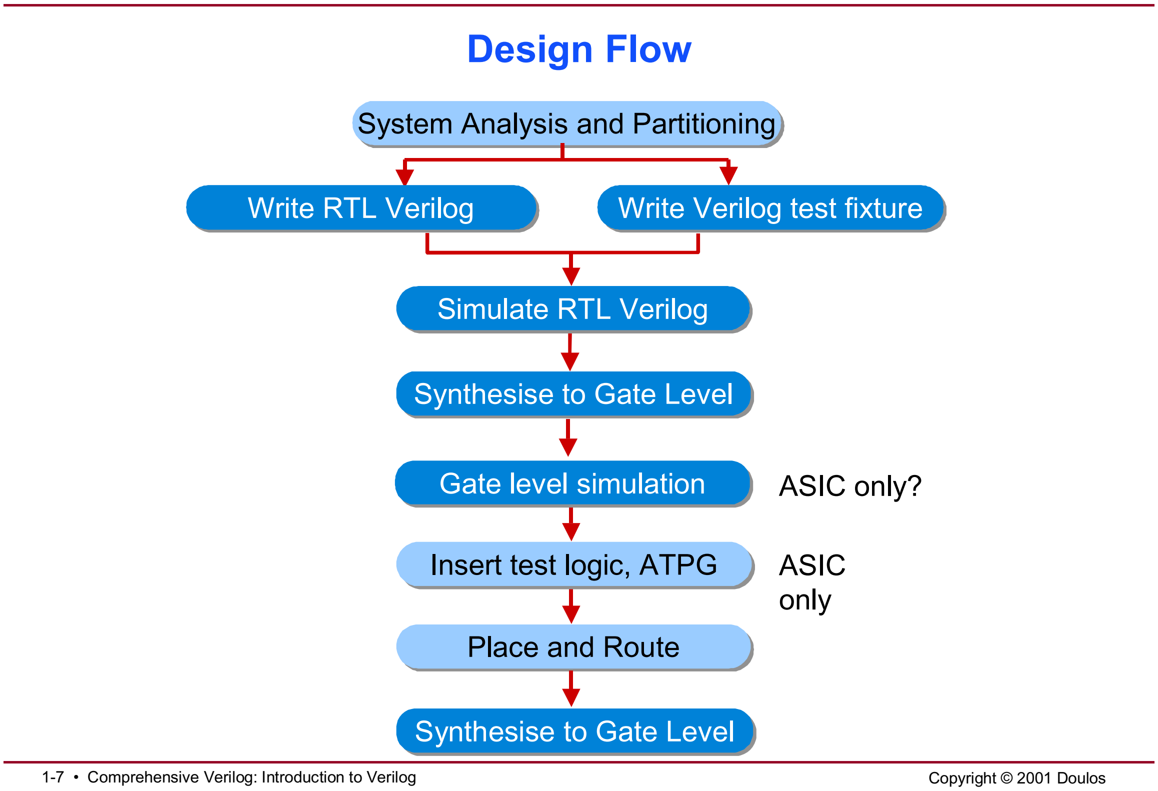
Once the overall system architecture and partitioning is stable, the detailed design of each ASIC can commence. This starts by capturing the ASIC/FPGA design in Verilog at the register transfer level, and capturing a set of test cases in Verilog. These two tasks are complementary, and are sometimes performed by different design teams in isolation to ensure that the specification is correctly interpreted. The RTL Verilog should be synthesisable if automatic logic synthesis is to be used. Test case generation is a major task that requires a disciplined approach and much engineering ingenuity: the quality of the final ASIC depends on the coverage of these test cases.

The RTL Verilog is then simulated to validate the functionality against the specification. RTL simulation is usually one or two orders of magnitude faster than gate level simulation, and experience has shown that this speedup is best exploited by doing more simulation, not spending less time on simulation.

In practice it is common to spend 70-80% of the ASIC/FPGA design cycle writing and simulating Verilog at and above the register transfer level, and 20-30% of the time synthesizing and verifying the gates.

Although some exploratory synthesis will be done early on in the design process, to provide accurate speed and area data to aid in the evaluation of architectural decisions and to check the engineer's understanding of how the Verilog will be synthesized, the main synthesis production run is deferred until functional simulation is complete. It is pointless to invest a lot of time and effort in synthesis until the functionality of the design is validated.

Design Flow



Design Flow

System Analysis and Partitioning

Write RTL Verilog

Write Verilog test fixture

Simulate RTL Verilog

Synthesise to Gate Level

Gate level simulation

ASIC only?

Insert test logic, ATPG

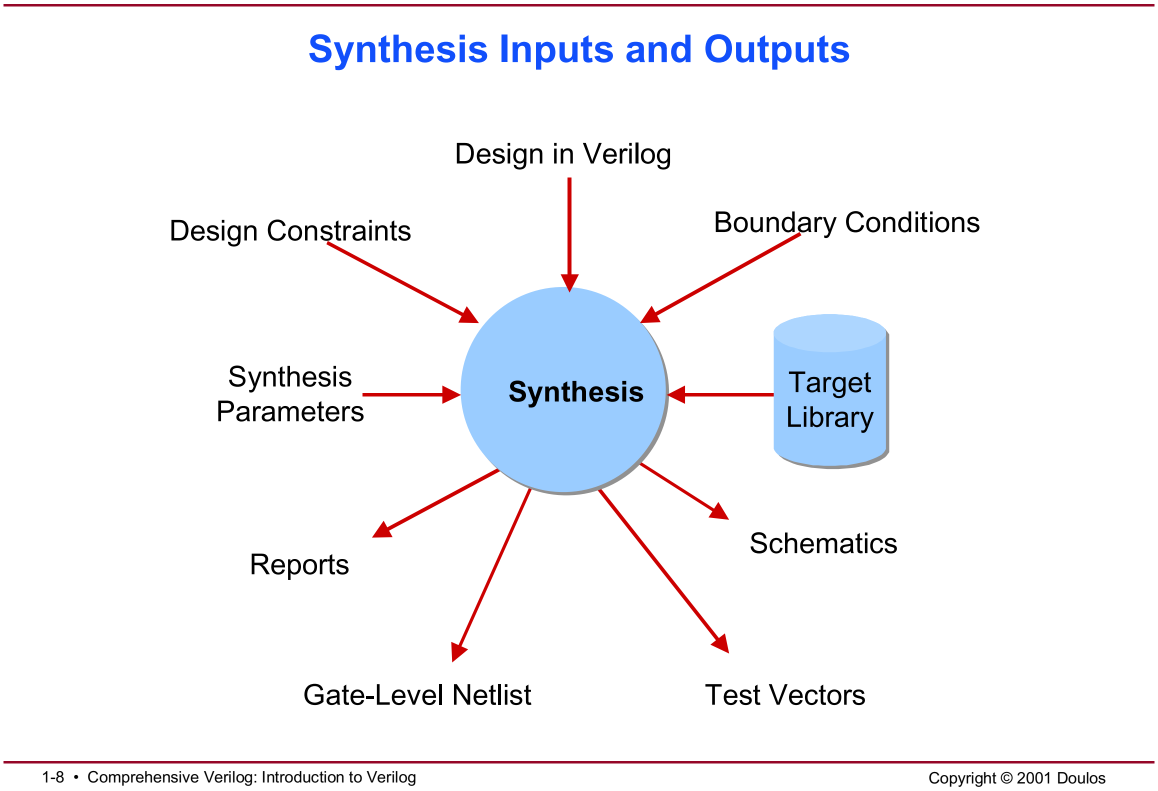
ASIC only

Place and Route

Synthesise to Gate Level

Notes:

Synthesis Inputs and Outputs



Synthesis Inputs and Outputs

Design in Verilog
Design Constraints
Boundary Conditions
Synthesis Parameters
Synthesis
Target Library
Reports
Schematics
Gate-Level Netlist
Test Vectors

1-8 • Comprehensive Verilog: Introduction to Verilog

Notes:

Focusing now on synthesis, this diagram shows the various inputs and outputs to a typical RTL synthesis tool.

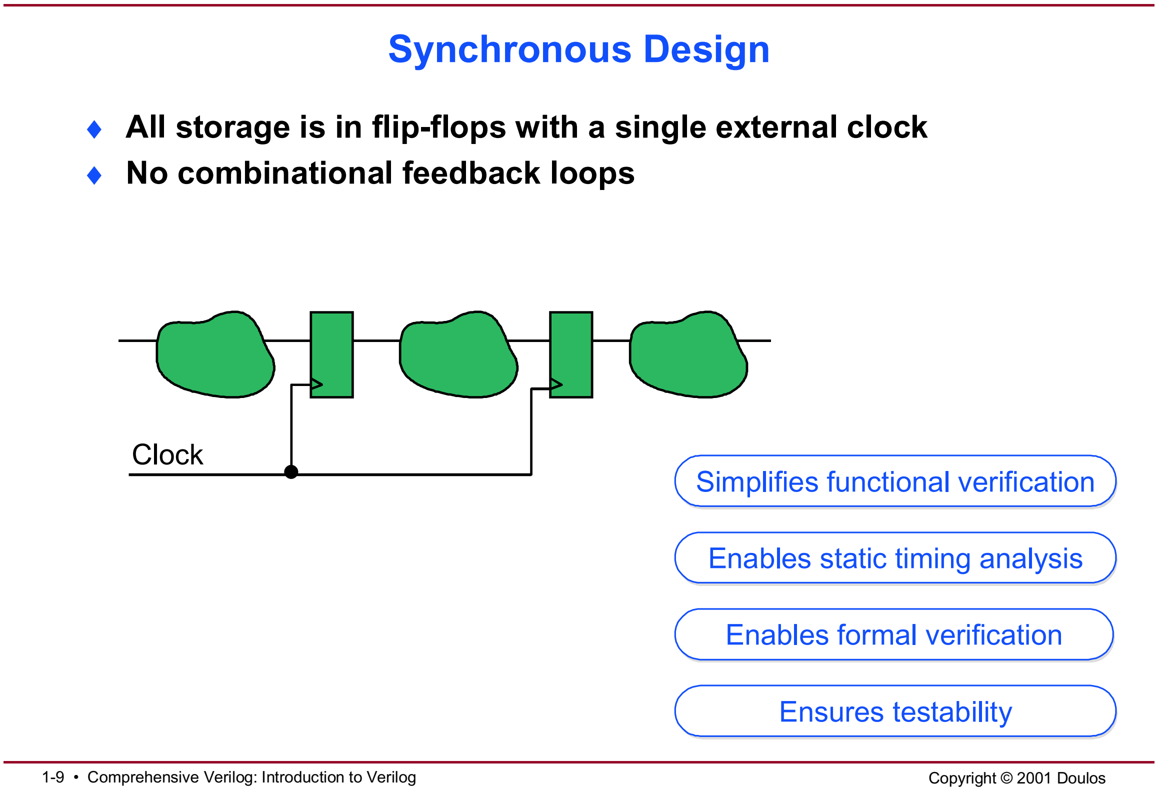
Inputs

The inputs to a synthesis tool includes not only the Verilog source code, but also design constraints, boundary conditions and synthesis parameters and controls. For example, design constraints might include maximum clock period and maximum gate count, boundary conditions might include the capacitive load on output pins, and synthesis parameters might include the CPU time allowed for optimization.

Outputs

The outputs from a synthesis tool include not only the synthesized design (usually in the form of a netlist), but also synthesized schematics and reports which aid in determining the quality of the design. Test synthesis tools will also output a set of automatically generated manufacturing test vectors.

Synchronous Design



Synchronous Design

- All storage is in flip-flops with a single external clock
- No combinational feedback loops

Simplifies functional verification

Enables static timing analysis

Enables formal verification

Ensures testability

1-9 • Comprehensive Verilog: Introduction to Verilog

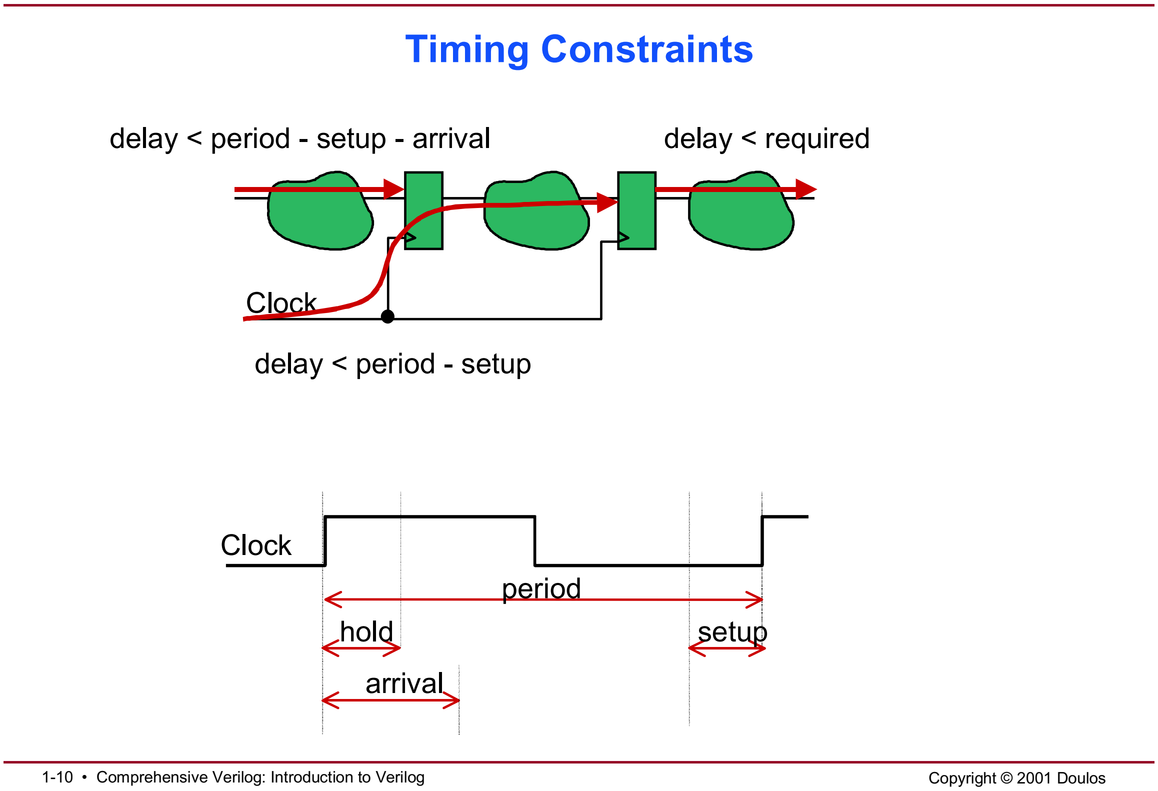
Notes:

RTL synthesis assumes synchronous design, as do many other tools in the design flow. Many common problems that occur during and after synthesis, especially timing problems, can be traced back to using non-synchronous design techniques when the detailed design was conceived and captured in Verilog. Thus, an understanding of synchronous design techniques is an essential foundation for the successful use of RTL synthesis.

A pure synchronous design consists of storage elements - registers - that are triggered on the edge of a single clock, and combinational logic. All storage is in edge-triggered flip-flops with the same clock. All flip-flops (other than pipeline registers) have resets.

Combinational logic is defined as a digital circuit in which the current steady state values of the outputs depend on the current steady state values of the inputs, and on that alone. Any cycles - feedback loops - in the logic must pass through registers.

Timing Constraints



Timing Constraints

1-10 • Comprehensive Verilog: Introduction to Verilog

Copyright © 2001 Doulos

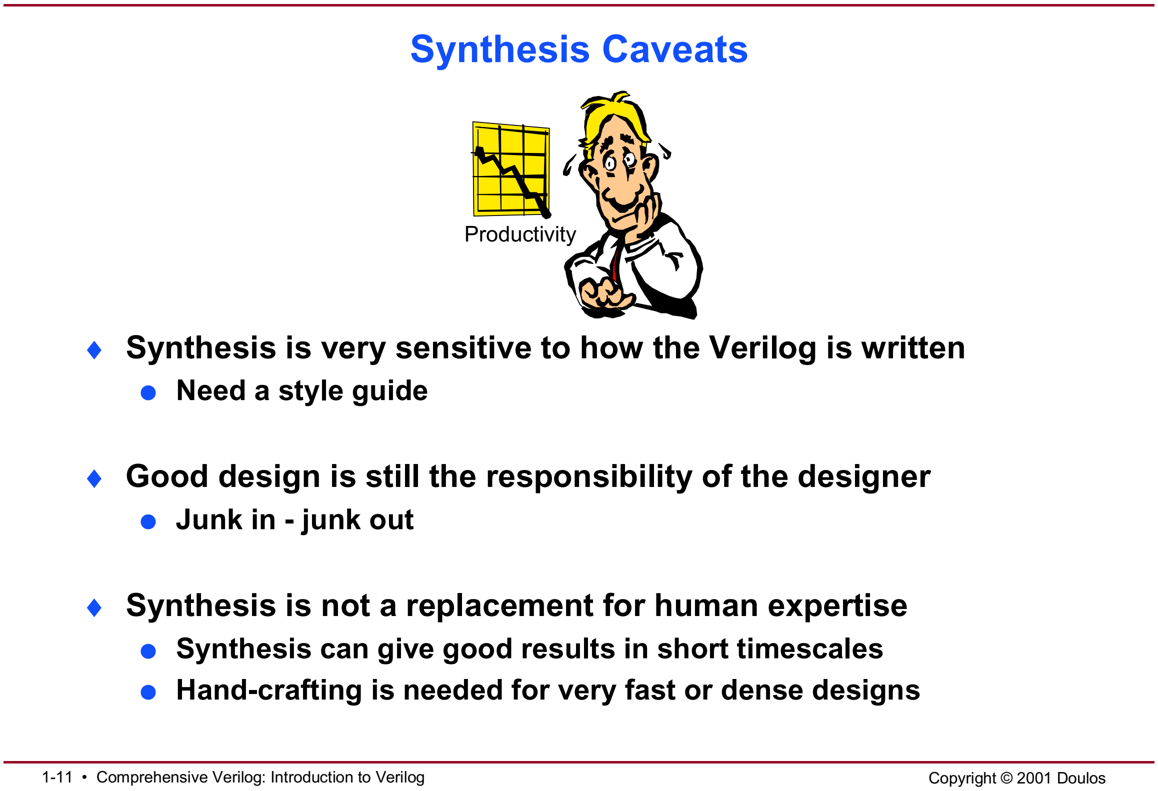
Notes:

Synthesis tools transform Verilog into gates in such a way that the functionality of the design is preserved by the transformation. However, this is only really of any use if the timing of the design is also correct, and synthesis only preserves the timing of synchronous designs.

Synthesis preserves the clock cycle level behavior of the design by optimizing the combinational logic between the registers to meet the target clock period. The data arrival time is calculated by enumerating all of the possible paths through the logic between registers, clock arrival is determined by tracing the Clock path, period is specified as an independent design constraint, and setup is defined in the technology library.

You will need to consider the timing relationships of external signals arriving at the primary inputs of your design to ensure that they do not violate setup and hold times, taking into account any delays through the I/O pads and into the core of the chip. Unfortunately this is a common pitfall; you cannot assume that your design will work when inserted into the wider environment of your system just because the synthesis tool says that the timing is okay!

Synthesis Caveats


Productivity

Synthesis Caveats

- Synthesis is very sensitive to how the Verilog is written
  - Need a style guide
- Good design is still the responsibility of the designer
  - Junk in - junk out
- Synthesis is not a replacement for human expertise
  - Synthesis can give good results in short timescales
  - Hand-crafting is needed for very fast or dense designs

1-11 • Comprehensive Verilog: Introduction to Verilog

Notes:

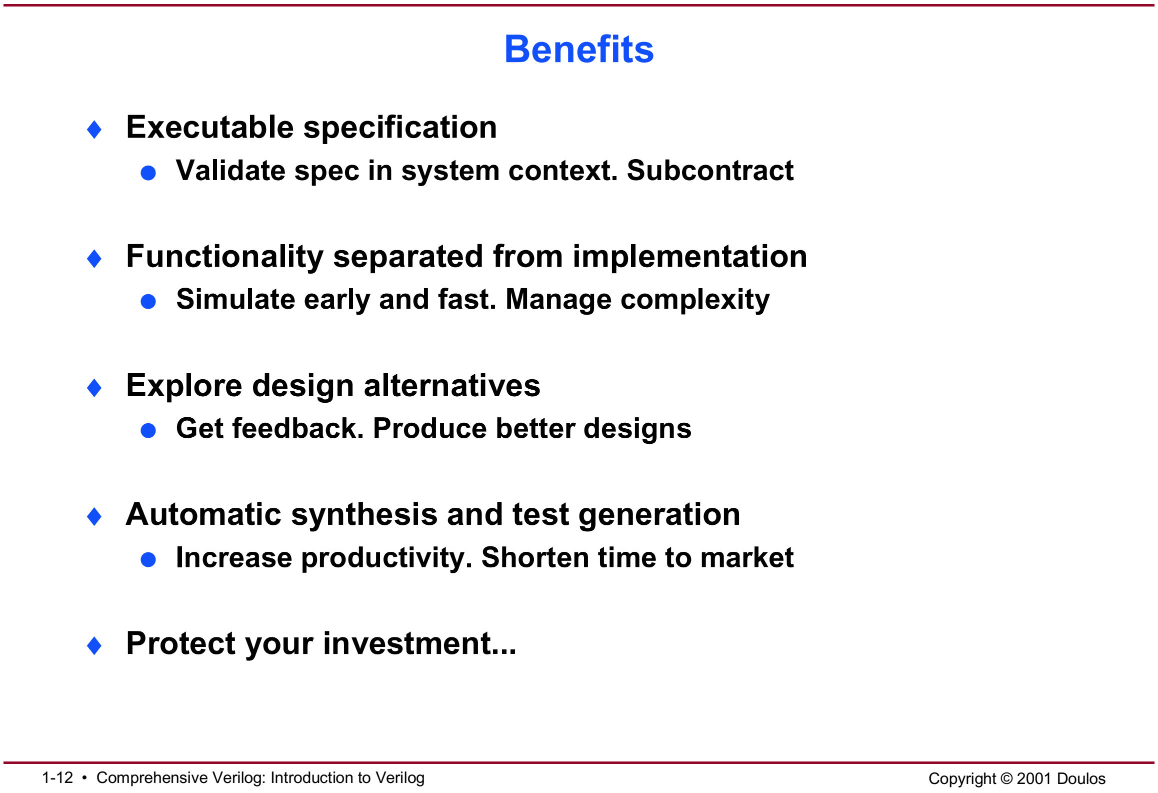
Synthesis is not a panacea! It is vital that everyone working on a project starts with realistic expectations of synthesis. Here are a few pointers.

The results of synthesis are very sensitive to the style in which the Verilog code is written. It is not sufficient that the Verilog is functionally correct; it must be written in such a way that it directs the synthesis tool to generate good hardware, and moreover the Verilog must be matched to the idiosyncrasies of the particular synthesis tool being used.

With RTL synthesis, the designer must be firmly in control of the design! Most of the design decisions are still made by the human, with the synthesis tool automating the gate level implementation. With synthesis, bad designers still produce bad designs, they just do so faster!

In many practical design situations, synthesis will produce results which equal or excel those of the human designer. However, for designs that push the limits of the technology in terms of speed or density, synthesis alone is often not good enough, and manual intervention becomes necessary. Synthesis tools do not offer a substitute for human design knowledge and expertise!

Benefits



Benefits

- Executable specification
  - Validate spec in system context. Subcontract
- Functionality separated from implementation
  - Simulate early and fast. Manage complexity
- Explore design alternatives
  - Get feedback. Produce better designs
- Automatic synthesis and test generation
  - Increase productivity. Shorten time to market
- Protect your investment...

1-12 • Comprehensive Verilog: Introduction to Verilog

Notes:

Executable specification.

It is often reported that a large number of ASIC designs meet their specifications first time, but fail to work when plugged into a system. Verilog allows this issue to be addressed in two ways: A Verilog specification can be executed in order to achieve a high level of confidence in its correctness before commencing design, and may simulate one to two orders of magnitude faster than a gate level description. A Verilog specification for a part can form the basis for a simulation model to verify the operation of the part in the wider system context (E.g. printed circuit board simulation). This depends on how accurately the specification handles aspects such as timing and initialization.

Simulate early

With the high level design methodology, the functionality of the design can be validated before completing the detailed implementation. Early simulation permits errors to be found earlier in the design process, where they are easier and cheaper to fix.

Exploration

Using behavioral and RTL synthesis tools, it is possible to explore more of the design space by trying a number of alternative implementations and picking the best.

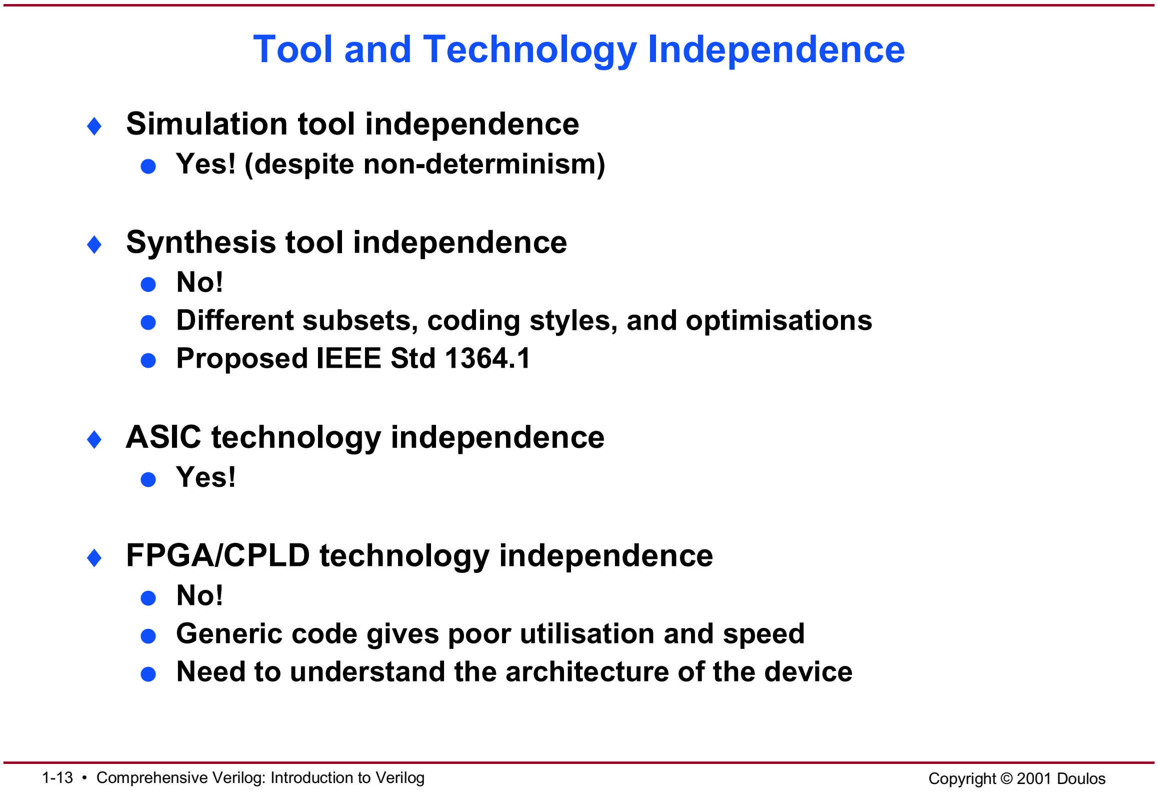
Automation

Automating the generation of the logic circuit and the manufacturing test vectors can shorten the design cycle. However, the most significant contribution of the high level design process to achieving reduced time-to-market is that a very high percentage of Verilog designs are right-first-time, eliminating the need for costly re-spins.

Protect your investment

Many companies are hoping to protect their investment by achieving both tool and technology independence through the use of Verilog. See the next page.

Tool and Technology Independence



Tool and Technology Independence

- Simulation tool independence
  - Yes! (despite non-determinism)
- Synthesis tool independence
  - No!
  - Different subsets, coding styles, and optimisations
  - Proposed IEEE Std 1364.1
- ASIC technology independence
  - Yes!
- FPGA/CPLD technology independence
  - No!
  - Generic code gives poor utilisation and speed
  - Need to understand the architecture of the device

1-13 • Comprehensive Verilog: Introduction to Verilog

Copyright © 2001 Doulos

Notes:

Tools

Verilog descriptions of hardware design and test fixtures are usually portable between design tools, and portable between design centers and project partners. While the Verilog language is inherently non-deterministic (different simulators may give different results for certain models), you shouldn't write non-deterministic code. So you can safely invest in Verilog modeling effort and training, knowing that you will not be tied in to a single tool vendor, but will be free to preserve your investment across tools and platforms. Also, the design automation tool vendors are themselves making a large investment in Verilog, ensuring a continuing supply of state-of-the-art Verilog tools.

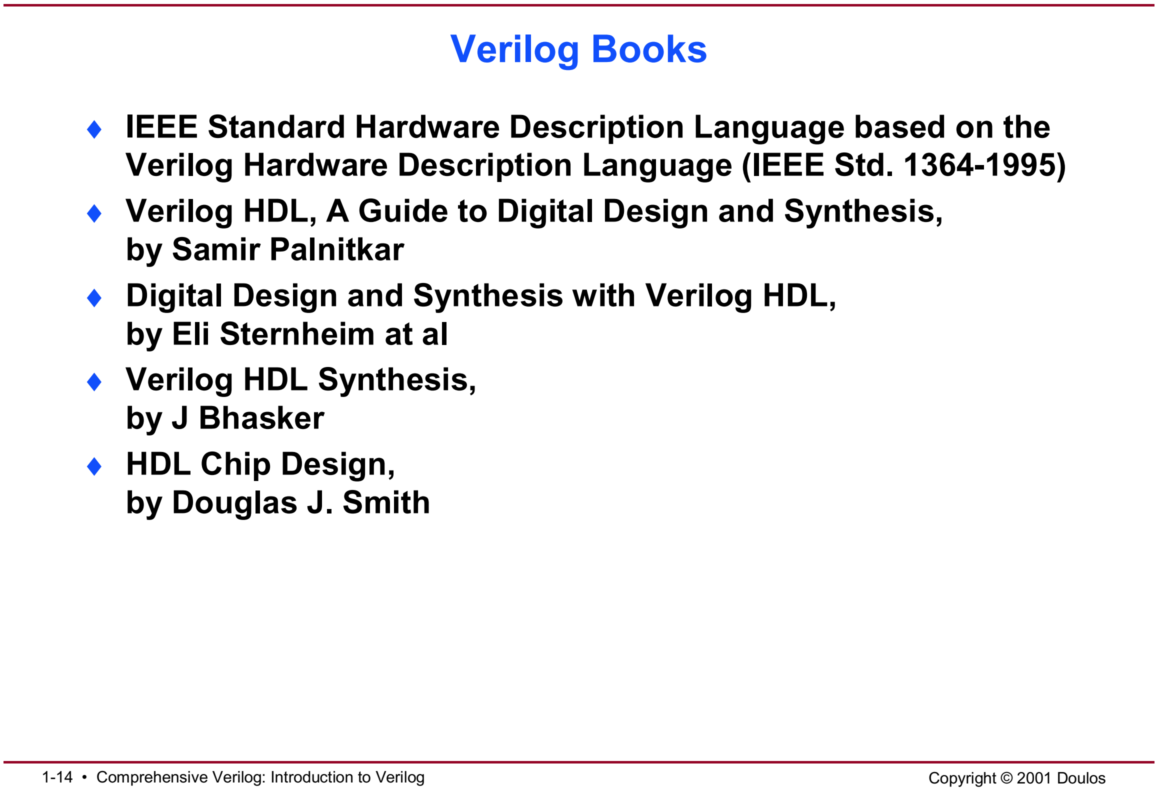
While the above is true for simulation, the use of synthesis tools is a different story, because there is no standard subset or interpretation of Verilog for synthesis. There is a proposed IEEE standard 1364.1 which provides a standard Verilog subset for synthesis, but it will be a while before this is implemented.

Technology

Verilog permits technology independent design through support for top down design and logic synthesis. To move a design to a new technology you need not start from scratch or reverse-engineer a specification - instead you go back up the design tree to a behavioral Verilog description, then implement that in the new technology knowing that the correct functionality will be preserved. This technique can be used to re-implement an FPGA as an standard cell ASIC, for example.

The Verilog based design flow has proved itself very robust in transferring designs between ASIC technologies, but is less successful in transfer between FPGA or CPLD technologies. The sort of generic Verilog coding style that works well for ASICs does not achieve good device utilization or speed when targeting programmable logic devices. To achieve best results with programmable logic, it is necessary to have a good understanding of the architectural features of the target device, and build that knowledge into the Verilog design.

Verilog Books



Verilog Books

- IEEE Standard Hardware Description Language based on the Verilog Hardware Description Language (IEEE Std. 1364-1995)
- Verilog HDL, A Guide to Digital Design and Synthesis, by Samir Palnitkar
- Digital Design and Synthesis with Verilog HDL, by Eli Sternheim at al
- Verilog HDL Synthesis, by J Bhasker
- HDL Chip Design, by Douglas J. Smith

1-14 • Comprehensive Verilog: Introduction to Verilog

Notes:

A number of books about Verilog have been written. Those listed are recommended and are available from the publishers at the time of writing. A complete list of books can be found on the Internet in the Verilog FAQ (see next page).

The IEEE standard provides the official definition of the language, its syntax and simulation semantics, in the form of a reference manual. It was derived from the original Gateway and Cadence reference manual.

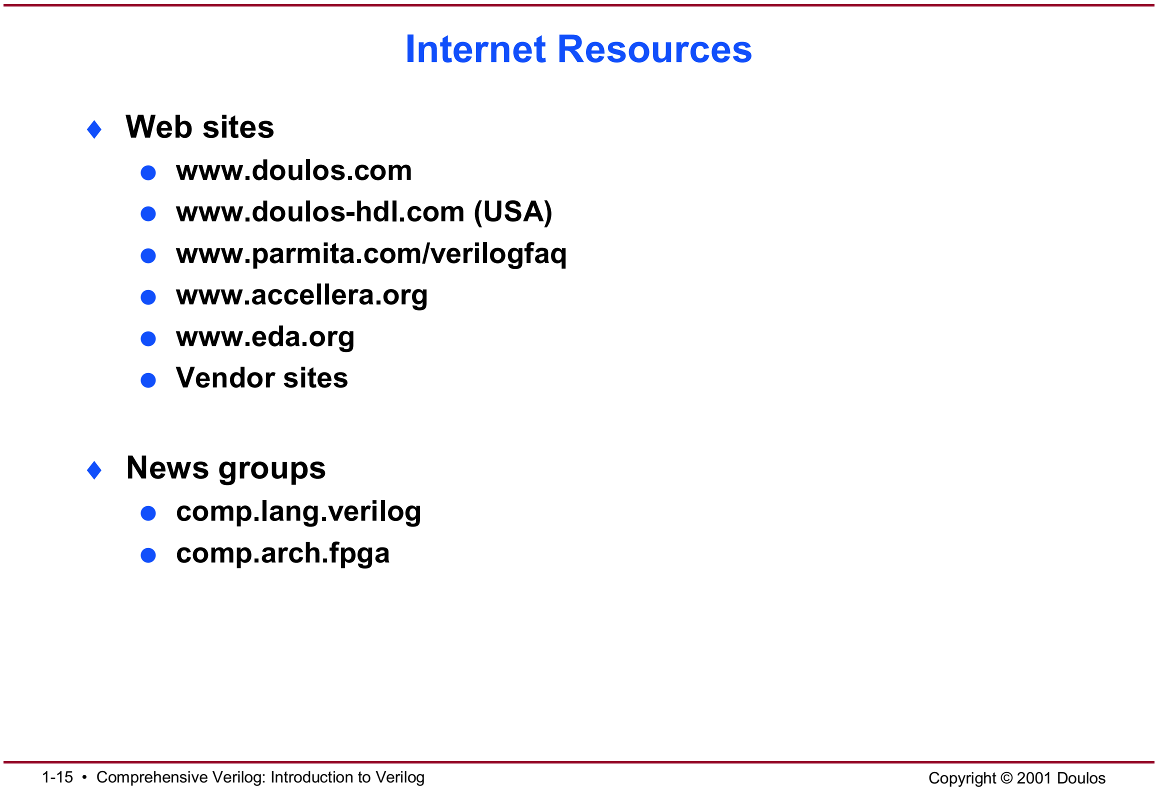
Palnitkar's book is a good introductory textbook on Verilog. It covers a wide range of topics, including synthesis and the PLI. (SunSoft Press/Prentice Hall, 1996. ISBN 0-13-451675-3)

Digital Design and Synthesis with Verilog HDL by Eli Sternheim, Rajvir Singh, Rajeev Madhavan and Yatin Trivedi provides a short introduction to the language, but the main part of the book presents a number of extended examples of Verilog models. (Automata Publishing Company, San Jose, CA 95014, USA. 1993. ISBN 0-9627488-2-X)

Bhasker's book assumes a basic knowledge of the Verilog language and (as the title suggests) is concerned with synthesis. (Star Galaxy Publishing, 1058 Treeline Drive, Allentown, PA 18103,USA. http://members.aol.com/SGalaxyPub. 1998. ISBN 0-9650391-5-3)

Doug Smith's book covers both Verilog and VHDL (another IEEE standard hardware description language). It is both comprehensive and practical, with lots of synthesizable "cookbook" examples. It is especially helpful if you are interested in both Verilog and VHDL. (Doone Publications, 7950, Highway 72W #G106, Madison, Al, 35758, USA. http://www.doone.com.1996. ISBN: 0-9651934-3-8)

Internet Resources



Internet Resources

- Web sites
  - www.doulos.com
  - www.doulos-hdl.com (USA)
  - www.parmita.com/verilogfaq
  - www.accellera.org
  - www.eda.org
  - Vendor sites
- News groups
  - comp.lang.verilog
  - comp.arch.fpga

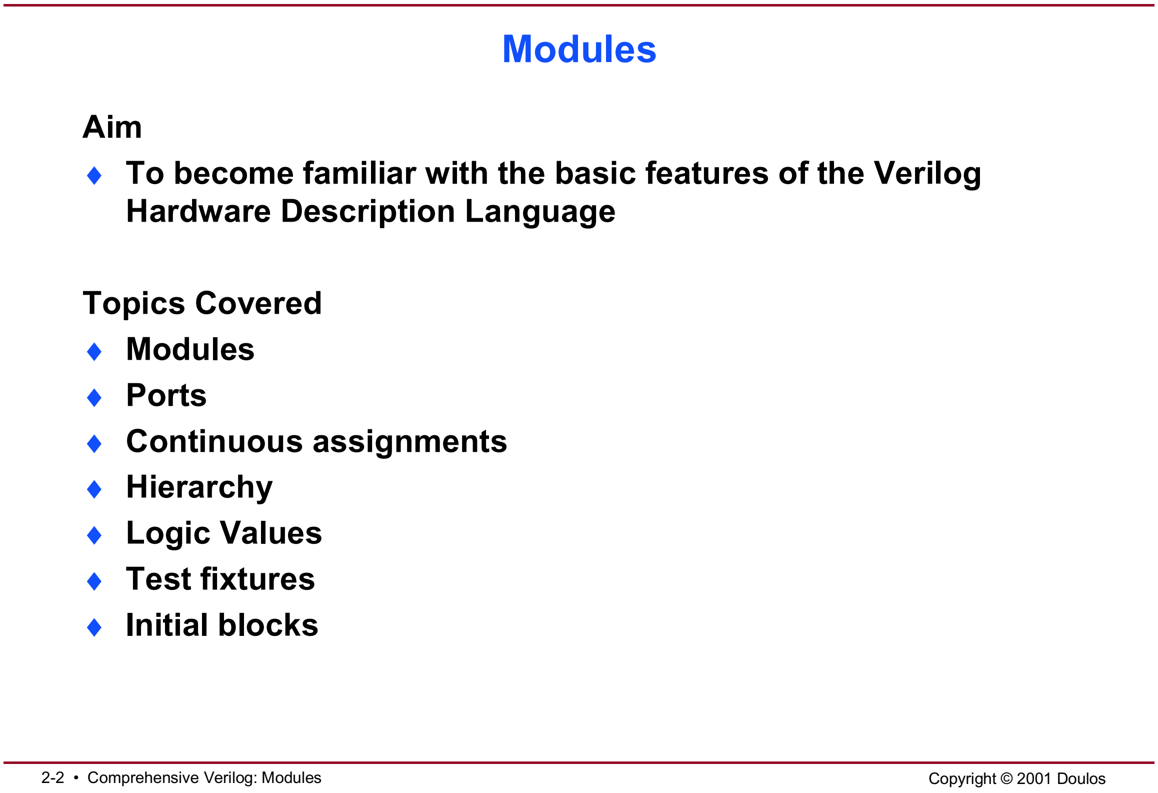
1-15 • Comprehensive Verilog: Introduction to Verilog

Notes:

The Internet is a useful source of information about Verilog related topics. These websites and news groups are given as useful starting points.

Module 2
Modules

Modules



Modules

Aim

- To become familiar with the basic features of the Verilog Hardware Description Language

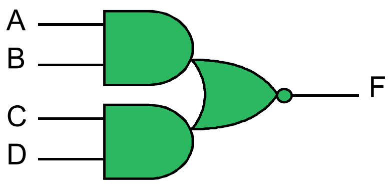
Topics Covered

- Modules
- Ports
- Continuous assignments
- Hierarchy
- Logic Values
- Test fixtures
- Initial blocks

Notes:

Modules and Ports

Modules and Ports



```
// An and-or-invert gate

module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

// The function of the AOI module is described here...

endmodule
```

Notes:

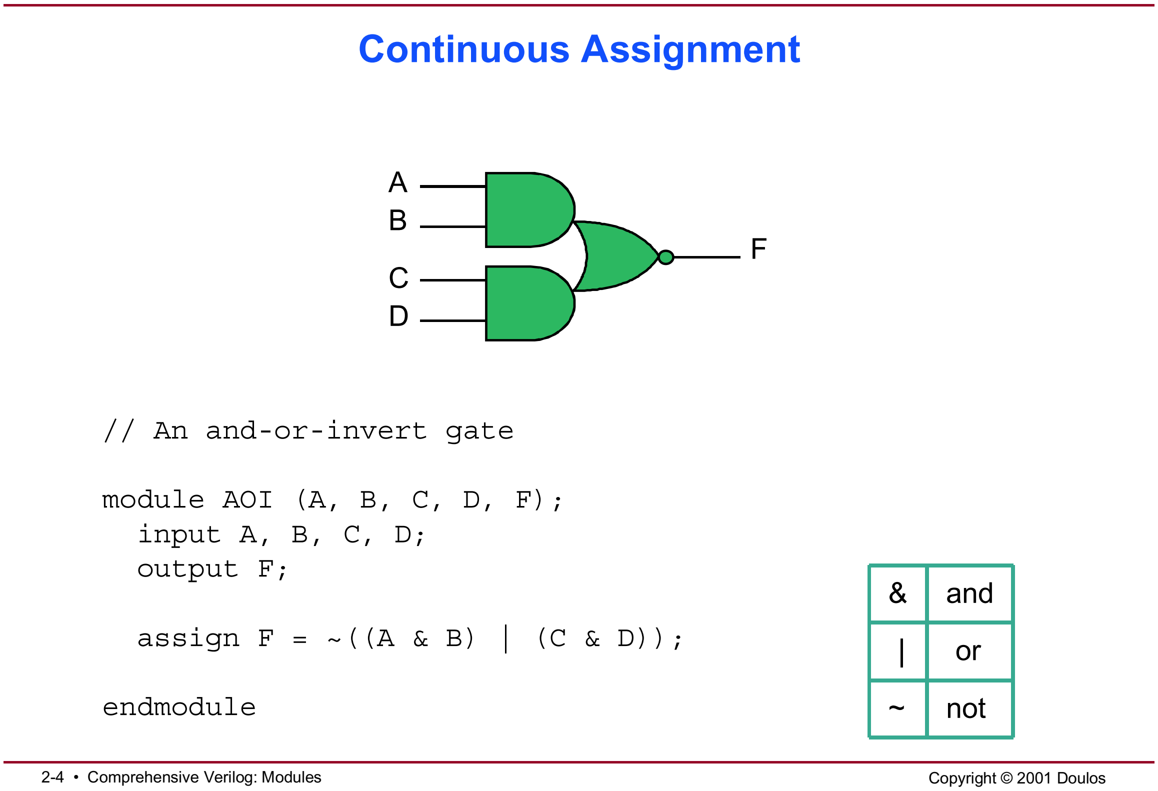
A module represents a block of hardware with well defined inputs and outputs and a well defined function.

A module can be considered to have two parts. The port declarations represent the external interface to the module. The rest of the module consists of an internal description - its behavior, its structure, or a mixture of both.

Ports

Following the keyword module is the name of the module (AOI in this example), and a list of port names. A port may correspond to a pin on an IC, an edge connector on a board, or any logical channel of communication with a block of hardware. Each port in the list is then declared in a port declaration. Each port declaration includes the name of one or more ports (for example: A, B,...), and defines the direction that information is allowed to flow through the ports (input, output or inout).

Continuous Assignment



Continuous Assignment

```
// An and-or-invert gate

module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

  assign F = ~((A & B) | (C & D));

endmodule
```

&	and
|	or
~	not

2-4 • Comprehensive Verilog: Modules

Copyright © 2001 Doulos

Notes:

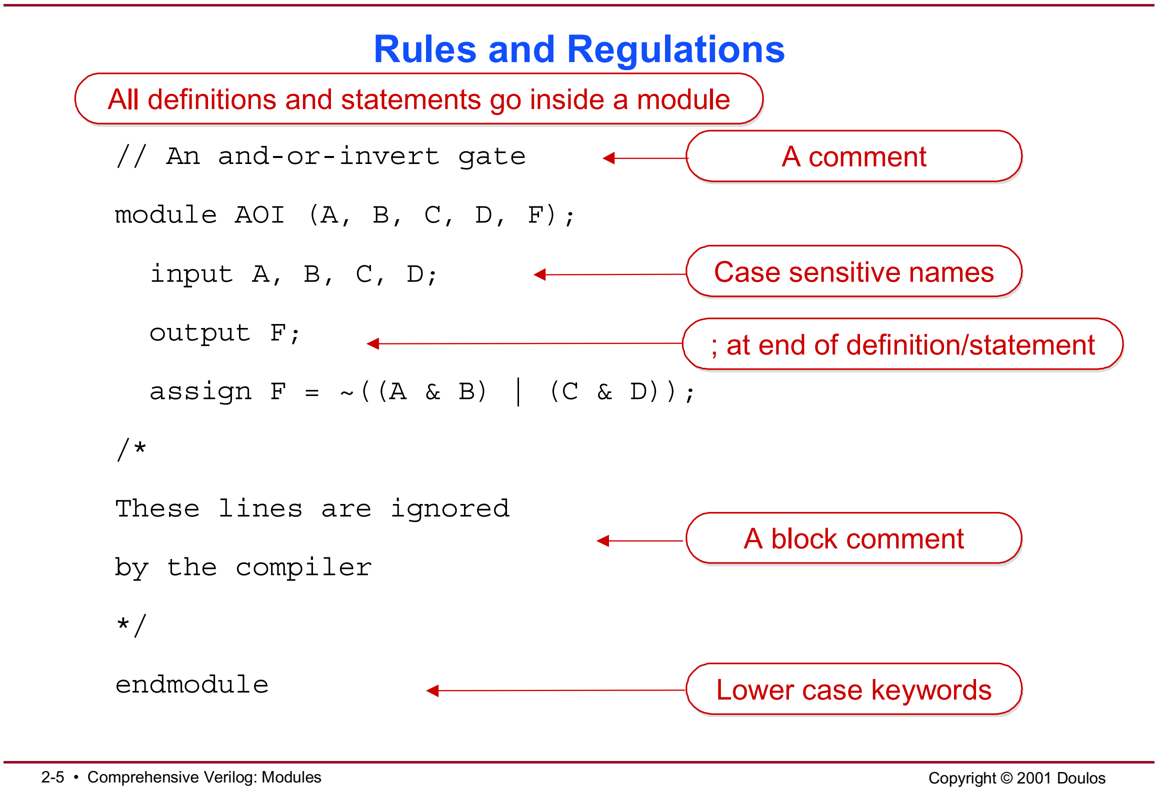
assign

This module contains a continuous assignment, which describes the function of the module. It is called continuous because the left hand side is continuously updated to reflect the value of the expression on the right hand side. During simulation, the continuous assignment is executed whenever one of the four ports A, B, C or D on the right hand side changes value.

Operator

The continuous assignment in this example uses the three operators: & | and ~ which mean bitwise and, or and not respectively. Operators and expressions will be discussed more fully later on.

Rules and Regulations



Rules and Regulations

All definitions and statements go inside a module

```
// An and-or-invert gate
module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;
  assign F = ~((A & B) | (C & D));
/*
These lines are ignored
by the compiler
*/
endmodule
```

A comment

Case sensitive names

; at end of definition/statement

A block comment

Lower case keywords

2-5 • Comprehensive Verilog: Modules

Notes:

Here's a summary of the rules and regulations for the syntax we've seen so far:

Keywords in Verilog are written in lower case. User defined names can be lower or upper case or a mixture of the two, but are case sensitive.

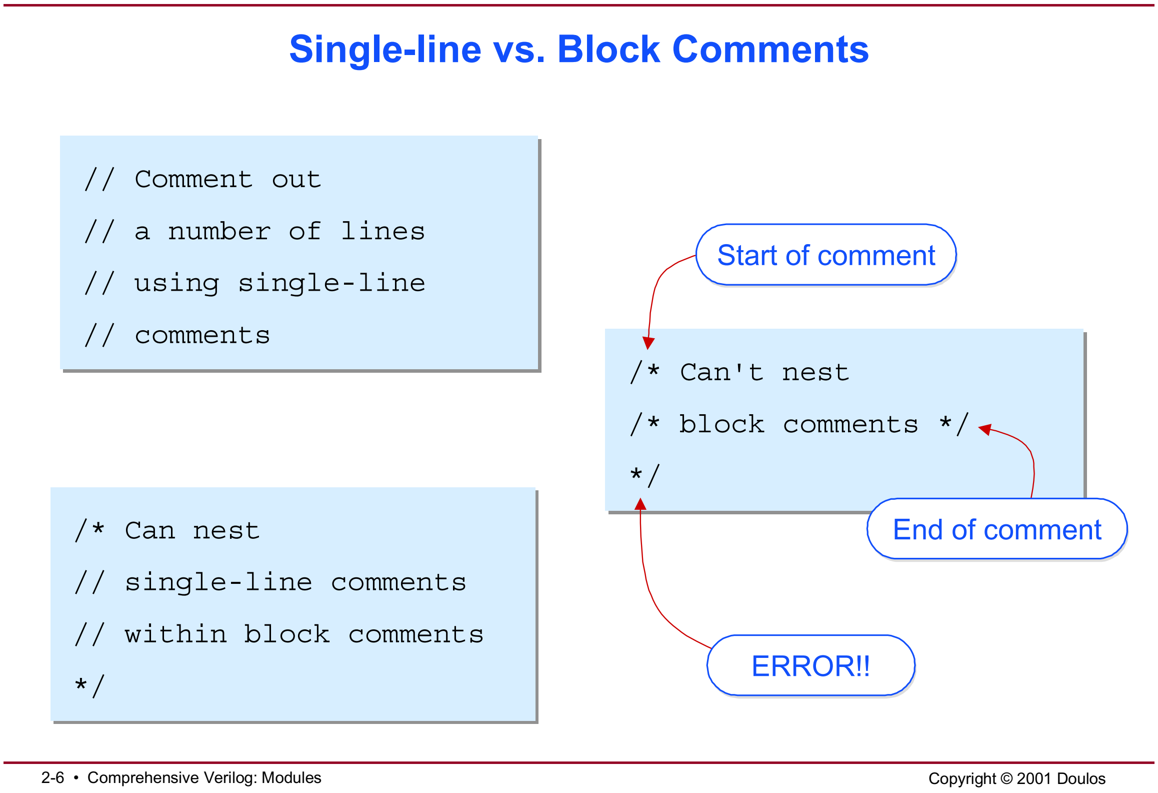
Every definition and statement ends with a semicolon. It is best to write one statement per line, and indent the statements to show the structure of the code.

The characters // mark the beginning of a comment. The rest of the line is ignored.

Block comment

A block comment begins with /* and ends with */ and may span several lines.

Single-line vs. Block Comments



Single-line vs. Block Comments

```
// Comment out
// a number of lines
// using single-line
// comments
```

```
/* Can nest
// single-line comments
// within block comments
*/
```

Start of comment

```
/* Can't nest
/* block comments */
*/
```

End of comment

ERROR!!

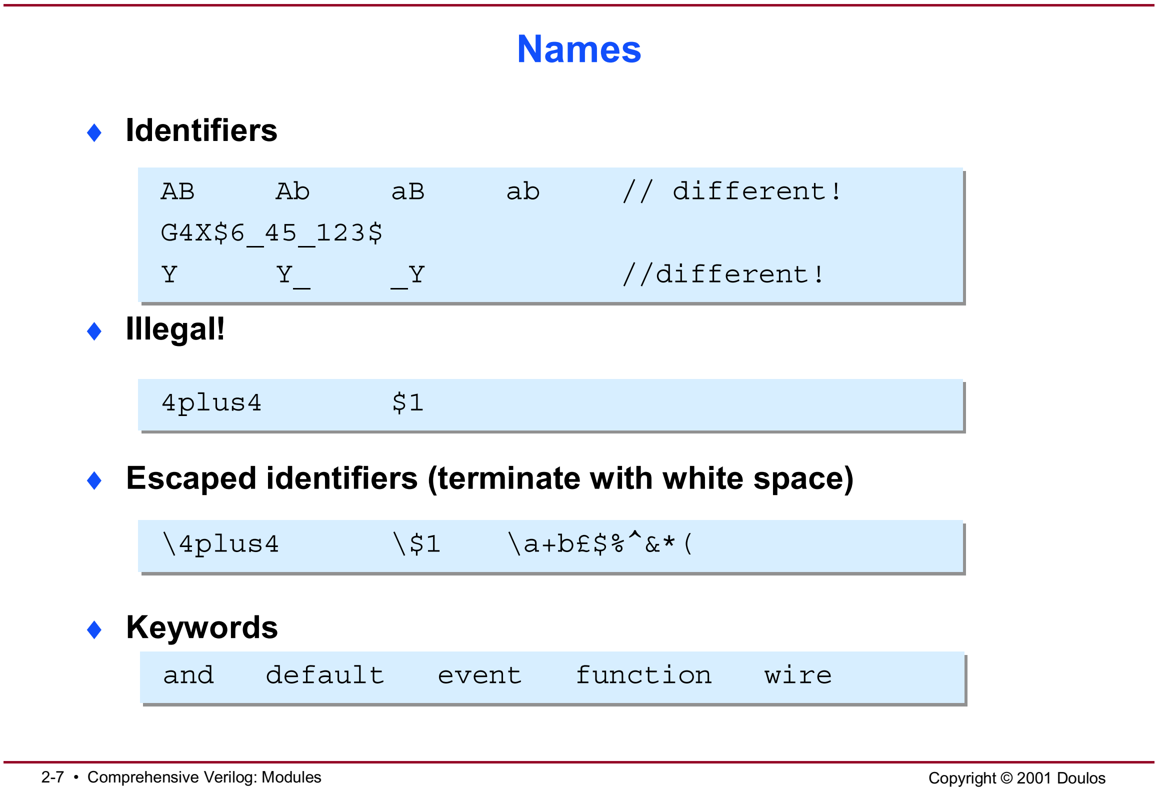
2-6 • Comprehensive Verilog: Modules

Copyright © 2001 Doulos

Notes:

Generally, single line comments (//) should be used in preference to block comments (/* … */), even if several lines need to be commented. Block comments should only be used for commenting very large numbers of lines (e.g. to comment out an entire module). This is because block comments cannot be nested: the end of the innermost block comment (*/) is interpreted as being the end of the outermost comment! Single line comments within block comments are just considered part of the (block) comment.

Names



Names

- **Identifiers**

```
AB      Ab      aB      ab      // different!
G4X$6_45_123$
Y       Y_      _Y              //different!
```

- **Illegal!**

```
4plus4         $1
```

- **Escaped identifiers (terminate with white space)**

```
\4plus4        \$1     \a+b£$%^&*(
```

- **Keywords**

```
and    default    event    function    wire
```

Notes:

Identifiers

Names of modules, ports etc. are properly called identifiers. Identifiers can be of any length, and consist of letters, digits, underscores (_) and dollars ($). The first character must be a letter or an underscore.

Case

The case of identifiers is significant, so two identifiers that differ only in case do in fact name different items.

Escaped identifier

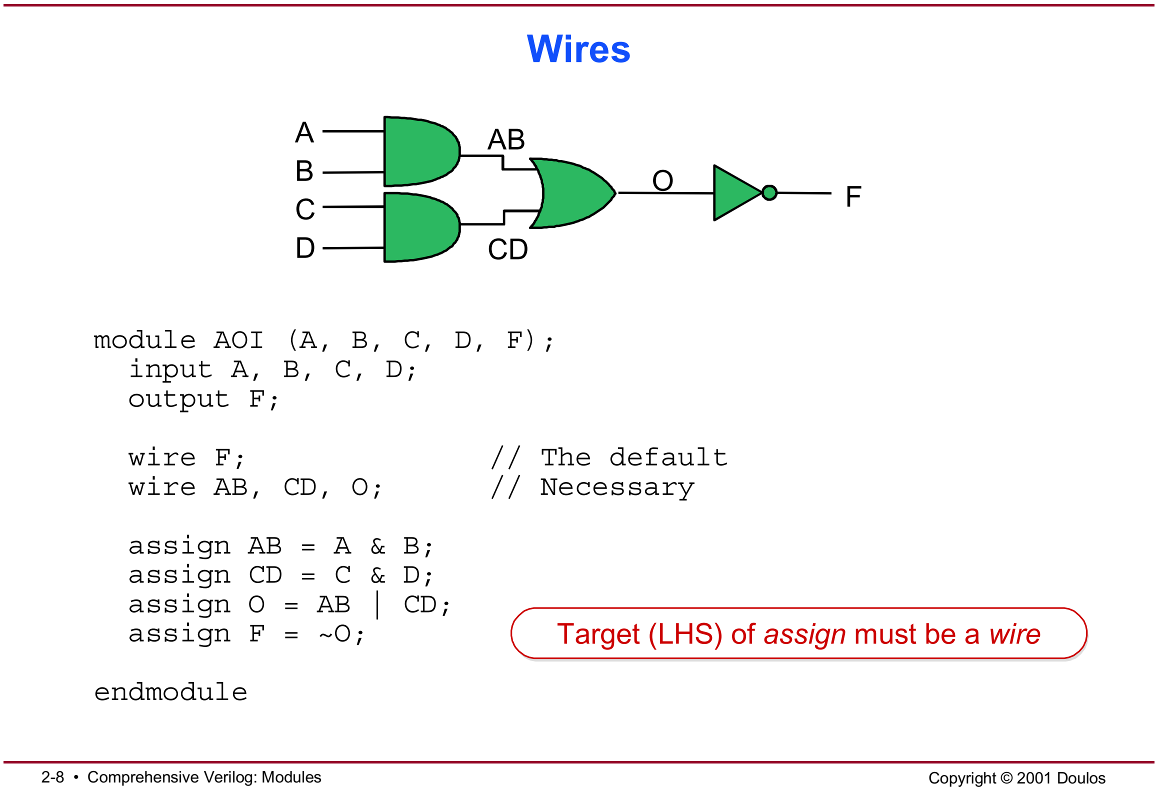
An escaped identifier begins with a backslash (\) and ends with a white space (space, tab, or new line). They may contain any characters, except white space. Escaped identifiers are intended for use by tools that write Verilog, but that have different naming conventions. Examples are schematic editors and synthesis tools.

Reserved identifiers

There are a large number of reserved identifiers that cannot be declared as names. The complete list is as follows....

and	endfunction	join	pull0	supply0	wire
always	endprimitive	large	pull1	supply1	wor
assign	endmodule	macromodule	rcmos	table	xnor
begin	endspecify	medium	real	task	xor
buf	endtable	module	realtime	tran	
bufif0	endtask	nand	reg	tranif0	
bufif1	event	negedge	release	tranif1	
case	for	nor	repeat	time	
casex	force	not	mmos	tri	
casez	forever	notif0	rpmos	trian	
cmos	fork	notif1	rtran	trior	
deassign	function	nmos	rtranif0	trireg	
default	highz0	or	rtranif1	tri0	
defparam	highz1	output	scalared	tri1	
disable	if	parameter	small	vectored	
edge	ifnone	pmos	specify	wait	
else	initial	posedge	specparam	wand	
edge	inout	primitive	strength	weak0	
end	input	pulldown	strong0	weak1	
endcase	integer	pullup	strong1	while	

Wires



Wires

```
module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

  wire F;              // The default
  wire AB, CD, O;      // Necessary

  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;

endmodule
```

Target (LHS) of *assign* must be a *wire*

2-8 • Comprehensive Verilog: Modules

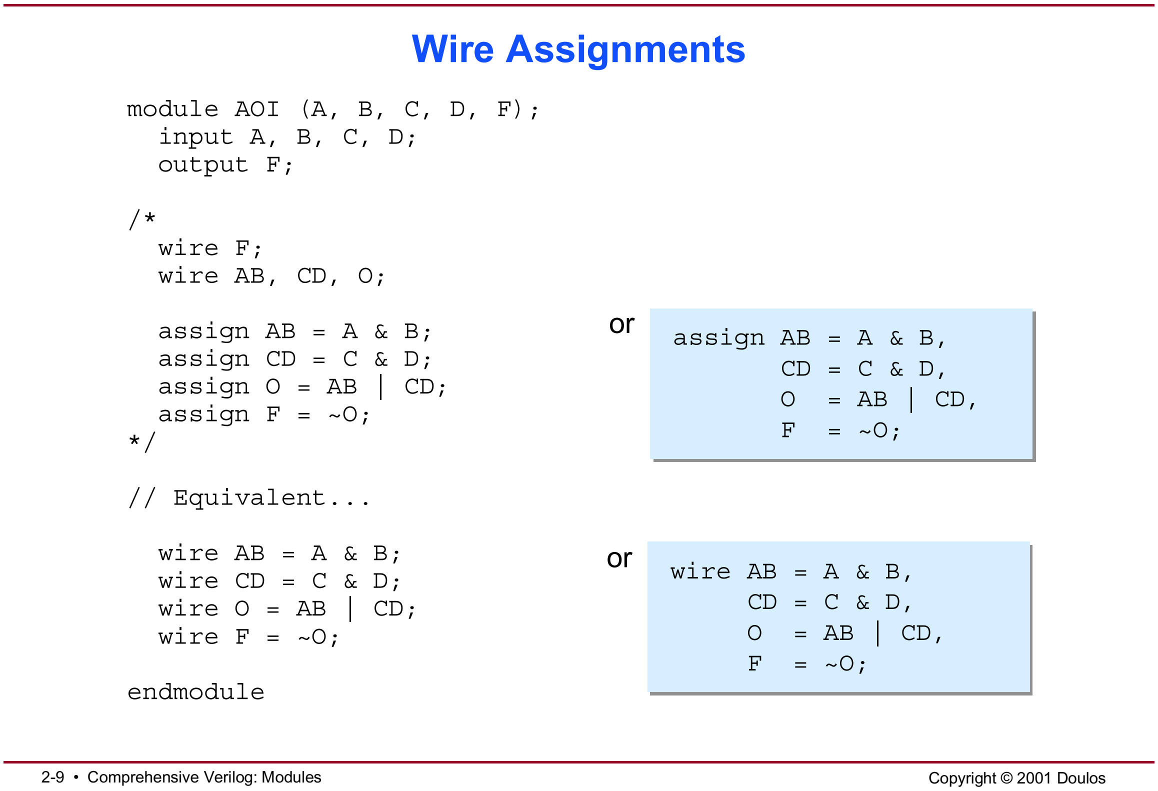
Notes:

The module we have been looking at is simple enough for the output to be a simple function of the inputs using a single continuous assignment. Usually, modules are more complex than this, and internal connections are required.

To make a continuous assignment to an internal signal, the signal must first be declared as a wire. A wire declaration looks like a port declaration, with a type (wire), an optional vector width, and a name or list of names. (We shall be looking at vectors later.)

Ports default to being wires, so the declaration of wire F opposite is optional. The other declarations are required.

Wire Assignments



Wire Assignments

```
module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;

/*
  wire F;
  wire AB, CD, O;

  assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;
*/

// Equivalent...

  wire AB = A & B;
  wire CD = C & D;
  wire O = AB | CD;
  wire F = ~O;

endmodule
```

or

```
assign AB = A & B,
       CD = C & D,
       O  = AB | CD,
       F  = ~O;
```

or

```
wire AB = A & B,
     CD = C & D,
     O  = AB | CD,
     F  = ~O;
```

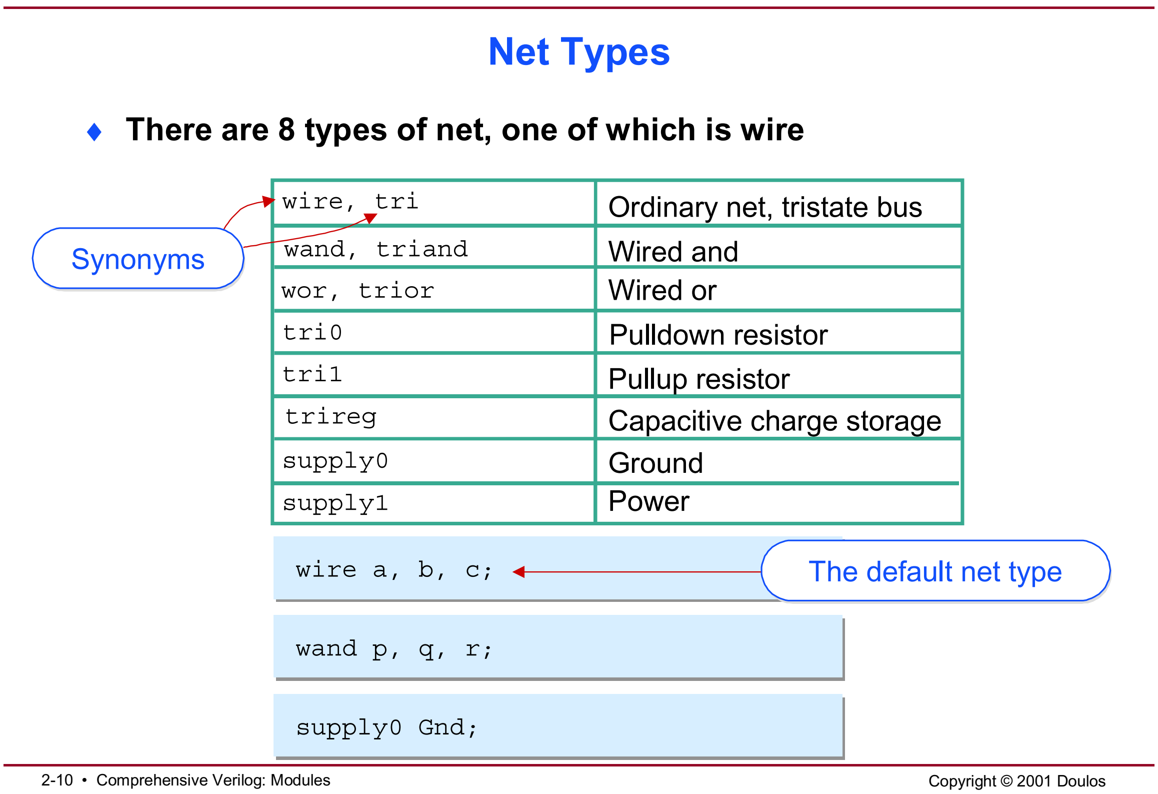
Notes:

Wires can be declared and continuously assigned in a single statement - a wire assignment. This is a shortcut which saves declaring and assigning a wire separately. There are no advantages or disadvantages between the two methods other than the obvious difference that wire assignments reduce the size of the text.

A delay in a wire assignment is equivalent to a delay in the corresponding continuous assignment, not a delay on the wire. Thus it could be necessary to separate the wire declaration from the continuous assignment to put the delay onto the wire rather than the assignment. Note that this is a subtle point that you are unlikely to encounter in practice!

This slide also illustrates a feature of the Verilog syntax where there is more than one declaration, instance or continuous assignment and the objects being declared, instanced or assigned are the same type. In this case, you can either have several separate statements or assignments (separated by semicolons) or you can make several declarations or assignments (separated by commas) in a single statement.

Net Types



Net Types

- There are 8 types of net, one of which is wire

`wire, tri`	Ordinary net, tristate bus
`wand, triand`	Wired and
`wor, trior`	Wired or
`tri0`	Pulldown resistor
`tri1`	Pullup resistor
`trireg`	Capacitive charge storage
`supply0`	Ground
`supply1`	Power

Synonyms

```
wire a, b, c;
```

The default net type

```
wand p, q, r;
```

```
supply0 Gnd;
```

2-10 • Comprehensive Verilog: Modules

Copyright © 2001 Doulos

Notes:

Nets are one of the main three data types in Verilog. (The others are registers and Parameters. We shall meet these later.) Nets represent electrical connections between points in a circuit.

In Verilog, a wire is a type of net, but there are also seven other net types, each modeling a different electrical phenomenon. Wire is the default net type (undefined names used in a connection list default to being one bit wires), and in all probability, 99% of the nets in your Verilog descriptions will be wires. However, the remaining types of net are important for modeling certain cases.

wire, tri

The wire (synonymous with tri) models either a point-to-point electrical connection, or a tristate bus with multiple drivers. In the event of a bus conflict, the value of the wire will become 'bx.

wand, wor

The wand (synonym triand) and wor (synonym trior) behave like AND and OR gates respectively when there exists more than one driver on the net.

tri0, tri1

The tri0 and tri1 nets behave like wires with a pulldown or pullup resistor attached, i.e. when these nets are not being driven, the net floats to 0 or 1 respectively. In the same situation, a wire would have the value z.

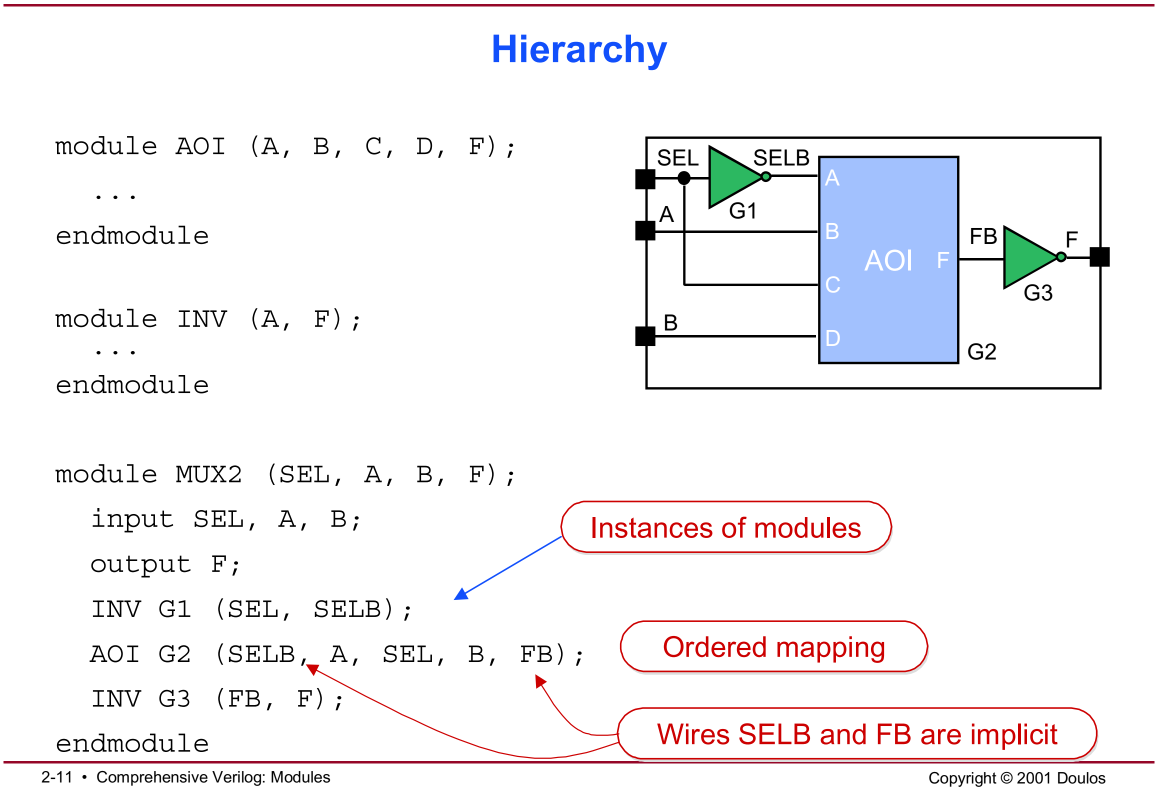
trireg

The trireg net models capacitive charge storage and charge decay. When a trireg net is not being driven, it retains its previous value. trireg nets are used in switch-level modelling.

supply0, supply1

The supply0 and supply1 nets represent ground and power rails respectively.

Hierarchy



Hierarchy

```
module AOI (A, B, C, D, F);
  ...
endmodule

module INV (A, F);
  ...
endmodule

module MUX2 (SEL, A, B, F);
  input SEL, A, B;
  output F;
  INV G1 (SEL, SELB);
  AOI G2 (SELB, A, SEL, B, FB);
  INV G3 (FB, F);
endmodule
```

Instances of modules

Ordered mapping

Wires SELB and FB are implicit

2-11 • Comprehensive Verilog: Modules

Notes:

Module Instance

Modules can reference other modules to form a hierarchy. Here we see a 2:1 multiplex or consisting of an AOI gate and two inverters. The MUX2 module contains references to each of the lower level modules, and describes the interconnections between them. In Verilog jargon, a reference to a lower level module is called a module instance.

Each instance is an independent, concurrently active copy of a module. Each module instance consists of the name of the module being instanced (e.g. AOI of INV), an instance name (unique to that instance within the current module) and a port connection list.

Note that an instance name is required. It is used to differentiate between multiple instances of the same module, as is the case with INV.

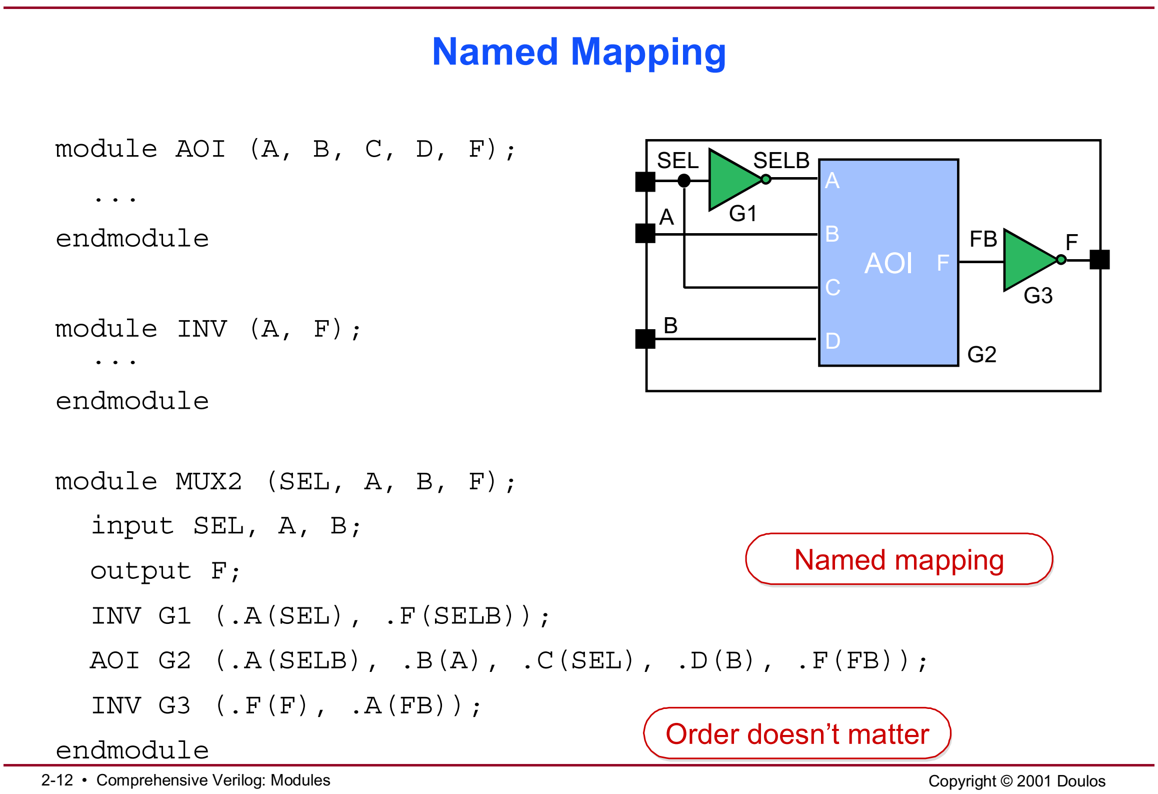
Ordered mapping

The module port connections are given in order, the order being derived from the first line of the module being instanced. The first item in the list is connected to the first port on the module, and so on. This is known as ordered mapping.

Implicit Wires

Any names that are used in a port connection list but are not declared elsewhere get declared implicitly as one bit wires. This is very important - and very error prone!

Named Mapping



Named Mapping

```
module AOI (A, B, C, D, F);
  ...
endmodule

module INV (A, F);
  ...
endmodule

module MUX2 (SEL, A, B, F);
  input SEL, A, B;
  output F;
  INV G1 (.A(SEL), .F(SELB));
  AOI G2 (.A(SELB), .B(A), .C(SEL), .D(B), .F(FB));
  INV G3 (.F(F), .A(FB));
endmodule
```

SEL
SELB
G1
A
B
AOI
A
B
C
D
F
FB
F
G3
G2

Named mapping

Order doesn’t matter

2-12 • Comprehensive Verilog: Modules

Copyright © 2001 Doulos

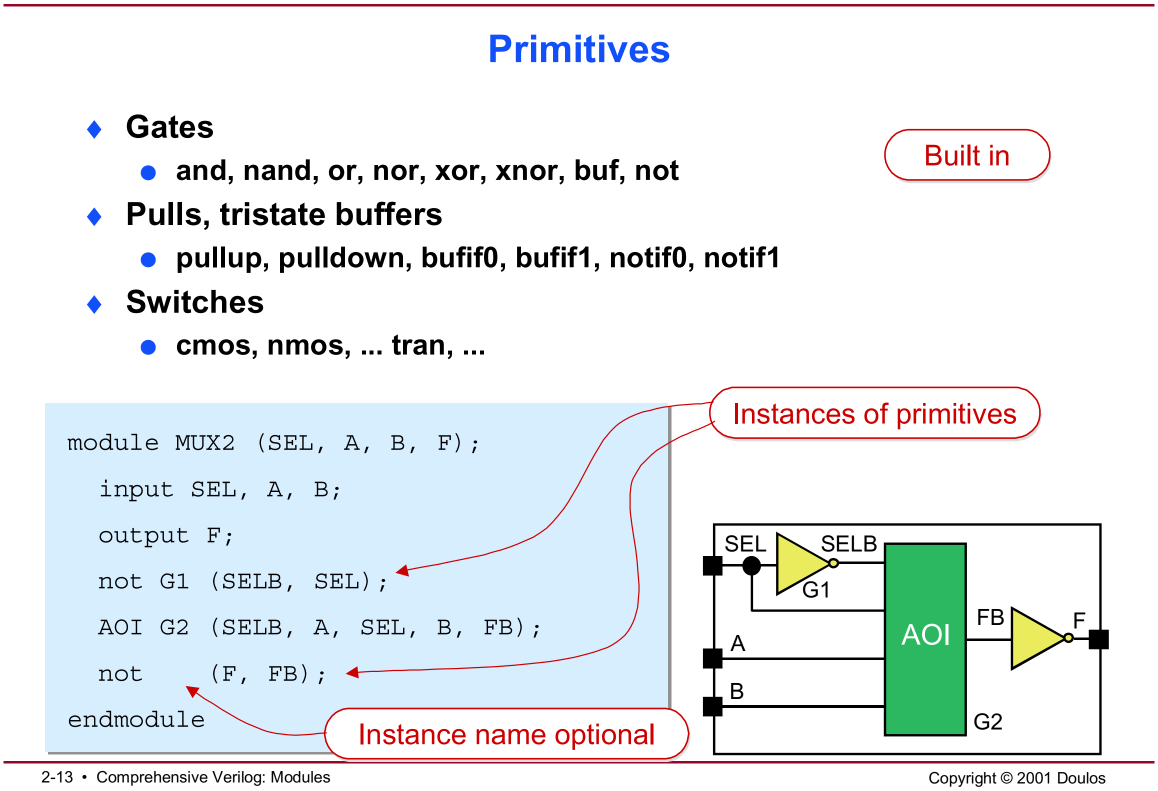
Notes:

Named Mapping

An alternative to listing the ports in the correct order is named mapping. Here, the connections are made by giving the name of the module's port, preceded by a full stop, and supplying the name of the wire or register that is to be connected to that port in brackets. Because the ports' names are given, the order in which they appear is no longer relevant.

Named mapping is often preferred to ordered mapping because it is easier to understand and less error-prone.

Primitives



Primitives

- Gates
  - and, nand, or, nor, xor, xnor, buf, not
- Pulls, tristate buffers
  - pullup, pulldown, bufif0, bufif1, notif0, notif1
- Switches
  - cmos, nmos, ... tran, ...

Built in

```
module MUX2 (SEL, A, B, F);
  input SEL, A, B;
  output F;
  not G1 (SELB, SEL);
  AOI G2 (SELB, A, SEL, B, FB);
  not    (F, FB);
endmodule
```

Instances of primitives

Instance name optional

2-13 • Comprehensive Verilog: Modules

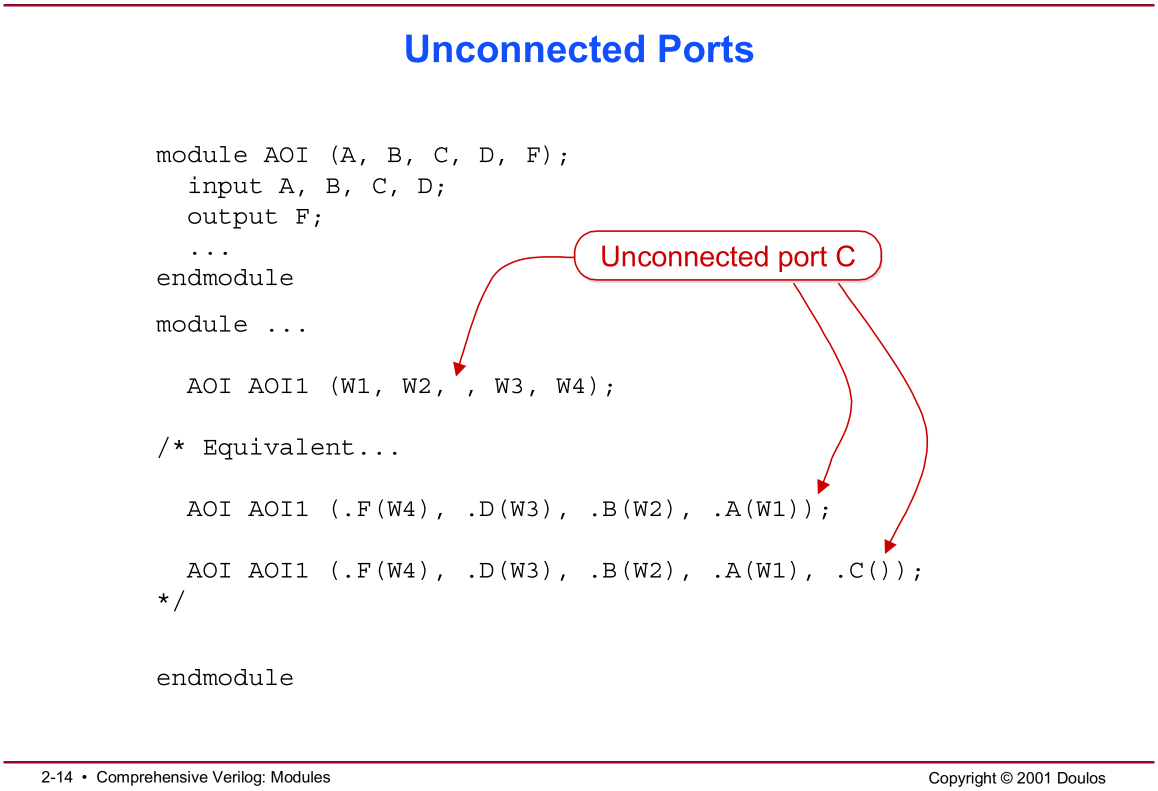
Notes:

In the previous example we have instaned a module INV. In fact, we do not need to write our own module for an inverter: Verilog includes a small number of built in primitives or gates. These include models of simple logic gates, tristate buffers, pullup and pulldown resistors, and a number of unidirectional and bidirectional switches.

Primitives are used extensively in detailed gate-level and switch-level models: for example, when modelling libraries and full-custom ASICs. It is sometimes appropriate to use primitives at other times too, although continuous assignments are usually preferred.

Primitives are instanced in the same way as modules, except that an instance name is not required. (You can include one if you want.) Also, named mapping cannot be used for primitives: you have to know the ordering of the ports; to help you, the outputs always come before the inputs.

Unconnected Ports



Unconnected Ports

```
module AOI (A, B, C, D, F);
  input A, B, C, D;
  output F;
  ...
endmodule

module ...

  AOI AOI1 (W1, W2, , W3, W4);

/* Equivalent...

  AOI AOI1 (.F(W4), .D(W3), .B(W2), .A(W1));

  AOI AOI1 (.F(W4), .D(W3), .B(W2), .A(W1), .C());
*/

endmodule
```

Unconnected port C

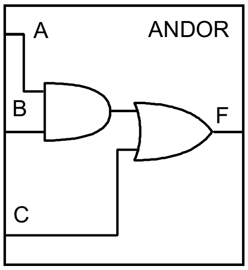
Notes:

If you are using ordered mapping, the ports of a module instance can be left unconnected by omitting them from the connection list. Simply leave the position blank by inserting two adjacent commas.

For named mapping, a port can be left unconnected in one of two ways: (i) by omitting the port altogether; (ii) by including the port with nothing in the brackets (the brackets must be present).

Quiz 1

Quiz 1



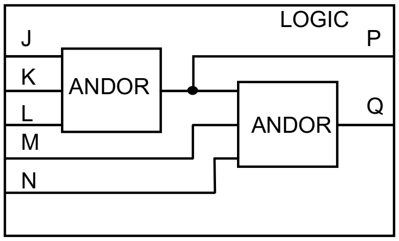
```
module ANDOR (



endmodule
```

Notes:

Complete the Verilog code to describe this simple circuit.

Quiz 2

Quiz 2



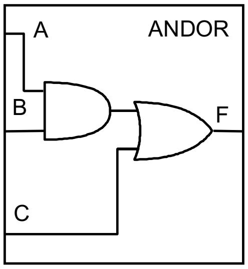
```
module LOGIC (



endmodule
```

Notes:

Now complete the Verilog code for this circuit, which uses the circuit on the previous page.

Quiz 1 – Solutions

Quiz 1 – Solution



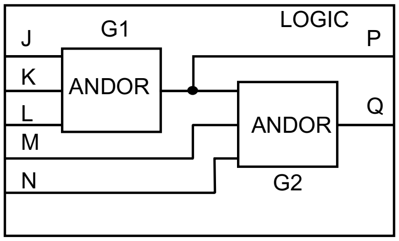
```
module ANDOR (A, B, C, F);
  input  A, B, C;
  output F;

  assign F = (A & B) | C;
endmodule
```

```
module ANDOR (A, B, C, F);
  input  A, B, C;
  output F;
  wire AB;

  assign AB = A & B;
  assign  F = AB | C;
endmodule
```

Notes:

Quiz 2 – Solutions

Quiz 2 – Solution

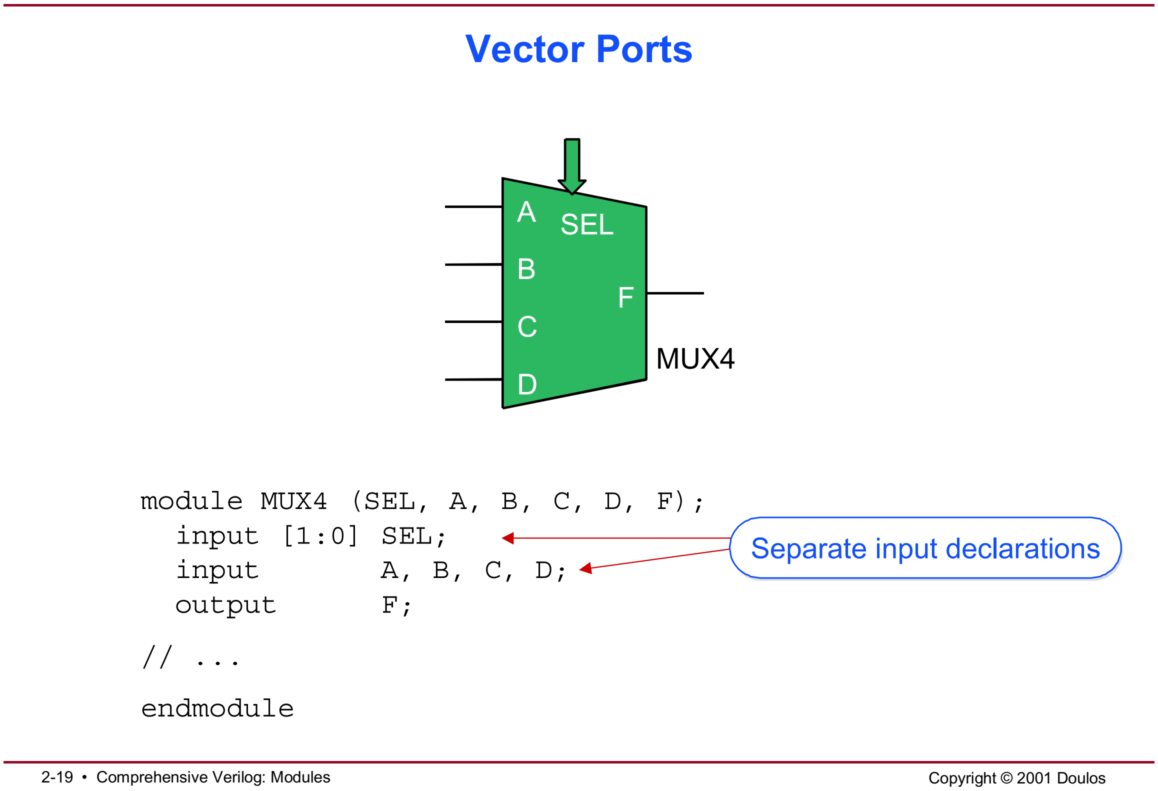


```
module LOGIC (J, K, L, M, N, P, Q);
  input  J, K, L, M, N;
  output P, Q;

  ANDOR G1 (J, K, L, P);
  ANDOR G2 (P, M, N, Q);
endmodule
```

```
module LOGIC (J, K, L, M, N, P, Q);
  input  J, K, L, M, N;
  output P, Q;

  ANDOR G1 (.A(J), .B(K), .C(L), .F(P));
  ANDOR G2 (.A(P), .B(M), .C(N), .F(Q));
endmodule
```

Notes:

Vector Ports



Vector Ports

```
module MUX4 (SEL, A, B, C, D, F);
  input  [1:0] SEL;
  input        A, B, C, D;
  output       F;
// ...
endmodule
```

Separate input declarations

Notes:

Ports (and wires) can represent buses or vectors as well as single bits. In the example above, the port SEL is a two bit bus, with the most significant bit (MSB) numbered 1 and the least significant bit (LSB) numbered 0.

Equally, we could have declared SEL as follows:

input [0:1] SEL;

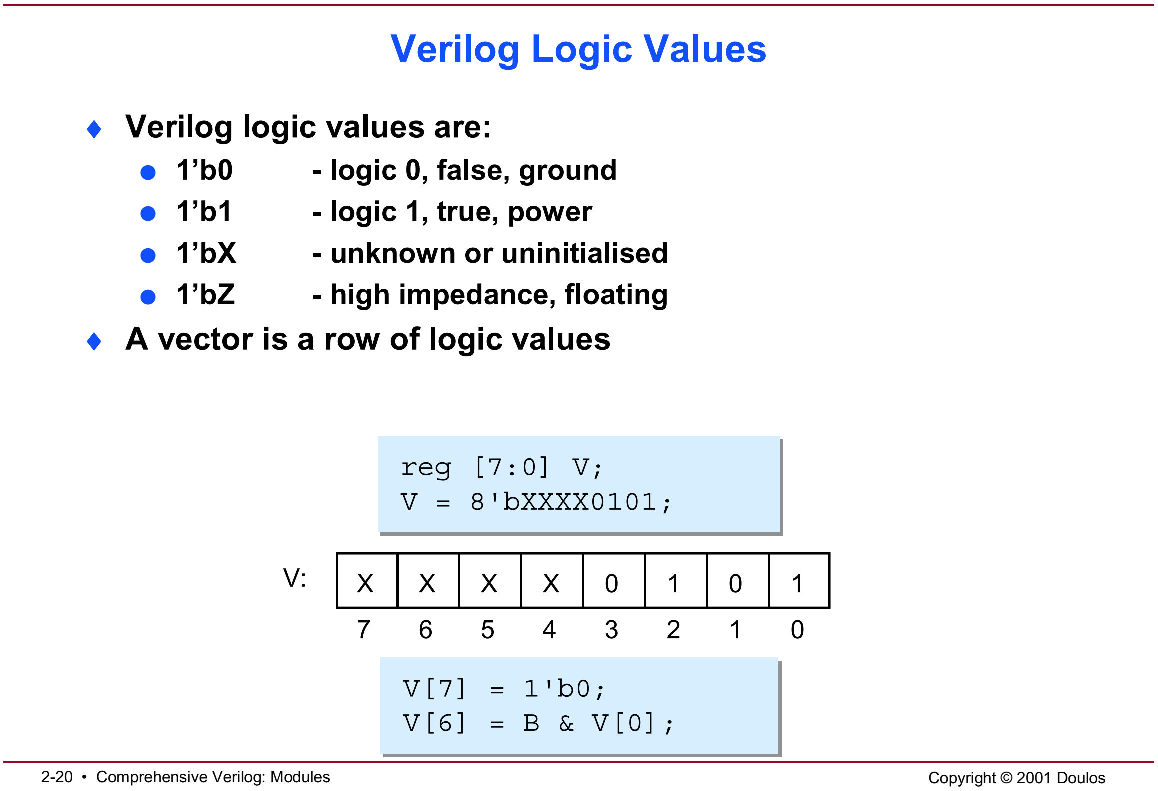
or

input [1:2] SEL;

etc.

In each case, the left-hand value of the range numbers the MSB. The actual values in the range declaration do not matter. Later, we will see that the value of a vector is interpreted as an unsigned number, which is why it is important to understand that the left-hand number in the declaration refers to the most significant bit.

Verilog Logic Values



Verilog Logic Values

- Verilog logic values are:
  - **1’b0** - logic 0, false, ground
  - **1’b1** - logic 1, true, power
  - **1’bX** - unknown or uninitialised
  - **1’bZ** - high impedance, floating
- A vector is a row of logic values

```
reg [7:0] V;
V = 8'bXXXX0101;
```

V:	X	X	X	X	0	1	0	1
	7	6	5	4	3	2	1	0

```
V[7] = 1'b0;
V[6] = B & V[0];
```

2-20 • Comprehensive Verilog: Modules

Notes:

In Verilog, there are four built-in logic values. These are 0 (logic 0 or false), 1 (logic 1 or true), X (unknown) and Z (high impedance or floating).

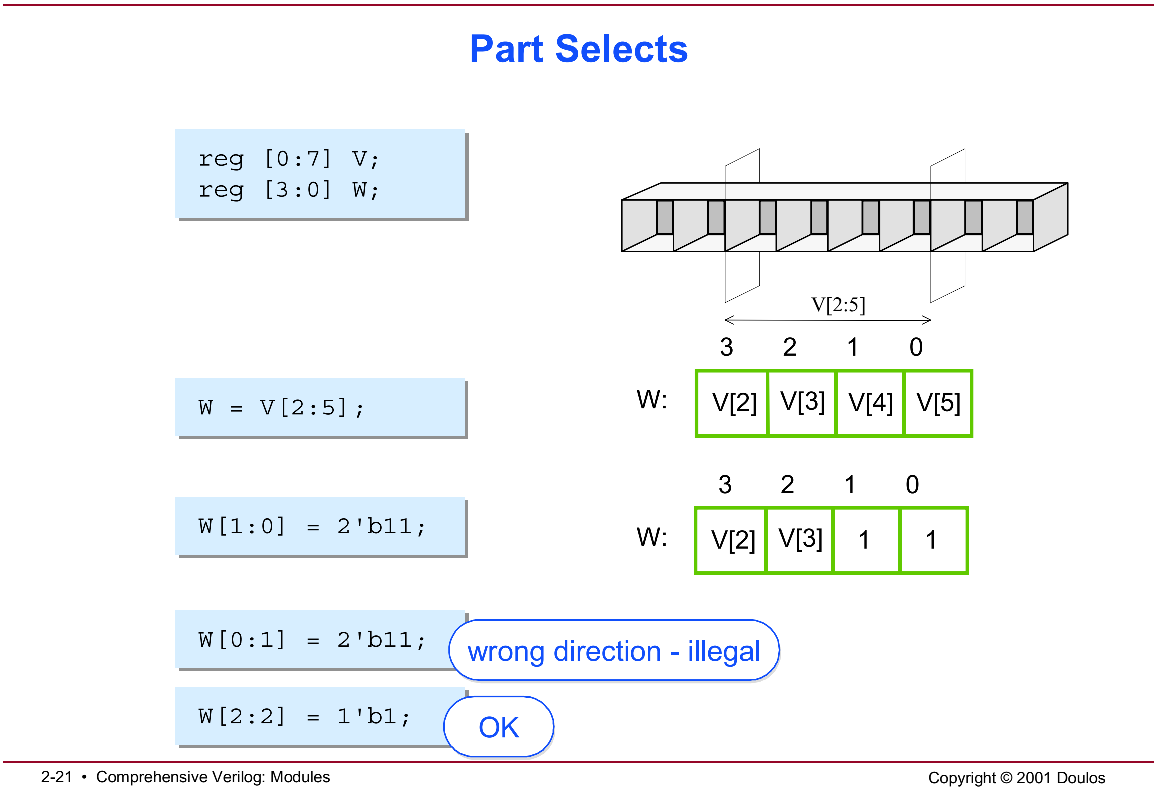
A single bit value is represented as a single bit integer literal. This consists of the value 1 (indicating a single bit) followed by 'b ('b stands for "binary") and the value (0, 1, X or Z), for example 1'bX.

The value of a vector in Verilog is simply a row of logic values (0, 1, X and Z). The value is preceded by the number of bits in the vector and the base. For example, 8'bXXXX0101 is an eight bit binary value. Note that the name of a vector denotes the entire vector (e.g. V in the example opposite).

Individual bits of a vector can be assigned values or used in an expression by using square brackets to indicate the bit. For example, V[7] refers to bit 7 of V (in this case the MSB). This is sometimes called a bit select.

The declaration reg [7:0] V; that appears on this slide will be explained shortly.

Part Selects



Part Selects

```
reg [0:7] V;
reg [3:0] W;
```

```
W = V[2:5];
```

```
W[1:0] = 2'b11;
```

```
W[0:1] = 2'b11;
```

wrong direction - illegal

```
W[2:2] = 1'b1;
```

OK

2-21 • Comprehensive Verilog: Modules

Copyright © 2001 Doulos

Notes:

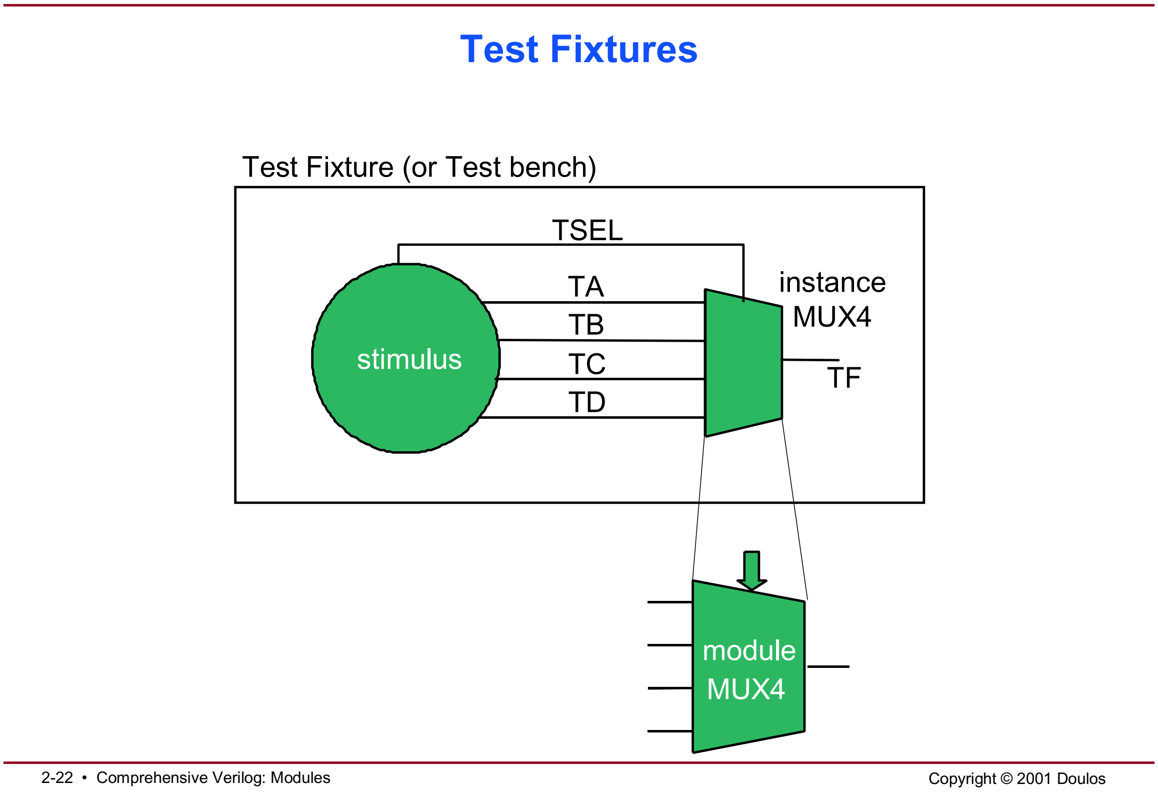
A *part select* denotes part of vector with contiguous index values e.g. V[2:5]. Part selects can be used to read or assign part of a vector.

Vectors are assigned bit by bit from left to right, so the assignment W = V[2:5]; opposite leaves W[3]=V[2] etc. The assignment W[1:0] = 2'b11; leaves W[3] and W[2] unchanged.

Note that the range of a vector can be specified as an ascending range (e.g. [0:7]) or a descending range (e.g. [3:0]). However, the range of a part select must have the same direction as the range in the original declaration. So, the expression W[0:1] is illegal, because W was declared with a descending range.

A part select denoting a single bit - for example, W[2:2] - is allowed.

Test Fixtures



Test Fixtures

Test Fixture (or Test bench)

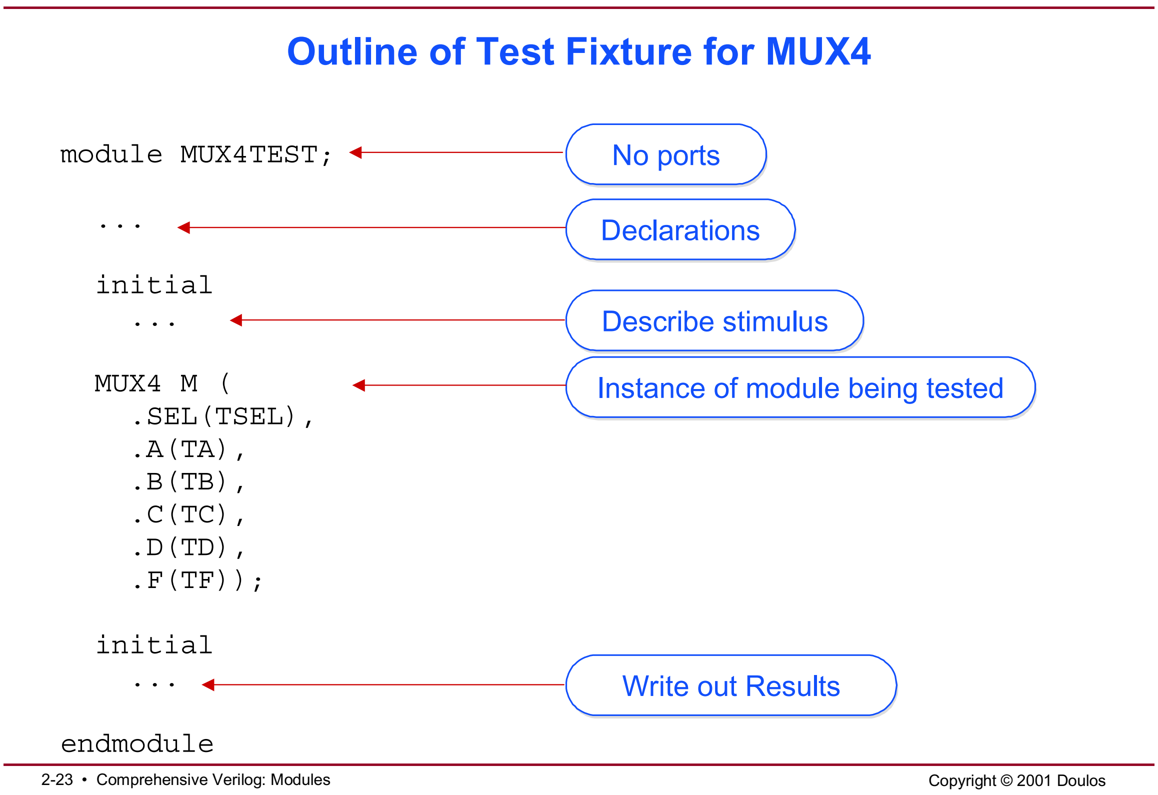
2-22 • Comprehensive Verilog: Modules

Notes:

A test fixture is a Verilog module that describes the simulation environment for a Verilog design. A test fixture usually includes code to generate the test vectors and analyze the results (or compare the results with the expected output). A text fixture will also contain an instance of the module being tested.

The term test fixture is the one commonly used in the Verilog world. The term test bench is also used and has the same meaning.

Outline of Test Fixture for MUX4



Outline of Test Fixture for MUX4

```
module MUX4TEST;

  ...

  initial
    ...

  MUX4 M (
    .SEL(TSEL),
    .A(TA),
    .B(TB),
    .C(TC),
    .D(TD),
    .F(TF));

  initial
    ...

endmodule
```

No ports

Declarations

Describe stimulus

Instance of module being tested

Write out Results

2-23 • Comprehensive Verilog: Modules

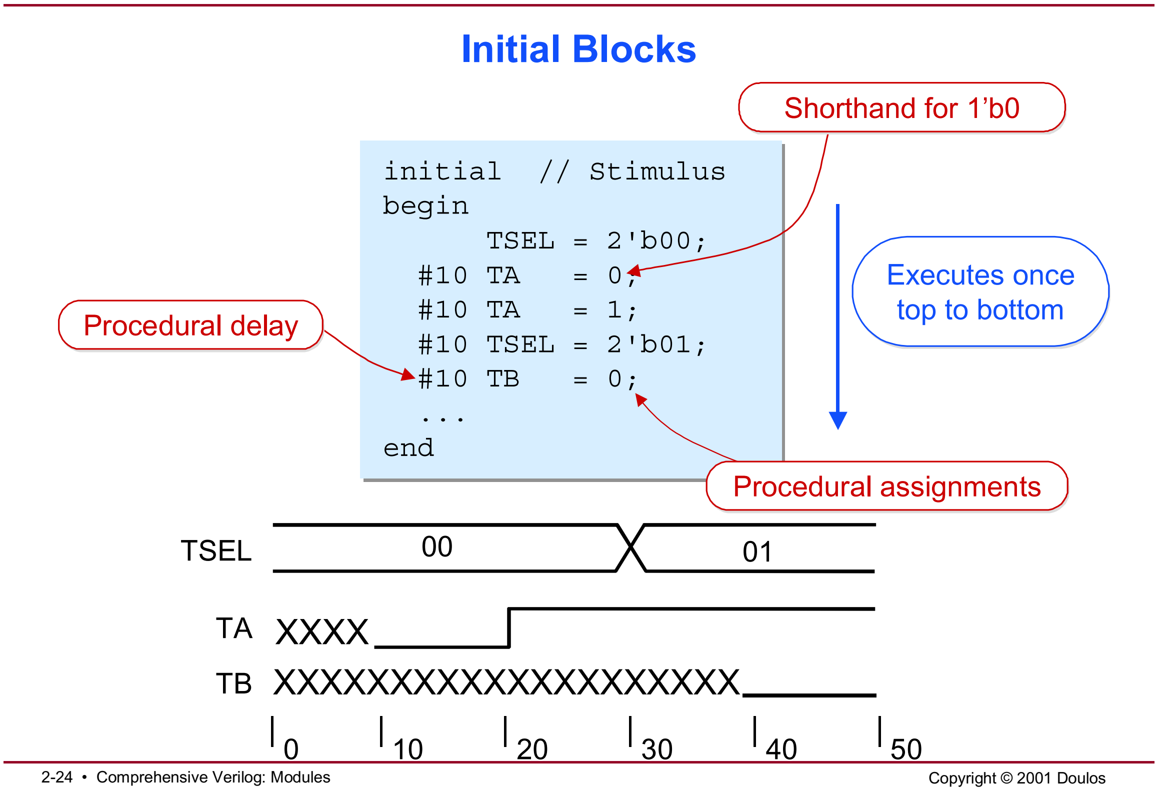
Copyright © 2001 Doulos

Notes:

Here is an outline of the Verilog code of a test fixture to test a module MUX4. It contains two initial statements, one for applying the test stimulus and the other for writing a table of results, and an instance of the module being tested. Initial statements are described in the following pages.

A characteristic of test fixtures is that they do not have any ports.

Initial Blocks



Initial Blocks

Shorthand for 1'b0

```
initial   // Stimulus
begin
        TSEL = 2'b00;
   #10 TA    = 0;
   #10 TA    = 1;
   #10 TSEL = 2'b01;
   #10 TB    = 0;
   ...
end
```

Procedural delay

Executes once top to bottom

Procedural assignments

2-24 • Comprehensive Verilog: Modules

Copyright © 2001 Doulos

Notes:

An initial block is one of Verilog's procedural blocks (the other is the always block). Procedural blocks are used to describe behavior, and are used in many different situations. Here, an initial block is being used to described the stimulus, or test patterns for the design.

The initial block starts executing at the start of simulation (time zero), and the statements between begin and end execute in sequence from top to bottom.

#10 is a procedural delay of 10 time units. A procedural delay suspends the execution of the initial block for the given period of time. Each statement may be preceded by none, one or more delays. They are part of the statement, and aren't followed by a semicolon (although one could be inserted):

#10 A = 0;

is the same as

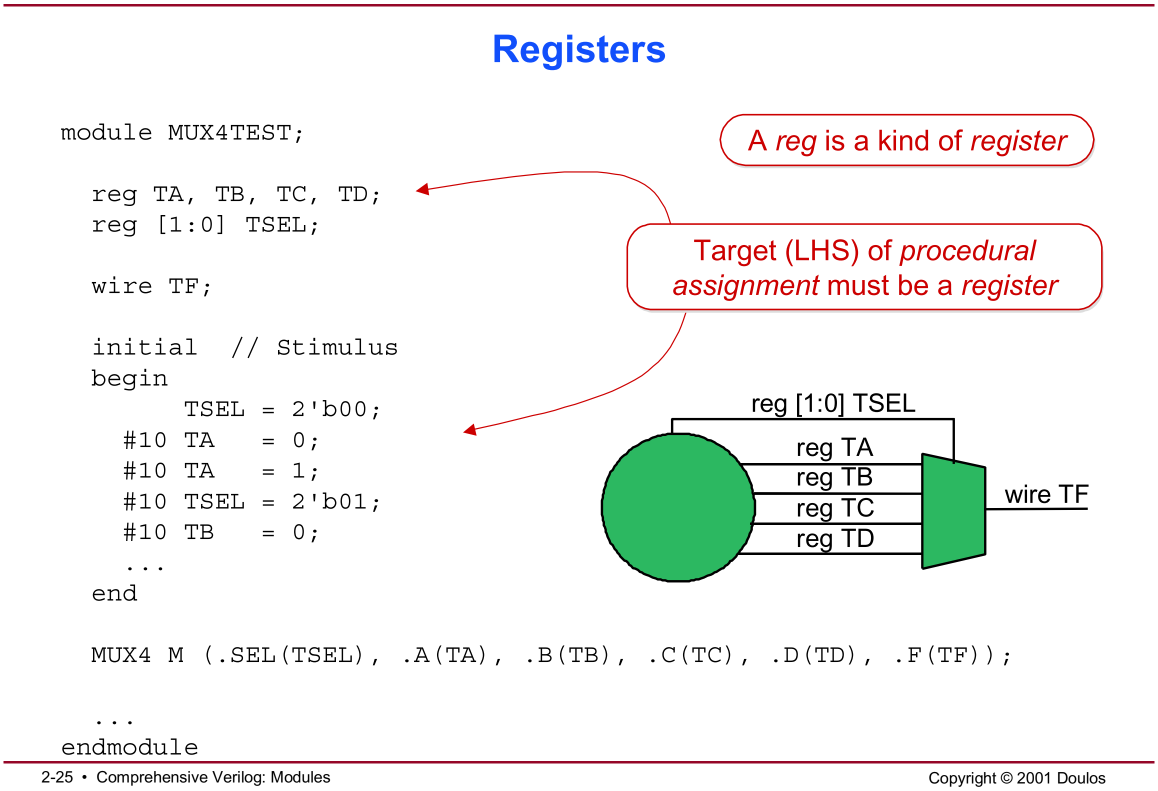
#10; A = 0;

(except that the latter is two statements)

Statements such as SEL = 2'b00 and A = 0 are called procedural assignments.

Note that Verilog allows you to use integer values such as 0 and 1 instead of numbers in the format 1'b0 and 1'b1. This topic will be discussed in detail later in the course.

Registers



Registers

```
module MUX4TEST;

  reg TA, TB, TC, TD;
  reg [1:0] TSEL;

  wire TF;

  initial  // Stimulus
  begin
        TSEL = 2'b00;
    #10 TA   = 0;
    #10 TA   = 1;
    #10 TSEL = 2'b01;
    #10 TB   = 0;
    ...
  end

  MUX4 M (.SEL(TSEL), .A(TA), .B(TB), .C(TC), .D(TD), .F(TF));

  ...
endmodule
```

A *reg* is a kind of *register*

Target (LHS) of *procedural assignment* must be a *register*

2-25 • Comprehensive Verilog: Modules

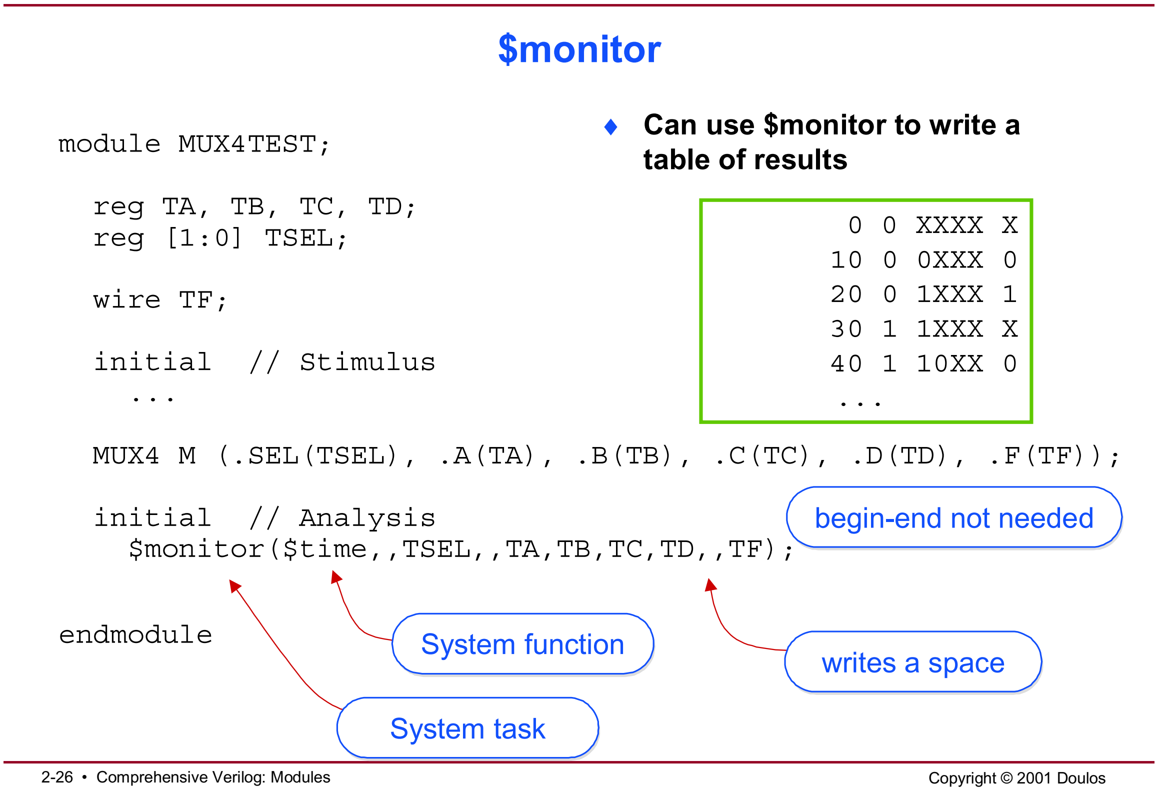
Notes:

So far we've seen values stored or carried by a Verilog wire. (Wires are the most common type of the more general category of nets.) Verilog has a second data type for storing a value called a register. Note that the term register does not imply necessarily that a hardware register (e.g. a flip-flop) is being described.

Registers are used in behavioral modeling with procedural blocks (initial and always). The rule for choosing between wires and registers is very simple, yet causes a lot of confusion! The rule is: registers can only be assigned within procedural blocks, and procedural blocks can only assign registers. Nets are assigned a value from a continuous assignment, or from a port.

Registers can be defined with the keyword reg. We will see other kinds of register later on.

$monitor



$monitor

```
module MUX4TEST;

  reg TA, TB, TC, TD;
  reg [1:0] TSEL;

  wire TF;

  initial  // Stimulus
    ...

  MUX4 M (.SEL(TSEL), .A(TA), .B(TB), .C(TC), .D(TD), .F(TF));

  initial  // Analysis
    $monitor($time,,TSEL,,TA,TB,TC,TD,,TF);

endmodule
```

- **Can use $monitor to write a table of results**

```
 0  0  XXXX  X
10  0  0XXX  0
20  0  1XXX  1
30  1  1XXX  X
40  1  10XX  0
 ...
```

begin-end not needed

System function

writes a space

System task

2-26 • Comprehensive Verilog: Modules

Copyright © 2001 Doulos

Notes:

Now for the results analysis, which is done by a second initial block.

System Task

This initial block contains a call to the system task $monitor. System tasks are built into the Verilog language (and simulator) and perform various utility functions. Their names always start with the $ character.

$monitor

$monitor writes out a line of text to the simulator log (i.e. window and/or log file) every time there is a change in the value of one of its reg or wire arguments (i.e. an event). Although $monitor is only called once (at time 0), the call creates a monitor which remains active until it is switched off or until simulation ends.

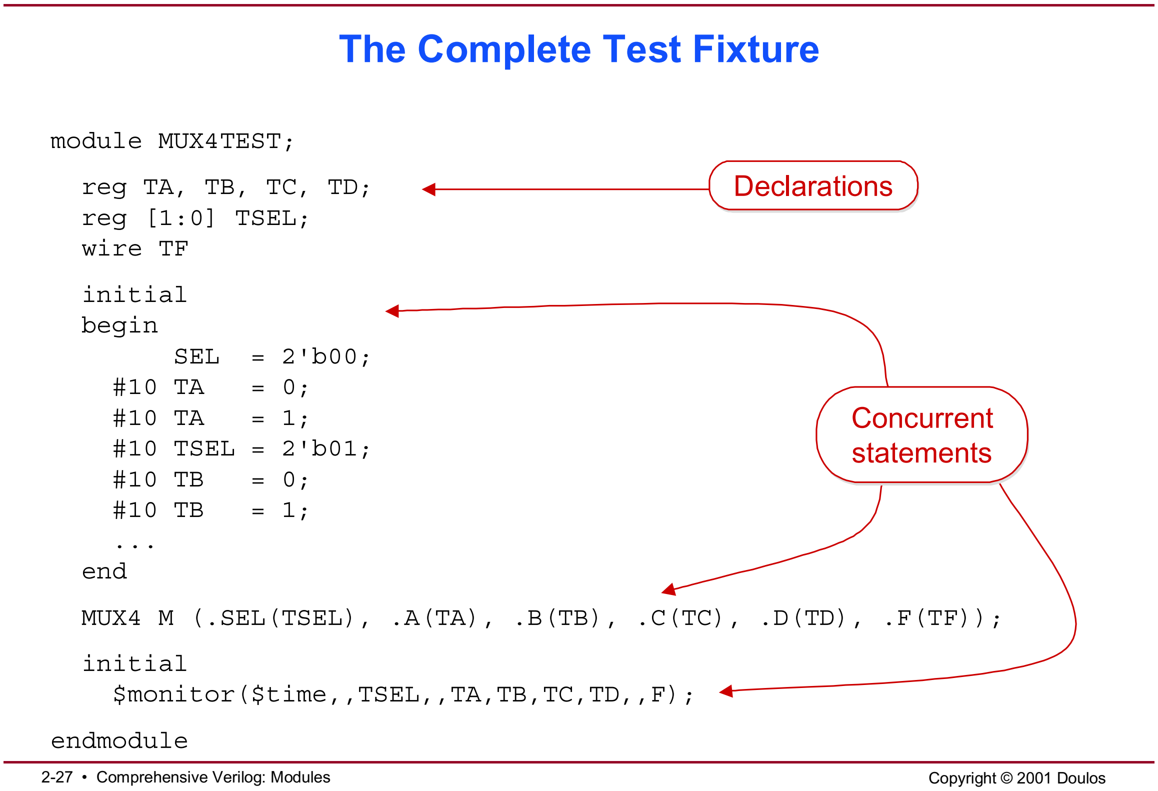
System Function, $time

In this example, the first argument in the call to $monitor is $time. $time is a system function. Like system tasks, system functions are built into the Verilog language. Unlike system tasks, system functions return a value, in this case the simulation time at which an event has triggered $monitor to write a line of text. (Note that a change of simulation time does not itself trigger $monitor.)

The remaining arguments to $monitor are expressions. The values of all the expressions are written out whenever one of them changes. A gap in the list (i.e. 2 adjacent commas) causes one space character to be written out.

Note that the initial block containing the call to $monitor does not contain the begin and end lines. They are not necessary here, because there is only one statement (the call to $monitor) in the initial block. However, they could have been included. The first initial block does require begin and end, because it contains more than one statement.

The Complete Test Fixture



The Complete Test Fixture

```
module MUX4TEST;

  reg TA, TB, TC, TD;
  reg [1:0] TSEL;
  wire TF

  initial
  begin
        SEL  = 2'b00;
    #10 TA   = 0;
    #10 TA   = 1;
    #10 TSEL = 2'b01;
    #10 TB   = 0;
    #10 TB   = 1;
    ...
  end

  MUX4 M (.SEL(TSEL), .A(TA), .B(TB), .C(TC), .D(TD), .F(TF));

  initial
    $monitor($time,,TSEL,,TA,TB,TC,TD,,F);

endmodule
```

Declarations
Concurrent statements

2-27 • Comprehensive Verilog: Modules

Notes:

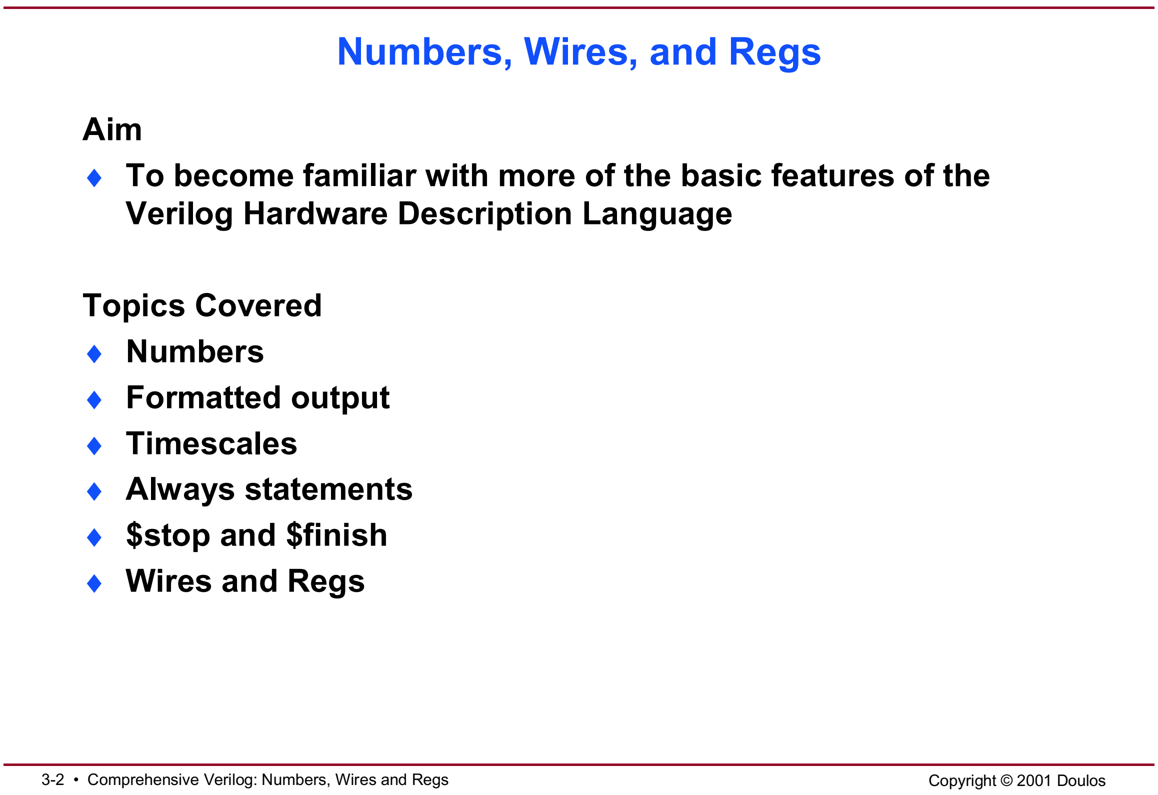
Here is the complete test fixture. It is a module with no ports, containing declarations and statements. The three main statements inside the module are concurrent; it does not matter what order they are written in. On the other hand, the data types (wire, reg etc.) must be defined before they are used.

An initial block is considered to be a single statement, even though it may contain a sequence of procedural statements. The statements inside the first initial block are bracketed by begin ... end. The statements between begin and end are executed in sequence from top to bottom.

Because the second initial block contains only one statement, the begin ... end statement brackets are not required. (They could be included if preferred.)

Module 3
Numbers, Wires, and Regs

Numbers, Wires, and Regs



Numbers, Wires, and Regs

Aim

- To become familiar with more of the basic features of the Verilog Hardware Description Language

Topics Covered

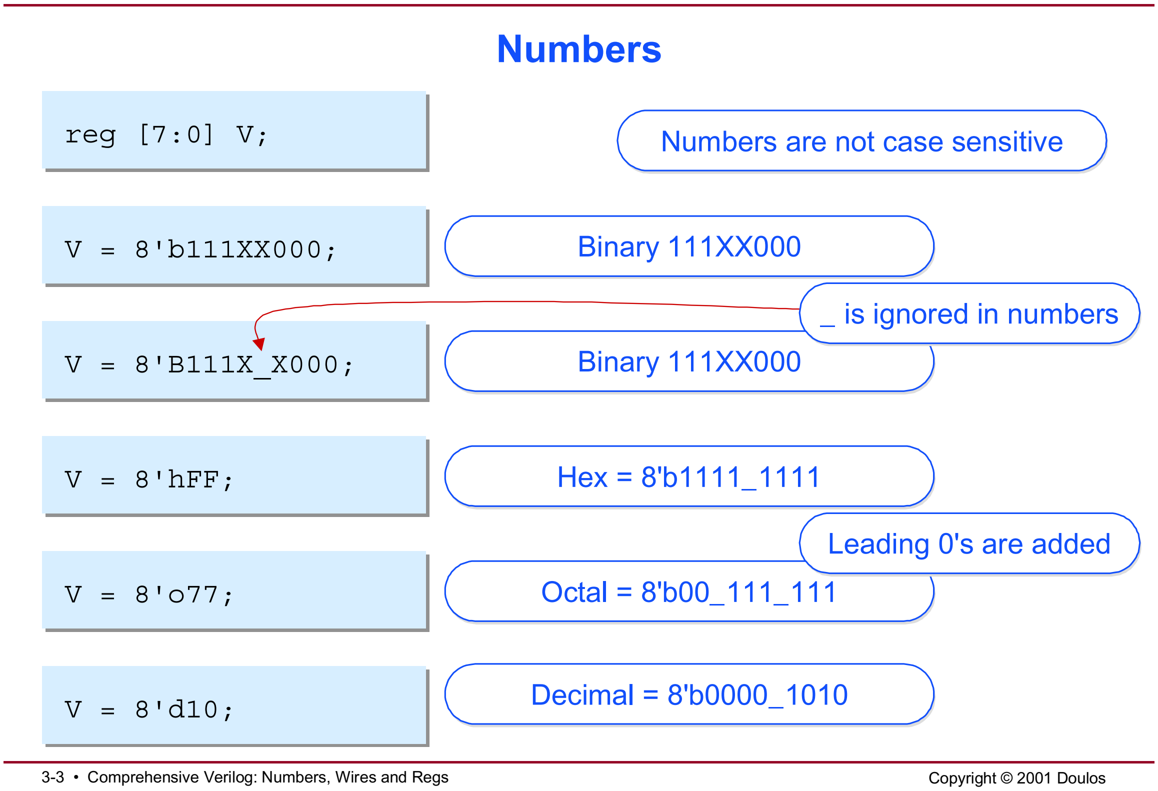
- Numbers
- Formatted output
- Timescales
- Always statements
- $stop and $finish
- Wires and Regs

3-2 • Comprehensive Verilog: Numbers, Wires and Regs

Copyright © 2001 Doulos

Notes:

Numbers



Numbers

```
reg [7:0] V;
```

Numbers are not case sensitive

```
V = 8'b111XX000;
```

Binary 111XX000

_ is ignored in numbers

```
V = 8'B111X_X000;
```

Binary 111XX000

```
V = 8'hFF;
```

Hex = 8'b1111_1111

Leading 0's are added

```
V = 8'o77;
```

Octal = 8'b00_111_111

```
V = 8'd10;
```

Decimal = 8'b0000_1010

3-3 • Comprehensive Verilog: Numbers, Wires and Regs

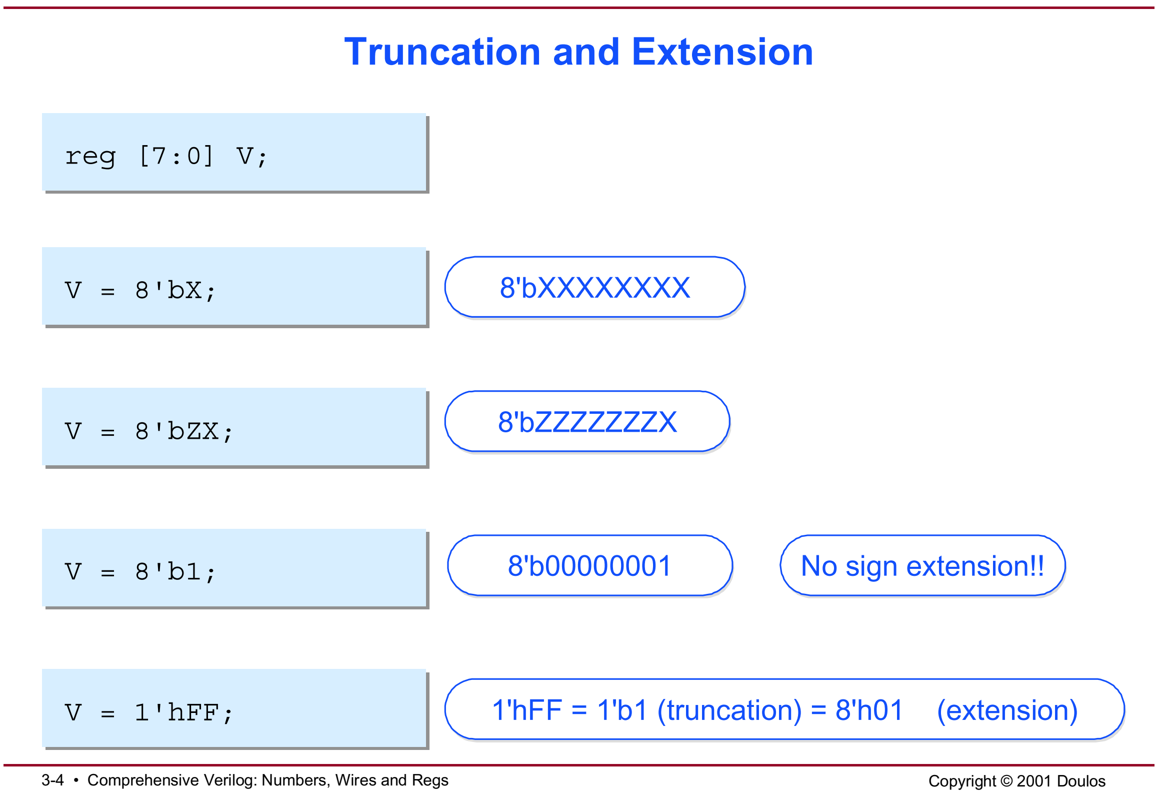
Notes:

Vector values can be written as integer literals, consisting of strings of 0s, 1s, Xs and Zs, preceded by the width (number of bits) and 'b (e.g. 8'b0101XXXX). To specify numbers in other bases use 'o (octal), 'd (decimal) or 'h (hexadecimal). Numbers may also contain underscores, which are ignored; they are used in long numbers to aid readability.

Note that numbers (including the base and the digits) are not case sensitive.

If all the digits of a number are not specified, the most significant bits are filled with zeroes. For example, 8'o77 represents 8'b00_111_111.

Truncation and Extension



Truncation and Extension

```
reg [7:0] V;
```

```
V = 8'bX;
```

8'bXXXXXXXX

```
V = 8'bZX;
```

8'bZZZZZZZX

```
V = 8'b1;
```

8'b00000001

No sign extension!!

```
V = 1'hFF;
```

1'hFF = 1'b1 (truncation) = 8'h01 (extension)

3-4 • Comprehensive Verilog: Numbers, Wires and Regs

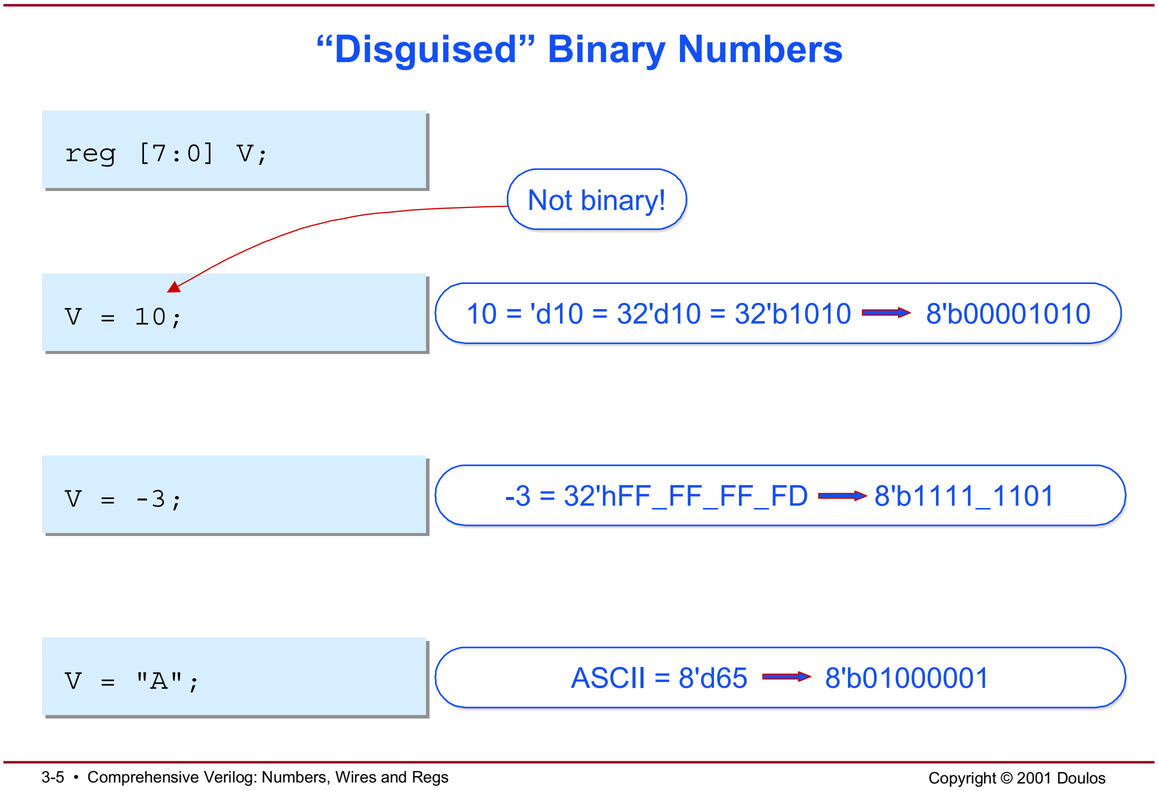
Notes:

We have seen that if all the digits of a number are not specified, the most significant bits are filled with zeroes. However, if the leftmost digit is X or Z, the missing bits are filled with Xs or Zs respectively. So 8'bx represents 8'bxxxx_xxxx, and 8'bzx represents 8'bzzzz_zzzx.

The values of regs and wires are considered to be unsigned quantities, even though negative numbers may be used in expressions. Care is required because sign extension does not take place.

In general, Verilog silently truncates or zero-fills vectors whenever a vector of the wrong size is given. Hence, in the example opposite, 1'hFF is truncated to 1'b1 (because the number is one bit wide), and the value 1'b1 is then extended to 8 bits by filling with zeros on the left (because we are assigning a one bit value (1'h...) to an 8 bit vector (V)). You must always beware vector truncation in Verilog!

“Disguised” Binary Numbers



“Disguised” Binary Numbers

```
reg [7:0] V;
```

Not binary!

```
V = 10;
```

10 = 'd10 = 32'd10 = 32'b1010 → 8'b00001010

```
V = -3;
```

-3 = 32'hFF_FF_FF_FD → 8'b1111_1101

```
V = "A";
```

ASCII = 8'd65 → 8'b01000001

3-5 • Comprehensive Verilog: Numbers, Wires and Regs

Copyright © 2001 Doulos

Notes:

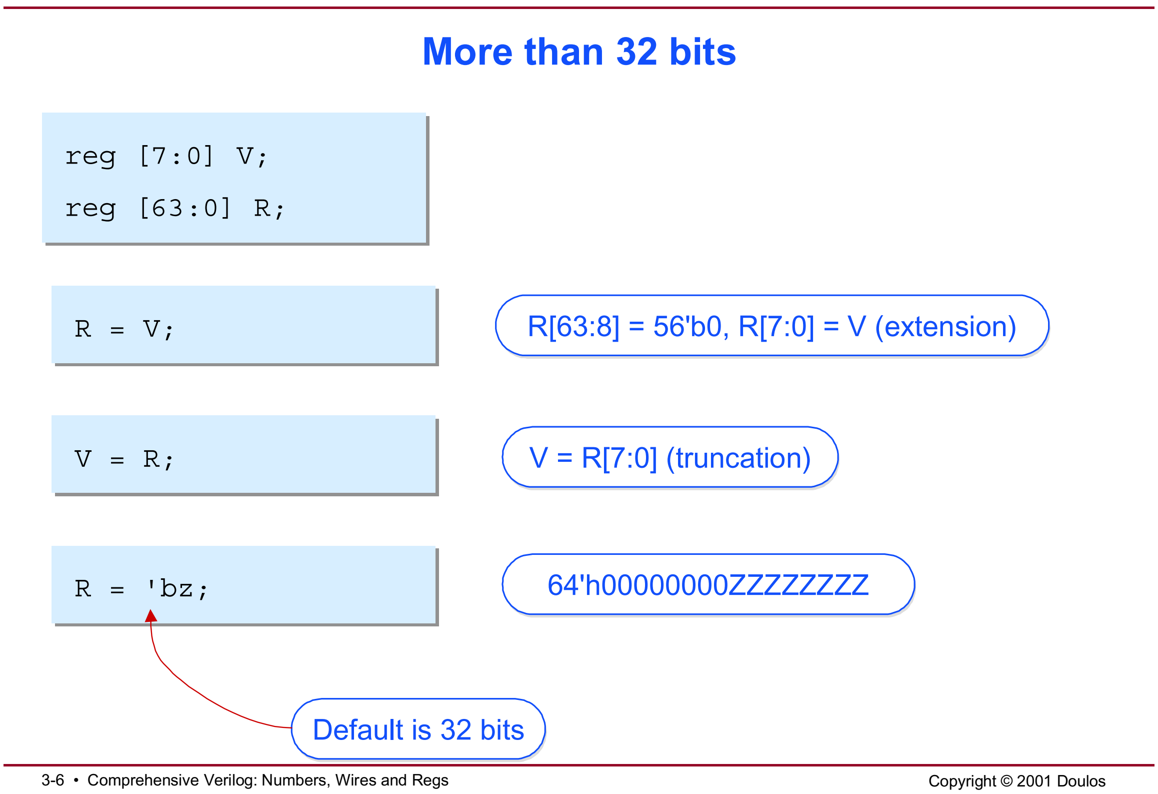
The preceding examples have all looked like binary numbers (i.e. collections of 0s, 1s, Xs and Zs). The examples here will still be represented as binary numbers, even though it may not be obvious.

10 is a 32 bit decimal number and not a two bit binary one!

-3 is also a 32 bit decimal number and will be represented in two's complement format. However, Verilog will interpret the value of V as an unsigned 8 bit number (i.e. 253)

Strings are converted to vectors by using the 8 bit ASCII code for each character.

More than 32 bits



More than 32 bits

```
reg [7:0] V;
reg [63:0] R;
```

```
R = V;
```

R[63:8] = 56'b0, R[7:0] = V (extension)

```
V = R;
```

V = R[7:0] (truncation)

```
R = 'bz;
```

64'h00000000ZZZZZZZZ

Default is 32 bits

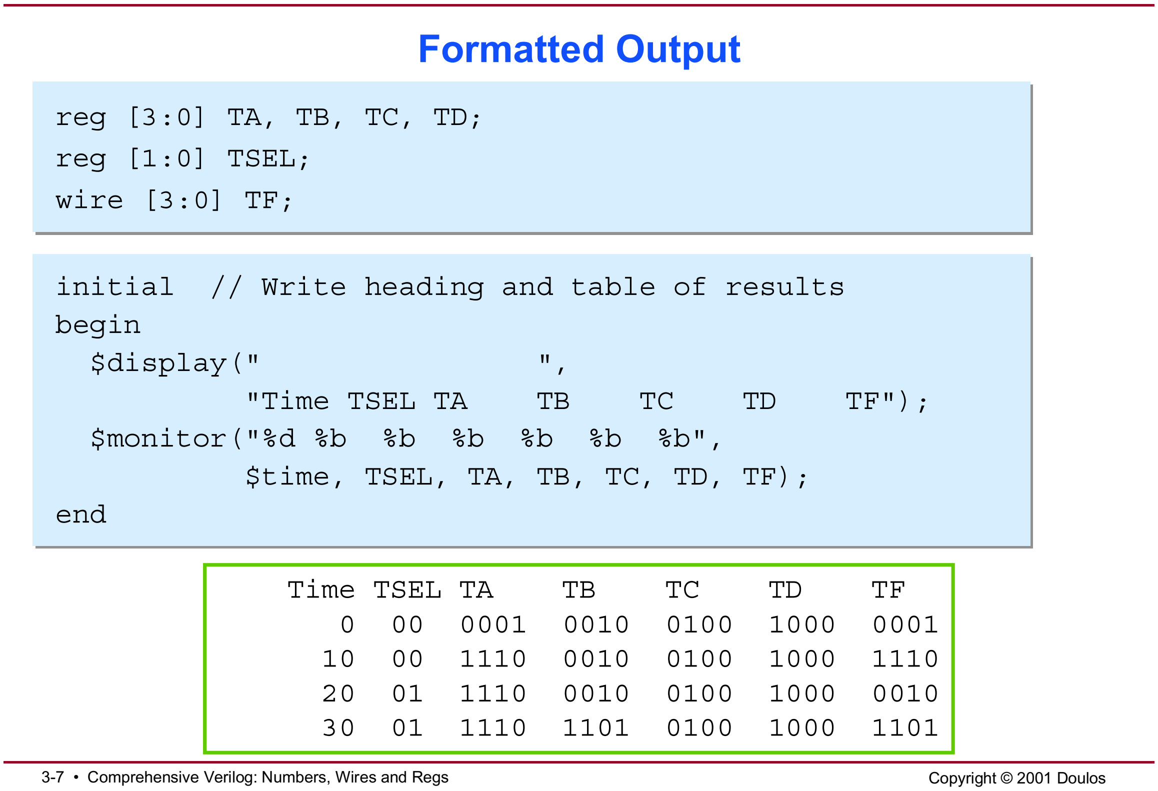
3-6 • Comprehensive Verilog: Numbers, Wires and Regs

Notes:

The default width of a number if none is specified is 32 bits. (Strictly, the default is specified by the simulator or synthesis tool implementation, but in practice it will almost certainly be 32 bits.) This may cause unexpected results if wires or regs greater than 32 bits are being used.

In the bottom example, R, which is 64 bits wide, is being assigned a value that is only 32 bits wide. This number is extended to 64 bits with zeroes. Note that Xs or Zs are only used to extend a number to its specified or default size, and not when assigning a value to a reg or wire. To put it another way, the Verilog compiler doesn't "look" at the right hand side of the assignment until after any X or Z extension has been completed.

Formatted Output



Formatted Output

```
reg [3:0] TA, TB, TC, TD;
reg [1:0] TSEL;
wire [3:0] TF;
```

```
initial  // Write heading and table of results
begin
  $display("                ",
           "Time TSEL TA    TB    TC    TD    TF");
  $monitor("%d %b  %b  %b  %b  %b  %b",
           $time, TSEL, TA, TB, TC, TD, TF);
end
```

```
Time TSEL TA    TB    TC    TD    TF
   0  00  0001  0010  0100  1000  0001
  10  00  1110  0010  0100  1000  1110
  20  01  1110  0010  0100  1000  0010
  30  01  1110  1101  0100  1000  1101
```

3-7 • Comprehensive Verilog: Numbers, Wires and Regs

Notes:

$display

In addition to $monitor, there is a second system task $display for writing output to the simulator log. $display writes out data, once only, immediately it is called; it does not create a continuous monitor.

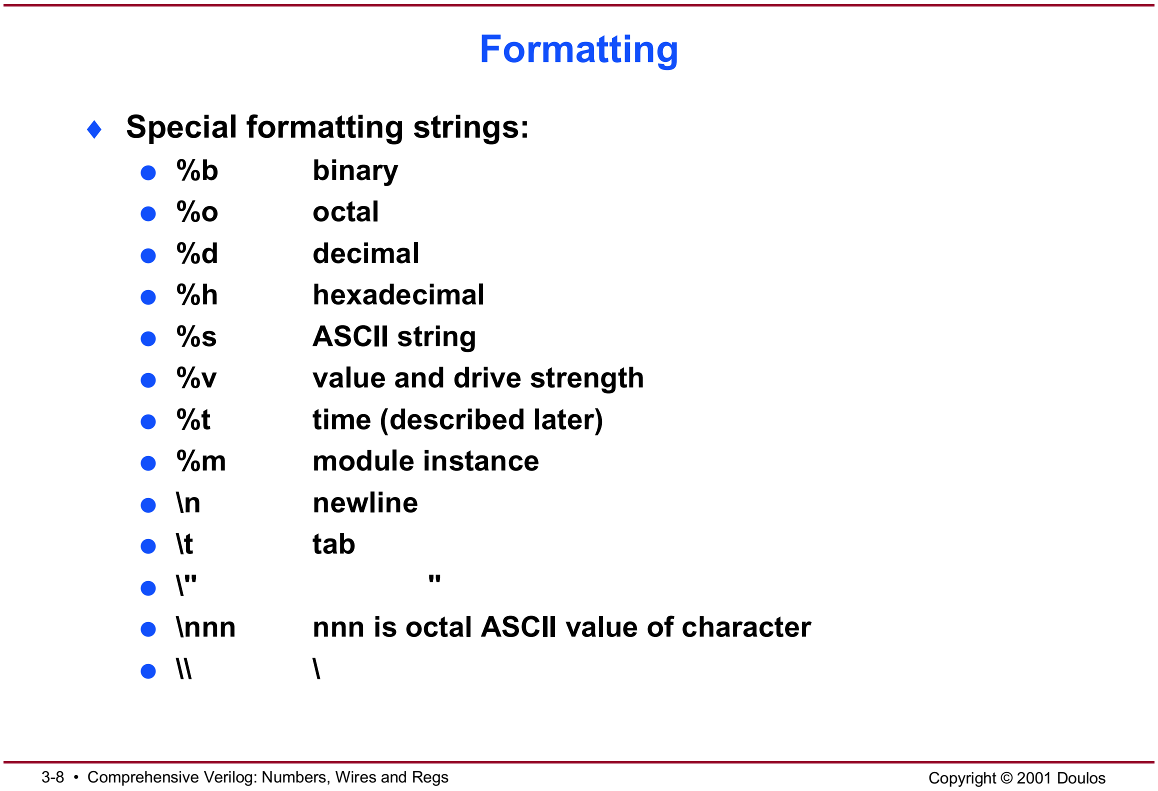
$time

$time is a system function that returns the current simulation time.

The output from $display and $monitor can be formatted by including special characters, called format specifiers, in the strings passed as arguments to the tasks.

Each format specifier %b, %d etc. is substituted with the value of the next argument to be written in the list, formatted accordingly.

Formatting



Formatting

- Special formatting strings:

%b	binary
%o	octal
%d	decimal
%h	hexadecimal
%s	ASCII string
%v	value and drive strength
%t	time (described later)
%m	module instance
\n	newline
\t	tab
\"	"
\nnn	nnn is octal ASCII value of character
\\	\

3-8 • Comprehensive Verilog: Numbers, Wires and Regs

Copyright © 2001 Doulos

Notes:

Here are the format specifiers that are understood by $display and $monitor.

Note that the backslash character is used as an escape character. If you need to write out a literal backslash in a string, you must include two backslashes. Similarly, if you want to write out a literal double quote, you must escape it with a backslash. (Otherwise " is interpreted as marking the end of the format string.)

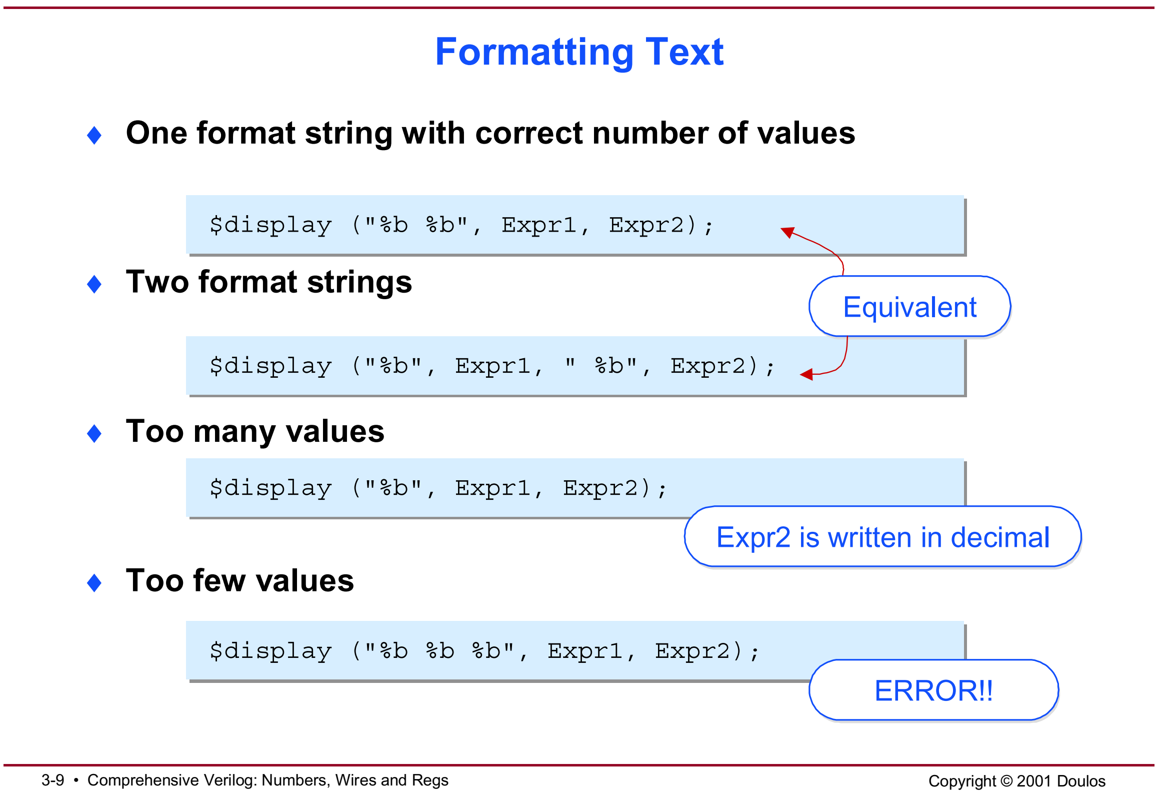
For example,

```
   $display("The file is \"C:\\Files\\File.txt\"");
```

results in the text

```
   The file is "C:\Files\File.txt"
```

Formatting Text



Formatting Text

- One format string with correct number of values

```
$display ("%b %b", Expr1, Expr2);
```

- Two format strings

```
$display ("%b", Expr1, " %b", Expr2);
```

Equivalent

- Too many values

```
$display ("%b", Expr1, Expr2);
```

Expr2 is written in decimal

- Too few values

```
$display ("%b %b %b", Expr1, Expr2);
```

ERROR!!

3-9 • Comprehensive Verilog: Numbers, Wires and Regs

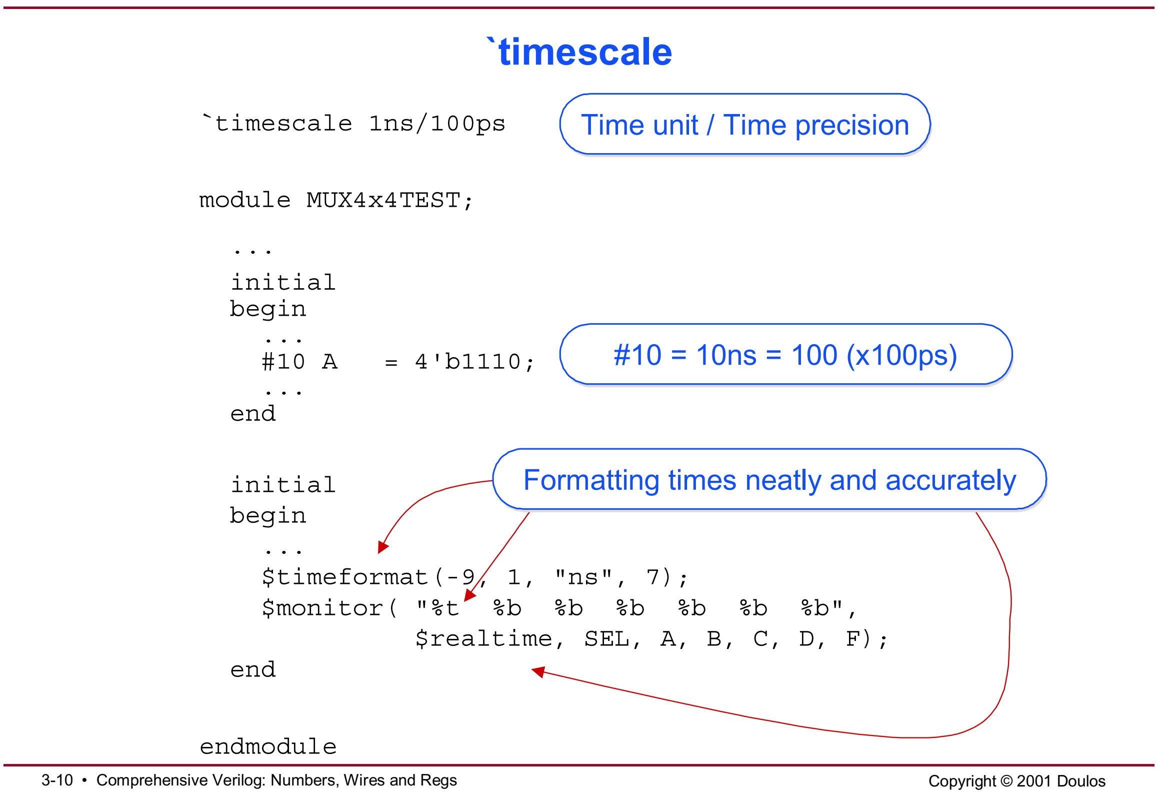
Notes:

Formatted arguments can be organized in one of two ways. The first way is to have a long string containing several special formatting strings, followed by a list of arguments to be formatted. The second way is to arrange the arguments in pairs, each pair consisting of a formatting string and a value to be formatted. Both styles are shown.

Generally there must be exactly the same number of expressions following a format string as there are format specifiers in the string. If there are too many expressions, the default format is decimal. If there are too few, it is an error.

The exception to this is %m, which does not require an expression. %m displays the instance name of the module in which the $display or $monitor statement appears.

‘timescale



`timescale

```
`timescale 1ns/100ps

module MUX4x4TEST;

  ...
  initial
  begin
    ...
    #10 A    = 4'b1110;
    ...
  end

  initial
  begin
    ...
    $timeformat(-9, 1, "ns", 7);
    $monitor( "%t  %b  %b  %b  %b  %b  %b",
              $realtime, SEL, A, B, C, D, F);
  end

endmodule
```

Time unit / Time precision

#10 = 10ns = 100 (x100ps)

Formatting times neatly and accurately

3-10 • Comprehensive Verilog: Numbers, Wires and Regs

Copyright © 2001 Doulos

Notes:

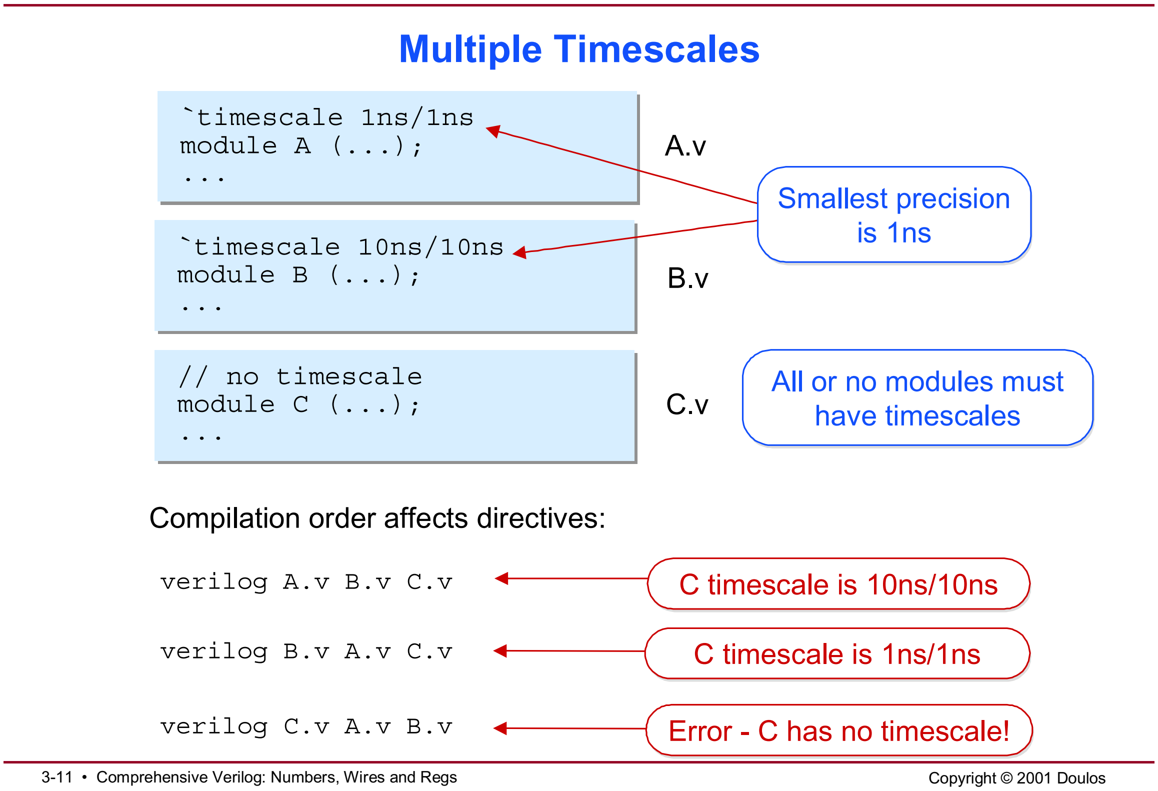
`timescale

So far, time within a Verilog simulation has been a dimensionless number. More usually, we want to simulate in terms of known physical time units, such as picoseconds or nanoseconds. The units of time in Verilog can be set using the `timescale compiler directive. (A compiler directive is a piece of Verilog syntax, beginning with a backwards apostrophe (`), that tells the compiler something about the rest of the Verilog code.)

The first quantity in the `timescale directive is the time unit. All delays in the Verilog source code are multiples of this time unit. In the example, #10 means a procedural delay of 10 ns.

The second quantity in the `timescale directive is the precision. In this example, the precision is 100ps, which means that delays are rounded to the nearest 100ps. So #10 is 10ns or 100 100ps units.

Multiple Timescales



Multiple Timescales

A.v

```
`timescale 1ns/1ns
module A (...);
...
```

B.v

```
`timescale 10ns/10ns
module B (...);
...
```

C.v

```
// no timescale
module C (...);
...
```

Smallest precision is 1ns

All or no modules must have timescales

Compilation order affects directives:

```
verilog A.v B.v C.v
```

C timescale is 10ns/10ns

```
verilog B.v A.v C.v
```

C timescale is 1ns/1ns

```
verilog C.v A.v B.v
```

Error - C has no timescale!

3-11 • Comprehensive Verilog: Numbers, Wires and Regs

Copyright © 2001 Doulos

Notes:

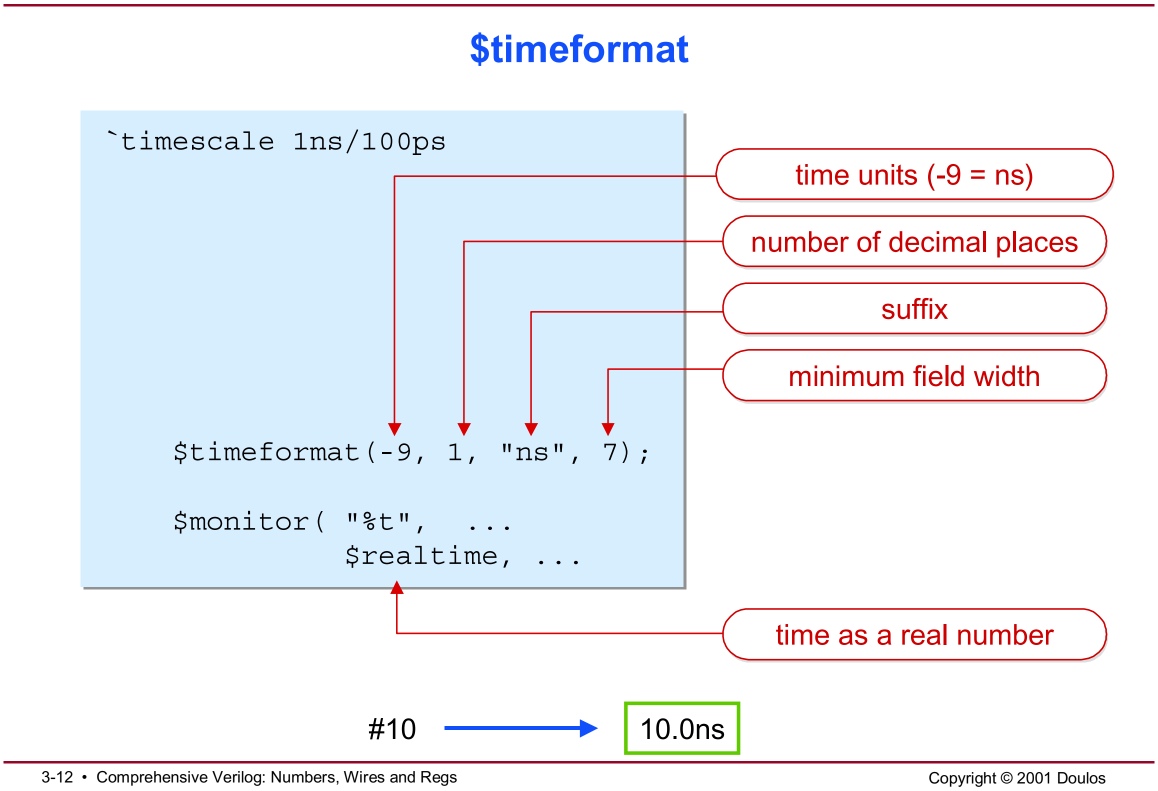
If one module in a design has a timescale, then a timescale must be supplied for all the modules in the design, even if some of the modules do not use delays. The recommended practice is always to specify a timescale before every module.

It is quite possible that each of the modules in a design has a different timescale. In this case the first value - the time units - is used in the way that has been described. For the second value - the precision - the simulator calculates the smallest precision from all the timescales in the design, and uses that value, ignoring the individual precision values. (But note that some simulators allow the precision to be changed when a design is loaded for simulation; others ignore the precision altogether).

Some Verilog tools allow several source files to be compiled at the same time. This is the same as first concatenating the files and compiling the single concatenated file. A compiler directive like `timescale affects all modules that follow the directive in the source file, and in all subsequent files compiled at the same time, until another directive appears. This means that if some modules have timescales and others do not, the latter may derive their timescales from previously compiled modules.

We have seen that if any module in a design has a timescale, then every module in the design must also have a timescale. A module can be given a default timescale by compiling it last. However, it is recommended that every module always be given an explicit timescale, even if there are no delays in that module.

$timeformat



$timeformat

```
`timescale 1ns/100ps




    $timeformat(-9, 1, "ns", 7);

    $monitor( "%t",  ...
              $realtime, ...
```

3-12 • Comprehensive Verilog: Numbers, Wires and Regs

Notes:

`timescale is used to specify the size of the units of delays ("#10 means 10ns"). When writing times using $display and $monitor, the $timeformat system task may be used to specify the way they are written. $timeformat is used with the format specifier %t.

$timeformat

$timeformat may only be called once in a design, so it is best included in the test fixture. The four parameters indicate respectively (i) the units in which times are written, with -12 meaning picoseconds, -9 nanoseconds, -6 microseconds etc.; (ii) the number of decimal places to display; (iii) a text string to follow the time; (iv) the minimum number of characters to display. If necessary, the field is padded on the left with spaces.

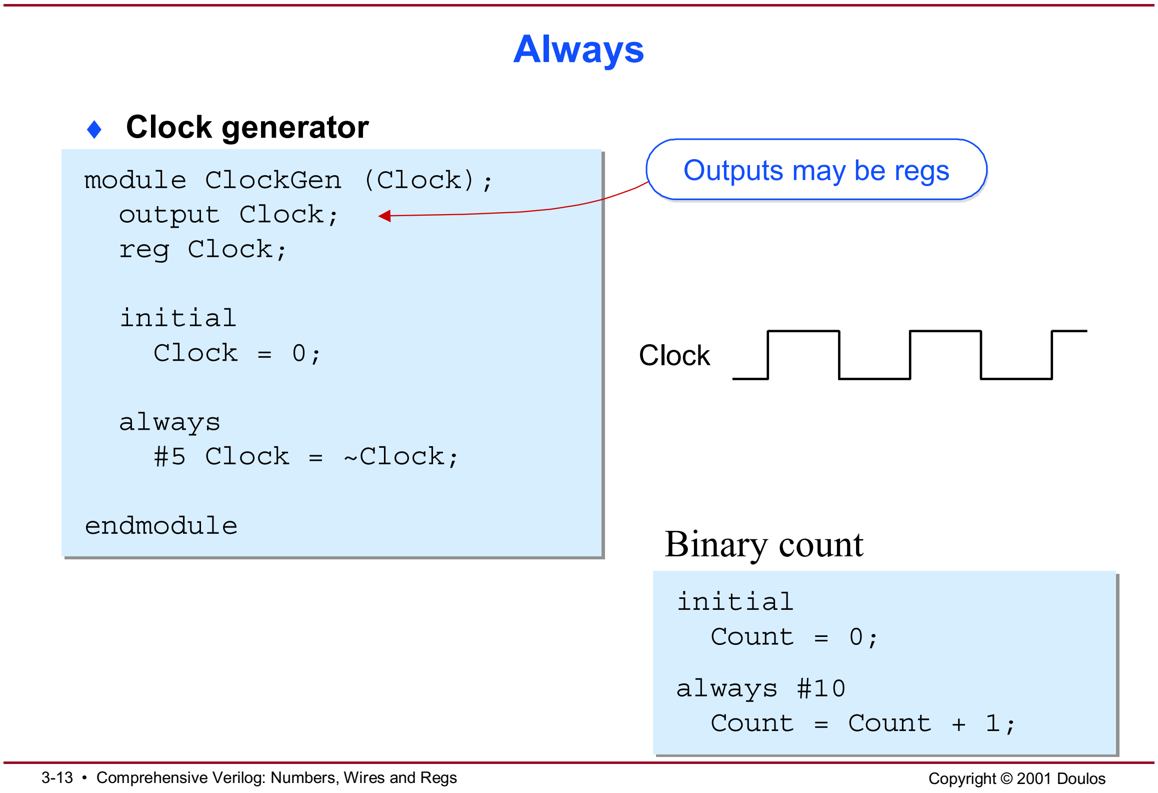
Note that $timeformat enables times to be written in a unit that is different to the `timescale unit.

$realtime

$realtime is a system function that returns the current simulation time as a real number in the `timescale units of the module from which it was called. $realtime should be used in preference to $time, because the latter will round delays to the nearest (integer) time unit.

In the example above, a delay of #10 is written as shown. This is because the `timescale directive indicates that #10 is 10ns, and the $timeformat task causes the value to be written out in ns, with one decimal place, using the suffix "ns" and with a leading space to give a total of 7 characters, including the decimal point and the suffix.

Always



Always

- Clock generator

```
module ClockGen (Clock);
  output Clock;
  reg Clock;

  initial
    Clock = 0;

  always
    #5 Clock = ~Clock;

endmodule
```

Outputs may be regs

Binary count

```
initial
  Count = 0;

always #10
  Count = Count + 1;
```

3-13 • Comprehensive Verilog: Numbers, Wires and Regs

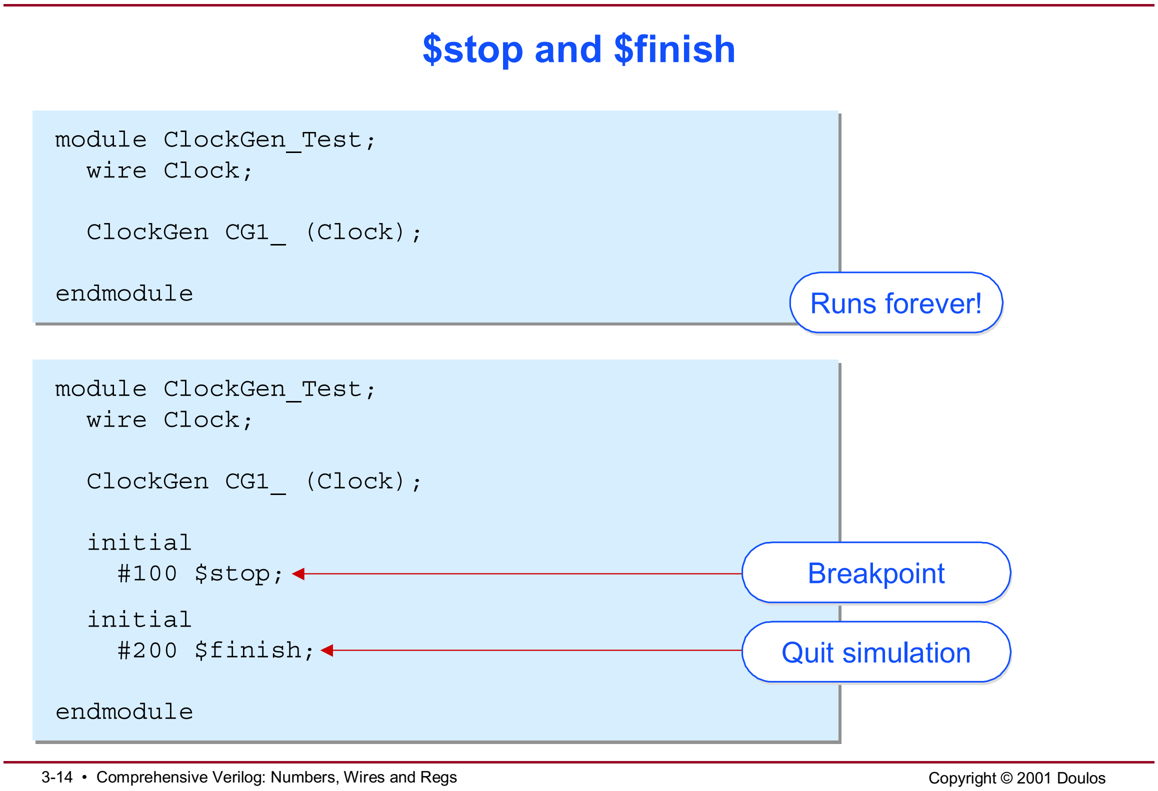
Notes:

We have already met one form of procedural block, the initial statement. The other kind of procedural block is the always block. An always block is very similar to an initial block, but instead of executing just once, it executes repeatedly throughout simulation. Having reached the end, it immediately starts again at the beginning, and so defines an infinite loop.

In this example, an always block is being used to generate a clock waveform. Note that anything assigned within the always block must be defined as a register. Note also that the register Clock is assigned in two different procedural blocks; the initial and the always. The initial block causes Clock to be set to 0 at the beginning of simulation. Thereafter, the always block makes new assignments to Clock. This explicit initialization is necessary here, because regs are initialized to 'bx by default.

The second example shows how to use an always block to generate a binary counting sequence. Again, an initial block is used to provide a starting value for Count. When Count reaches its maximum value (e.g. 4'b1111 for a four-bit reg), adding 1 causes its value to "roll over" to 0 and the count sequence recommences.

$stop and $finish



$stop and $finish

```
module ClockGen_Test;
  wire Clock;

  ClockGen CG1_ (Clock);

endmodule
```

Runs forever!

```
module ClockGen_Test;
  wire Clock;

  ClockGen CG1_ (Clock);

  initial
    #100 $stop;

  initial
    #200 $finish;

endmodule
```

Breakpoint

Quit simulation

3-14 • Comprehensive Verilog: Numbers, Wires and Regs

Notes:

These examples instance the clock generator module from the previous slide.

If a design contains always statements like those on the previous slide, then potentially it will simulate indefinitely. To prevent this, you can tell the simulator to execute for a limited time; you can interrupt the simulation; or you can include breakpoints in the Verilog code.

$stop

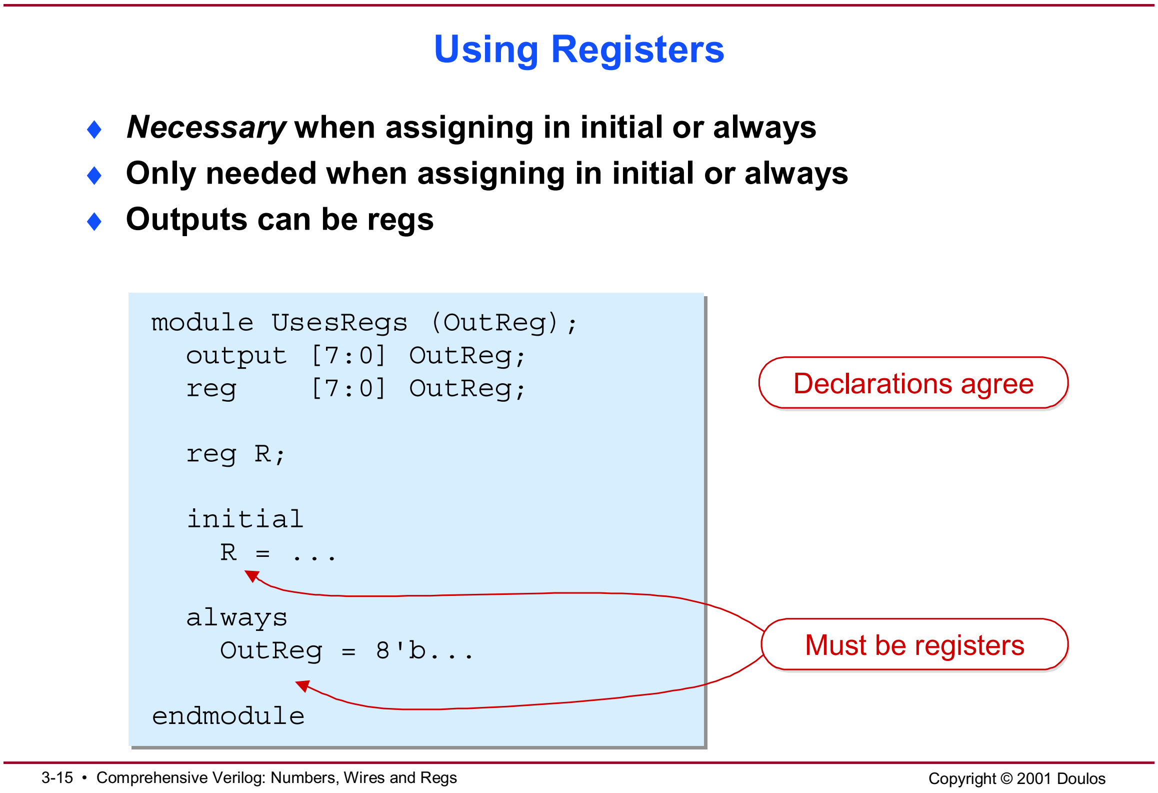
The system task $stop is used to set breakpoints. When $stop is called, simulation is suspended. The intention is that you can then use an interactive debugger to examine the state of the simulation before continuing.

$finish

The system task $finish is used to quit or abandon simulation. It can be used in contexts such as that shown to finish a simulation that includes a clock generator or other infinite loop. Alternatively, it may be possible to exit from the simulator using its commands or graphical interface.

Note that in this example, Clock must be a wire, not a register. This is because in the current module, Clock is driven by a port, not assigned within a procedural block. Clock is a wire, despite the fact that it is connected to a port which is itself a register within the module ClockGen.

Using Registers



Using Registers

- ***Necessary*** **when assigning in initial or always**
- **Only needed when assigning in initial or always**
- **Outputs can be regs**

```
module UsesRegs (OutReg);
  output [7:0] OutReg;
  reg    [7:0] OutReg;

  reg R;

  initial
    R = ...

  always
    OutReg = 8'b...

endmodule
```

Declarations agree

Must be registers

3-15 • Comprehensive Verilog: Numbers, Wires and Regs

Copyright © 2001 Doulos

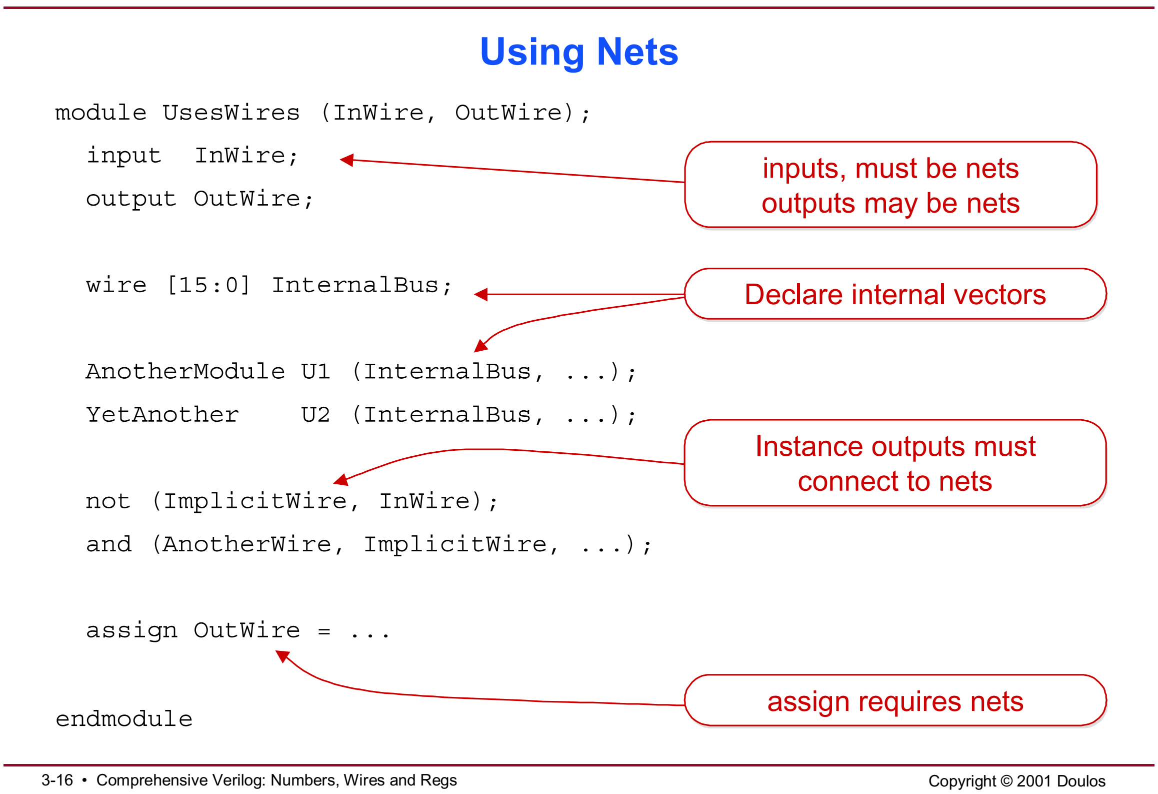
Notes:

To complete this section, we are going to look in more detail at the rules for using nets and registers.

As far as registers are concerned, the fundamental rule is in two parts: you must use registers when assigning values in initial and always blocks; in fact you only need to use registers in this context.

This rule must be followed even if it means that an output port has also to be declared as a reg. In this case the output and reg declarations must match, i.e. for vectors they must use the same LSB and MSB.

Using Nets



Using Nets

```
module UsesWires (InWire, OutWire);
  input  InWire;
  output OutWire;

  wire [15:0] InternalBus;

  AnotherModule U1 (InternalBus, ...);
  YetAnother    U2 (InternalBus, ...);

  not (ImplicitWire, InWire);
  and (AnotherWire, ImplicitWire, ...);

  assign OutWire = ...

endmodule
```

inputs, must be nets
outputs may be nets

Declare internal vectors

Instance outputs must connect to nets

assign requires nets

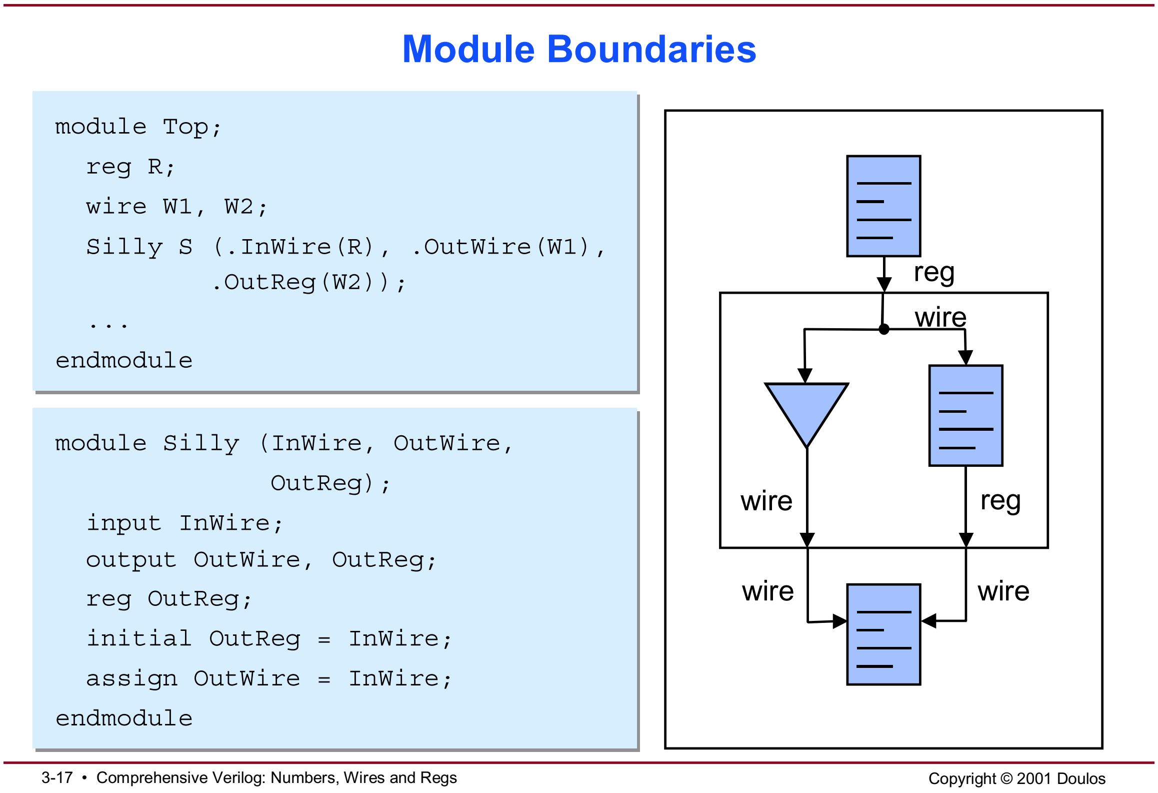
3-16 • Comprehensive Verilog: Numbers, Wires and Regs

Copyright © 2001 Doulos

Notes:

The rule for when to use nets is easy: You use nets whenever you don't use registers! As we have already seen, you must use nets for input ports, and when making continuous assignments. You must also use nets to connect to the outputs of module or primitive instances. You can use implicit declarations for scalar wires, but for vectors you have to declare appropriate wires explicitly.

Module Boundaries



Module Boundaries

```
module Top;
  reg R;
  wire W1, W2;
  Silly S (.InWire(R), .OutWire(W1),
          .OutReg(W2));
  ...
endmodule
```

```
module Silly (InWire, OutWire,
              OutReg);
  input InWire;
  output OutWire, OutReg;
  reg OutReg;
  initial OutReg = InWire;
  assign OutWire = InWire;
endmodule
```

3-17 • Comprehensive Verilog: Numbers, Wires and Regs

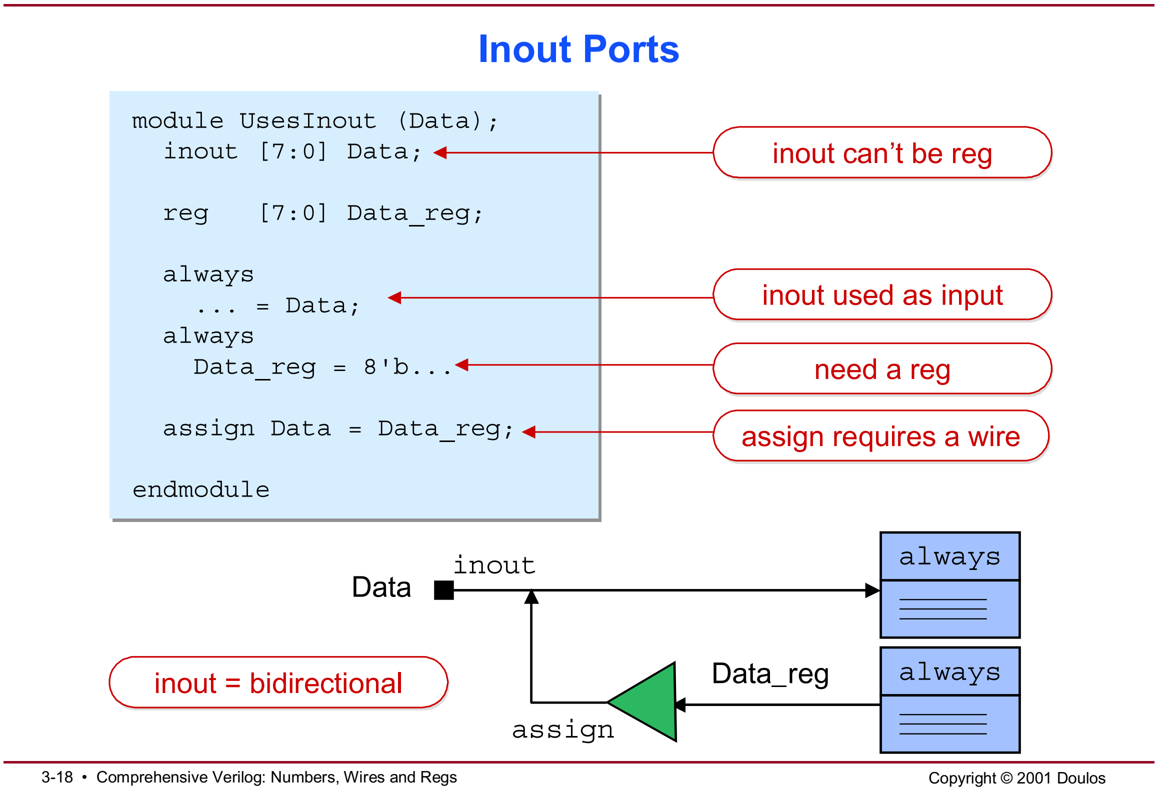
Notes:

The rules for nets and registers mean that a single connection may sometimes be modeled using both a net and a register! In the example above, the module Top has a reg R connected to the input port InWire of the instance S of Silly. But we know that module inputs must be wires.

Similarly, the output OutReg of Silly, which is a reg, must be connected to a wire when Silly is instanced.

Behind the scenes, there are implicit continuous assignments at the module boundaries.

Inout Ports



Inout Ports

```
module UsesInout (Data);
  inout  [7:0] Data;

  reg    [7:0] Data_reg;

  always
    ... = Data;
  always
    Data_reg = 8'b...

  assign Data = Data_reg;

endmodule
```

inout can’t be reg

inout used as input

need a reg

assign requires a wire

inout = bidirectional

3-18 • Comprehensive Verilog: Numbers, Wires and Regs

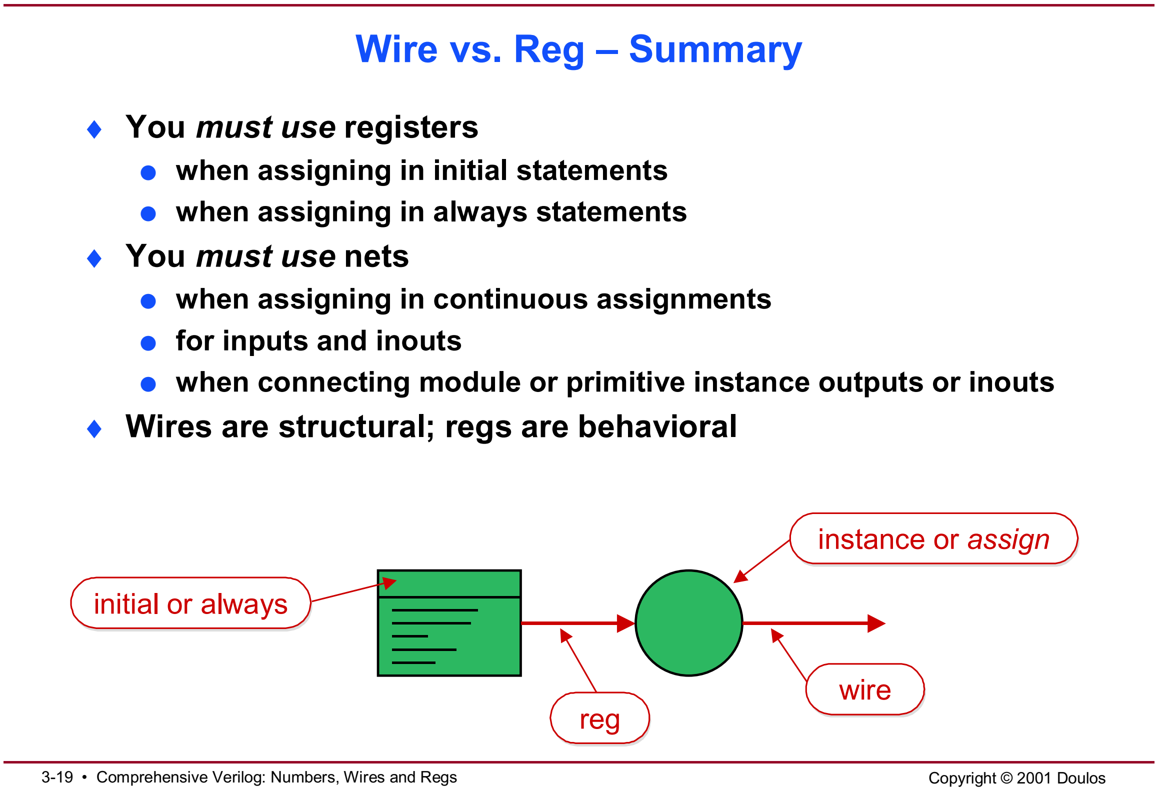
Copyright © 2001 Doulos

Notes:

inout

An inout port is bidirectional and, like an input, must be a net. This restriction is not a problem with inputs, because you should never need to declare an input as a reg. There is a potential problem with inout ports, because you may want to use an always block to assign a value to an inout - the rules mean that you can't! Instead you must use a reg in the always statement and a continuous assignment to drive the value of the reg onto the inout.

Wire vs. Reg – Summary



Wire vs. Reg – Summary

- You *must use* registers
  - when assigning in initial statements
  - when assigning in always statements
- You *must use* nets
  - when assigning in continuous assignments
  - for inputs and inouts
  - when connecting module or primitive instance outputs or inouts
- Wires are structural; regs are behavioral

instance or *assign*
initial or always
reg
wire

3-19 • Comprehensive Verilog: Numbers, Wires and Regs

Notes:

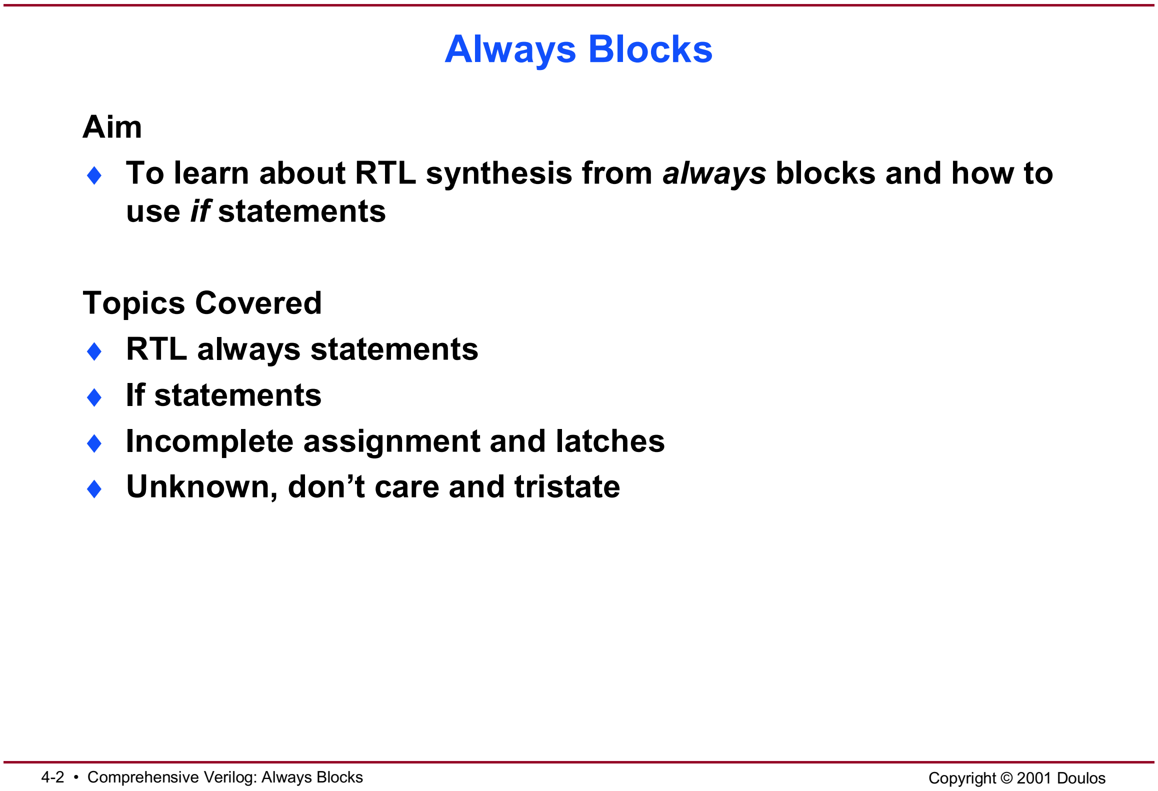
In summary,

- You must use regs as the target of assignments in initial or always blocks.
- You must use wires as the target of continuous assignments, for input and inout ports and for connecting to module or primitive instance outputs or inouts.

One way of understanding all this is that wires are used in structural models (netlists) and regs in behavioral models. A continuous assignment is considered as essentially a structural statement.

Module 4
Always Blocks

Always Blocks



Always Blocks

Aim

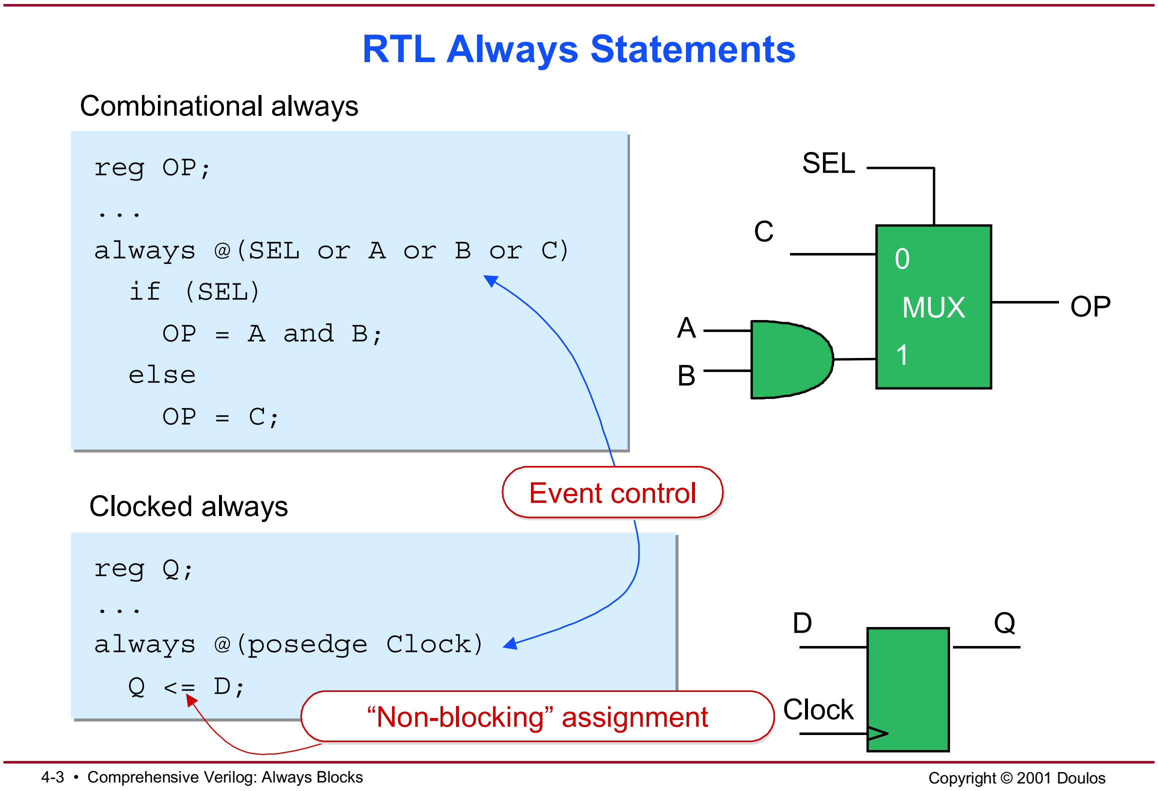
- **To learn about RTL synthesis from *always* blocks and how to use *if* statements**

Topics Covered

- **RTL always statements**
- **If statements**
- **Incomplete assignment and latches**
- **Unknown, don't care and tristate**

Notes:

RTL Always Statements



RTL Always Statements

Combinational always

```
reg OP;
...
always @(SEL or A or B or C)
  if (SEL)
    OP = A and B;
  else
    OP = C;
```

Event control

Clocked always

```
reg Q;
...
always @(posedge Clock)
  Q <= D;
```

“Non-blocking” assignment

4-3 • Comprehensive Verilog: Always Blocks

Notes:

Always

The examples above show two examples of always statements that may be synthesized. The first example shows an always statement being used to describe combinational logic; in the second example a D-type flip-flop - a sequential circuit - is being described.

The always statement is one of the most powerful and flexible statements in Verilog, for it can be used to describe the behavior of any part of a system at any level of abstraction from Boolean equations through register transfer level to complex algorithms.

Event control

The @ and list of names in brackets after the word always is called an event control. The always statement will execute whenever any of the wires or regs in the event control changes value. Note that the word 'or' in the event control is not a logical operator, but the separator that is used in an event control. In the first example, what it means is "whenever SEL changes value or A changes value or B changes value or C changes value, execute the always statement". The event control in the second example causes the always statement to be executed whenever there is a rising edge on Clock. Clocked always statements will be discussed more fully in a later section.

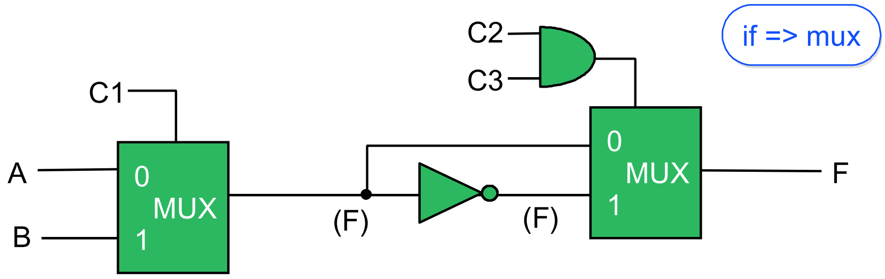
An event control as the first statement in an always block is sometimes called a sensitivity list.

Remember that only regs may be given values in an always statement. Hence both OP and Q must be declared as regs However, only Q is synthesized as a hardware register (a flip-flop). The Verilog term reg does not necessarily mean that a register will be inferred. Regs in clocked always statements are often assigned using the non-blocking assignment operator (<=), which will be described later.

If Statements

If Statements

```
always @(C1 or C2 or C3 or A or B)
begin

  if (~C1)
    F = A;
  else
    F = B;

  if (C2 & C3) F = ~F;

end
```



if => mux

Notes:

Here is an example of an always statement describing a combinational logic function. The always statement contains two if statements.

If

The first if statement tests the value of ~C1, and then executes one of the two alternative assignments depending on that value. If ~C1 has a logic one value (i.e. C1 is 1'b0), it is considered to be 'true', and the first assignment (F = A) is executed. If ~C1 is logic zero or unknown, it is considered 'false' and the second assignment is executed.

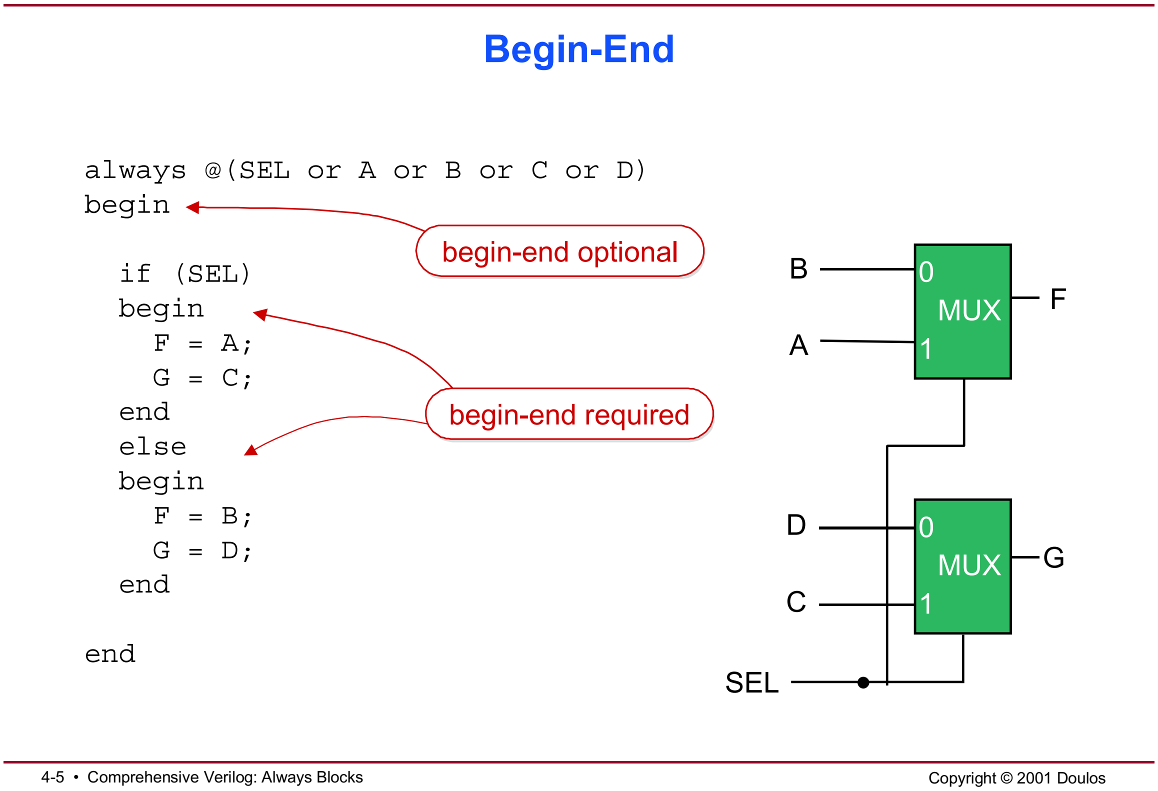
The else part of an if statement may be omitted. So the second if statement inverts the value of F if C2 and C3 are both high; otherwise (if either C2 or C3 is low) F is not inverted.

Synthesis

If statements are synthesized by generating a multiplexer for any reg assigned within the if statement. The select input on each mux is driven by logic determined by the if condition, and the data inputs are determined by the expressions on the right hand sides of the assignments. If a register is unassigned under some conditions, then the old value is passed through the mux, as in the second if statement shown opposite.

Note that there is not a one-to-one correspondence between the elements of the Verilog model, and the synthesized hardware. In particular, the reg F is in a sense synthesized as three separate wire segments. Furthermore, when the logic is optimized it may not be possible to see anything in the optimized design that corresponds to the original reg, F.

Begin-End



Begin-End

```
always @(SEL or A or B or C or D)
begin

  if (SEL)
  begin
    F = A;
    G = C;
  end
  else
  begin
    F = B;
    G = D;
  end

end
```

begin-end optional

begin-end required

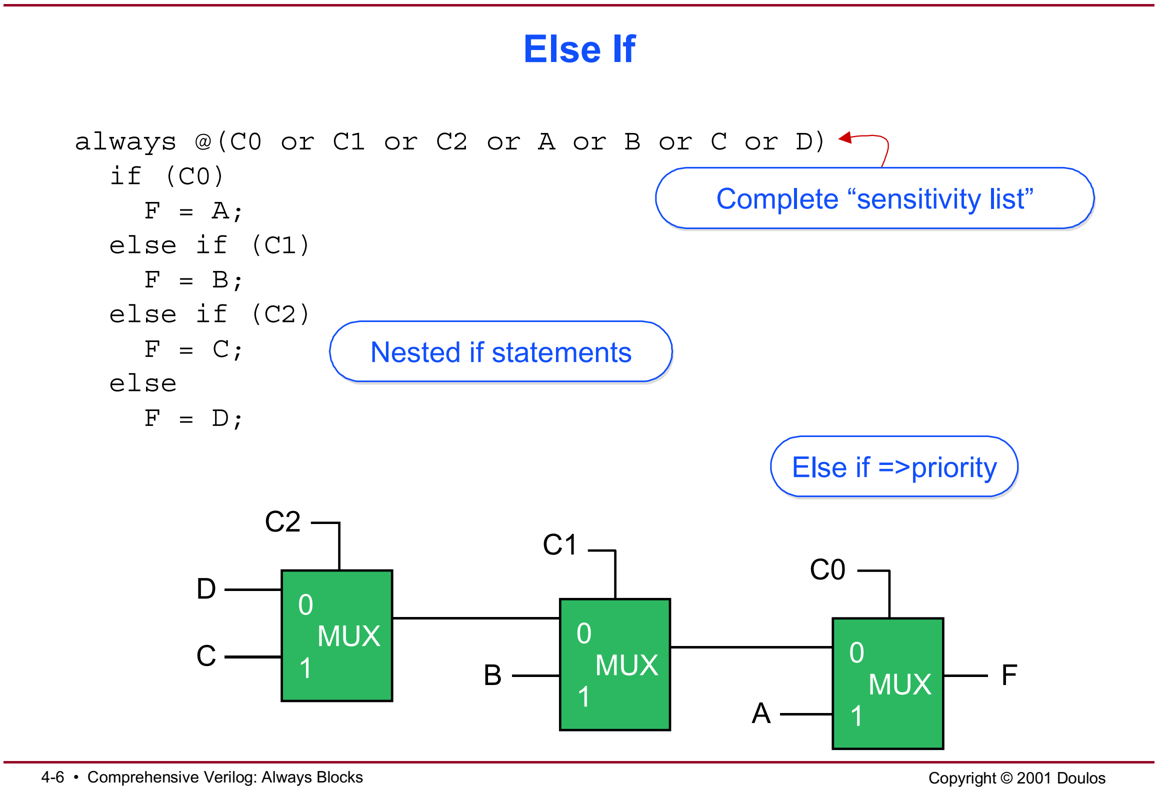
4-5 • Comprehensive Verilog: Always Blocks

Notes:

The syntax of the if statement allows for one statement in each branch. If more than one statement is needed, then the statements must be bracketed using begin and end. Here, two statements are executed in each branch, so begin-ends are required in both parts.

Note that an if statement, including its else and the statements in each of the two branches is considered as a single statement. Therefore the always statement opposite does not require a begin-end (although one could be used, as shown)

Else If



Else If

```
always @(C0 or C1 or C2 or A or B or C or D)
  if (C0)
    F = A;
  else if (C1)
    F = B;
  else if (C2)
    F = C;
  else
    F = D;
```

Complete “sensitivity list”

Nested if statements

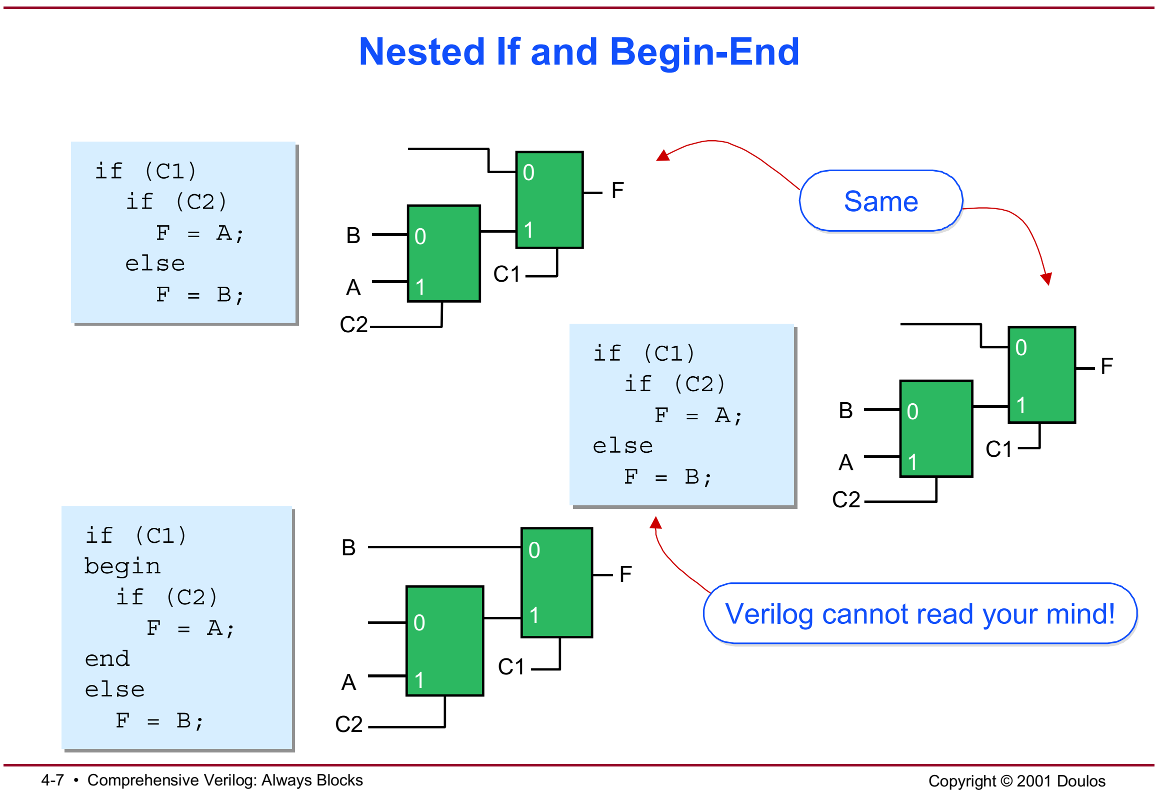
Else if =>priority

4-6 • Comprehensive Verilog: Always Blocks

Notes:

Testing a series of conditions is achieved by nesting if statements. This is best written as shown, with successive else if statements. Each condition is tested in turn until a condition is found which evaluates to true, then that branch of the nested if statement is executed, with the effect that earlier conditions take priority over later conditions.

Nested If and Begin-End



Nested If and Begin-End

```
if (C1)
  if (C2)
    F = A;
  else
    F = B;
```

Same

```
if (C1)
  if (C2)
    F = A;
else
  F = B;
```

```
if (C1)
begin
  if (C2)
    F = A;
end
else
  F = B;
```

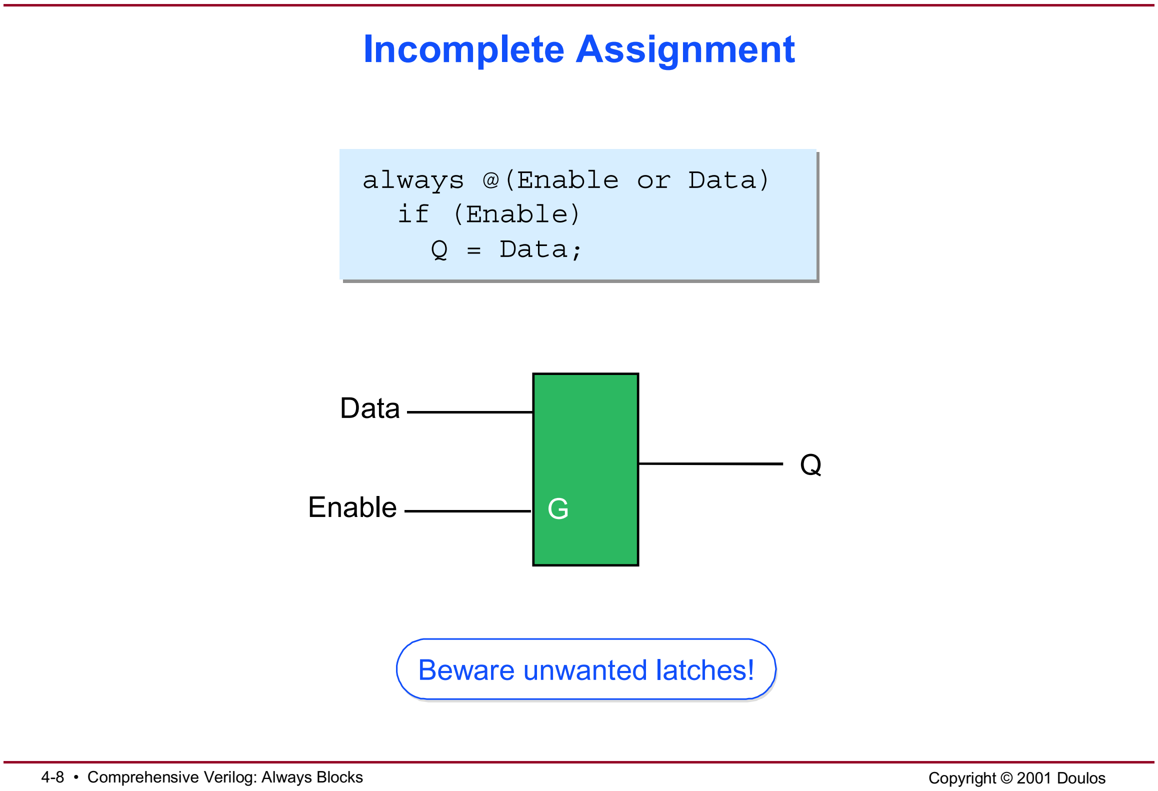
Verilog cannot read your mind!

4-7 • Comprehensive Verilog: Always Blocks

Notes:

Begin-end are sometimes needed to tell the Verilog compiler which if an else belongs to. In the examples above the two top-most if statements are the same, despite the different indentation. The begin-end in the third example "hides" the inner if from the else. Compilers don't look at the indentation! Nor do they read your mind!

Incomplete Assignment



Incomplete Assignment

```
always @(Enable or Data)
  if (Enable)
    Q = Data;
```

Beware unwanted latches!

4-8 • Comprehensive Verilog: Always Blocks

Notes:

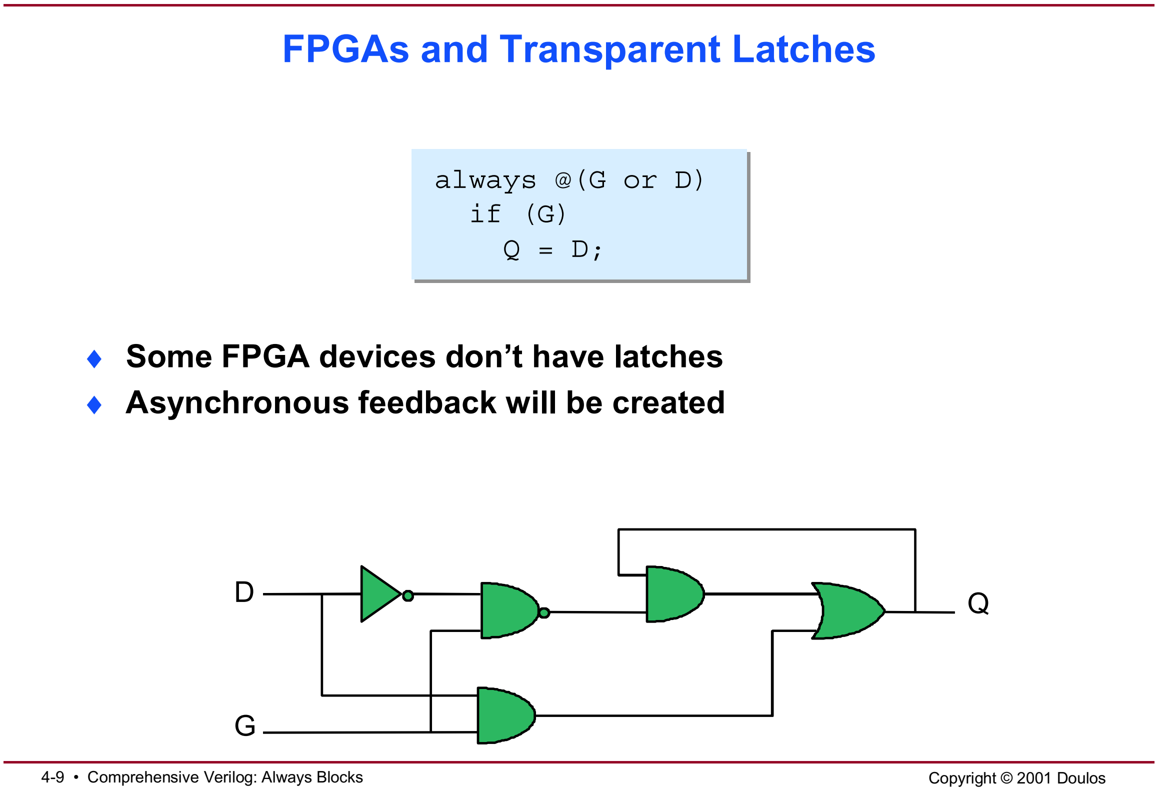
If a reg is only assigned under certain input conditions, then it will retain its old value under other input conditions.You can think of the always statement shown above as behaving like this:

```
always @(Enable or Data)
  if (Enable)
    Q = Data;
  else
```

```
Q = Q;
```

Synthesis tools will handle this by inserting a transparent latch to store the old value. This sometimes happens accidentally when trying to describe combinational logic using a procedural block, and can result in the synthesis of unwanted latches.

FPGAs and Transparent Latches



FPGAs and Transparent Latches

```
always @(G or D)
  if (G)
    Q = D;
```

- Some FPGA devices don’t have latches
- Asynchronous feedback will be created

4-9 • Comprehensive Verilog: Always Blocks

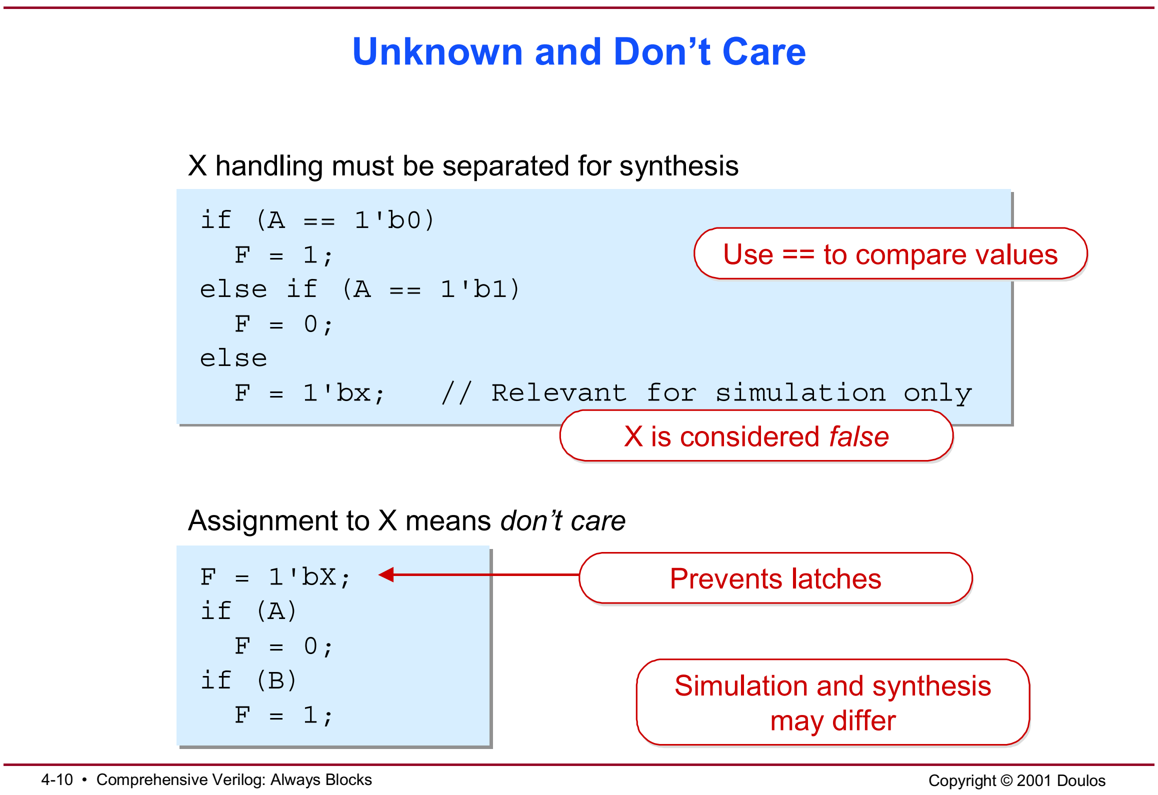
Copyright © 2001 Doulos

Notes:

FPGAs

Some FPGA architectures do not include transparent latches. To implement a transparent latch therefore requires the use of asynchronous feedback, and their use is discouraged in some FPGA devices; it is therefore a good idea to check the data sheet for your chosen device before you start writing your Verilog code.

Unknown and Don’t Care



Unknown and Don’t Care

X handling must be separated for synthesis

```
if (A == 1'b0)
  F = 1;
else if (A == 1'b1)
  F = 0;
else
  F = 1'bx;   // Relevant for simulation only
```

Use == to compare values

X is considered *false*

Assignment to X means *don’t care*

```
F = 1'bX;
if (A)
  F = 0;
if (B)
  F = 1;
```

Prevents latches

Simulation and synthesis may differ

4-10 • Comprehensive Verilog: Always Blocks

Notes:

Comparison with X is treated as FALSE by simulation and synthesis. You can write if statements to take account of possible X values, by explicitly testing for 0 and 1, as shown in the example above. The third condition is ignored by synthesis tools, because the hardware has been fully described.

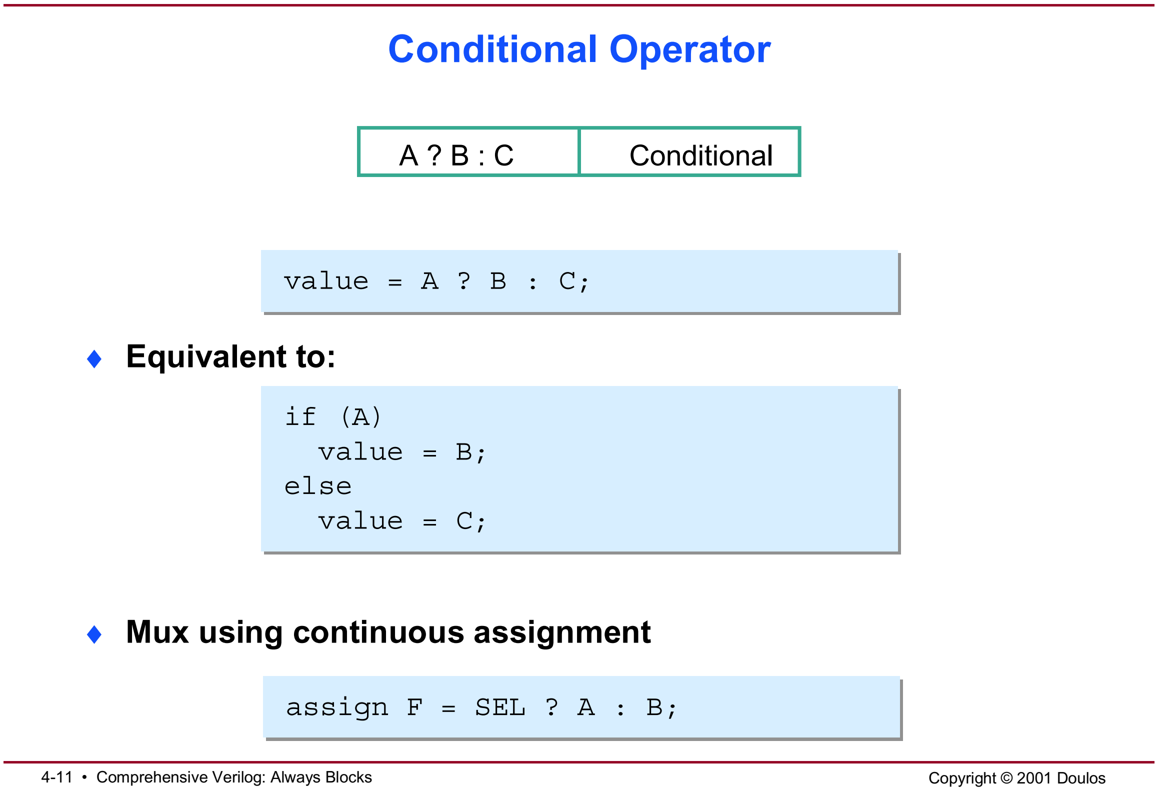
In practice, RTL code does not usually include X handling like this, because Xs are not usually generated, except when used as "don't care" values.

Explicit assignment to X is treated as "don't care" by synthesis tools. This may enable the synthesis tool to optimize more effectively, but with the drawback that the simulation results will differ between the RTL model and synthesized gates.

This is because in RTL simulation F may have the value 1'bX, when in simulation of the synthesized netlist, F is 1'b0 or 1'b1.

This example introduces the comparison operator ==. This is used in expressions to compare two values. The expression A == 1'b0 will have the value 1'b1 (TRUE) if A has the value 1'b0 and 1'b0 or 1'bX (FALSE) otherwise.

Conditional Operator



Conditional Operator

A ? B : C	Conditional

```
value = A ? B : C;
```

- **Equivalent to:**

```
if (A)
  value = B;
else
  value = C;
```

- **Mux using continuous assignment**

```
assign F = SEL ? A : B;
```

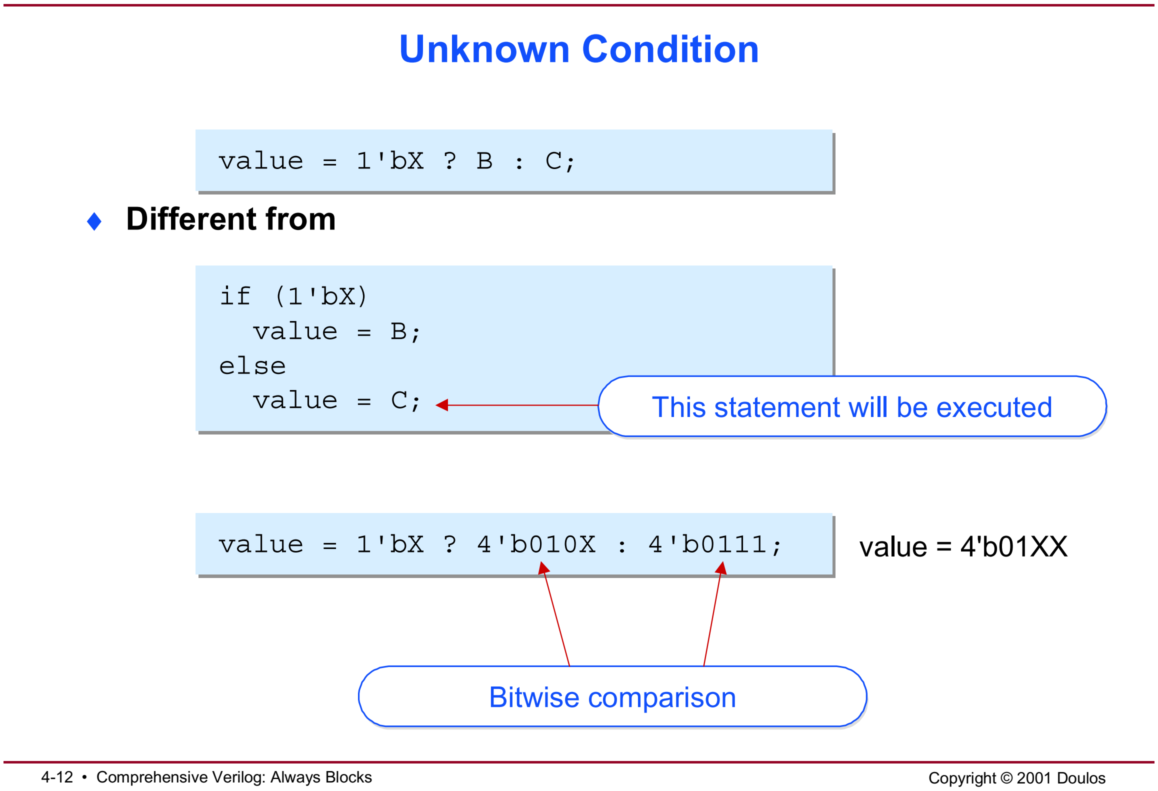
Notes:

As an alternative to using an if statement, Verilog provides the conditional operator. This can be used in expressions in initial and always statements and in continuous assignments.

The conditional operator consists of two characters, ? and :, and three operands. The value of an expression that uses this operator depends on the value of the first operand, which comes before the ?. If this operand is TRUE (non zero), the value of the whole expression is the value of the second operand (following the ?). If the first operand is FALSE, the value of the whole expression is the third operand (following the :).

The conditional operator is very useful in modeling multiplexors and tristate buffers.

Unknown Condition



Unknown Condition

```
value = 1'bX ? B : C;
```

- Different from

```
if (1'bX)
  value = B;
else
  value = C;
```

This statement will be executed

```
value = 1'bX ? 4'b010X : 4'b0111;
```

value = 4'b01XX

Bitwise comparison

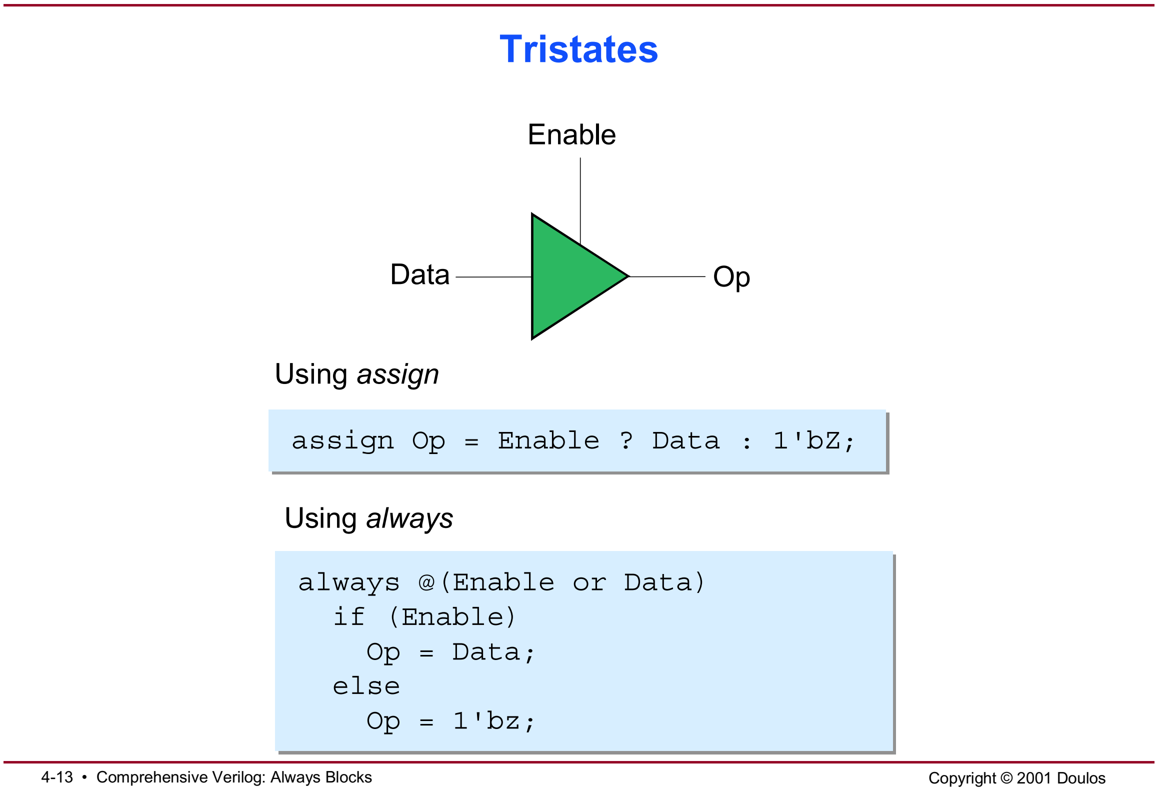
4-12 • Comprehensive Verilog: Always Blocks

Notes:

There is an important difference between the conditional operator and an if statement that is apparent when the condition is unknown. In an if statement, an unknown condition is considered to be false, and the else part of the if statement is executed. For the conditional operator, if the first operand is unknown, then the value of the conditional expression is formed by a bitwise comparison of the bits of the second and third operands. Where the values of the bits in corresponding positions are the same, the value of the corresponding bit in the result is that value. Where the bit values are different or if one of them is unknown, the corresponding bit in the result is X.

In the example shown, the resulting value is 4'b01XX because the two leftmost bits of 4'b010X and 4'b0111 are the same, the third bits are different, and one of the rightmost bits is X.

Tristates



Tristates

Using *assign*

```
assign Op = Enable ? Data : 1'bZ;
```

Using *always*

```
always @(Enable or Data)
  if (Enable)
    Op = Data;
  else
    Op = 1'bz;
```

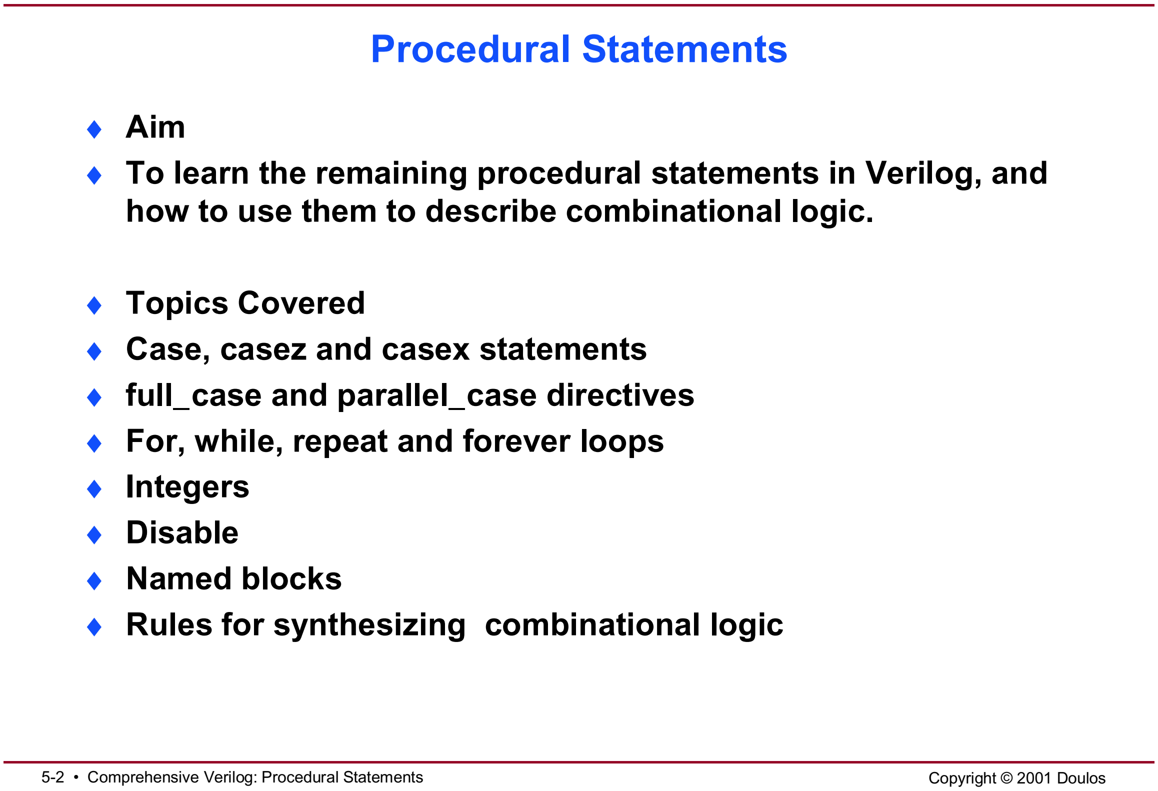
Notes:

Tristate drivers are inferred by assigning to Z. This can be done using a continuous assignment or an always statement.

When a tristate buffer is being used at an inout port, it is usually much easier to use a continuous assignment, similar to the one shown, rather than an always statement. This is because inout ports and connections to them must be nets (usually this means wires) and the target of a continuous assignment is a net.

Module 5
Procedural Statements

Procedural Statements



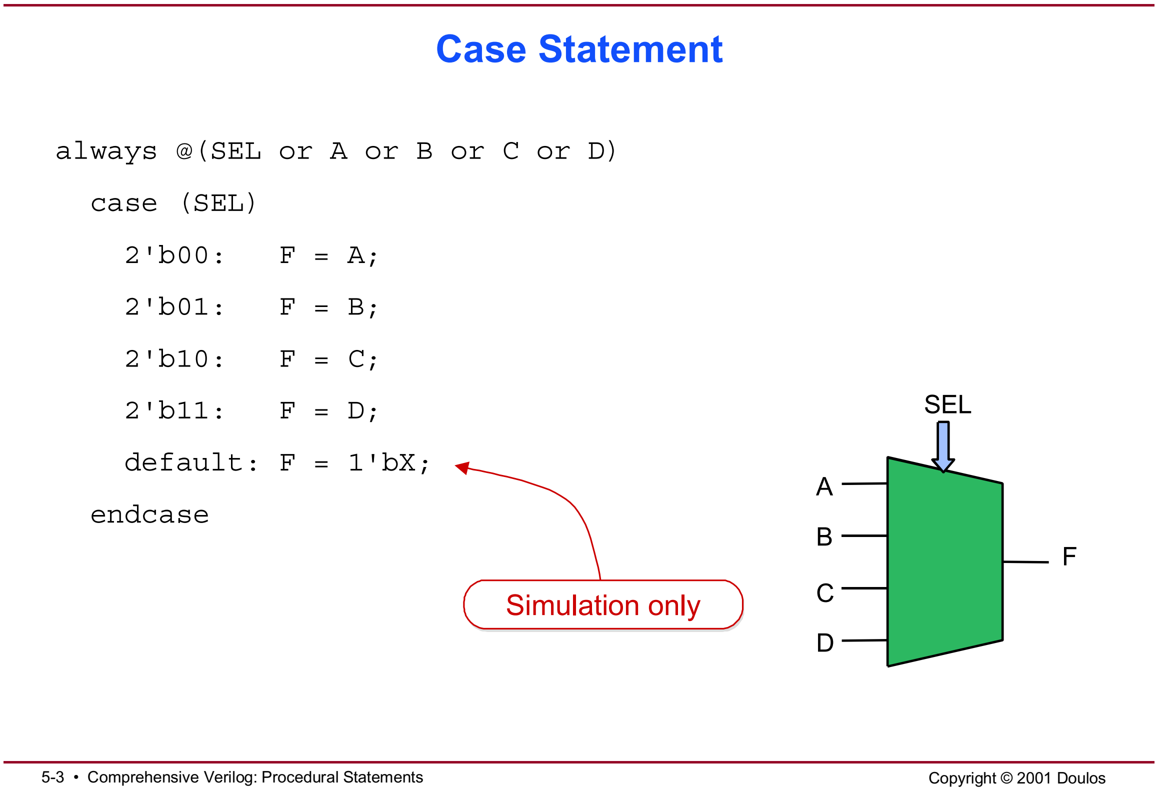
Procedural Statements

- Aim
- To learn the remaining procedural statements in Verilog, and how to use them to describe combinational logic.

- Topics Covered
- Case, casez and casex statements
- full_case and parallel_case directives
- For, while, repeat and forever loops
- Integers
- Disable
- Named blocks
- Rules for synthesizing combinational logic

Notes:

Case Statement



Case Statement

```
always @(SEL or A or B or C or D)
  case (SEL)
    2'b00:   F = A;
    2'b01:   F = B;
    2'b10:   F = C;
    2'b11:   F = D;
    default: F = 1'bX;
  endcase
```

Simulation only

Notes:

Case

The case statement is an efficient way of executing a multi-way branch. The case statement lists possible values of the expression at the top of the statement. When the case statement is executed, one and only one branch will be executed depending on the value of the expression. This will be the branch whose label matches the expression. Labels are separated from the statements by colons. The default branch will, if present, catch any cases not covered explicitly by the case statement. However, it is not necessary to cover all possible cases, so a default statement is always optional. In this example, the default statement is meaningless for synthesis, because all the hardware values that SEL can take have appeared explicitly as case labels.

Note that the selector of the case statement can be a vector and indeed any Verilog expression.

Synthesis

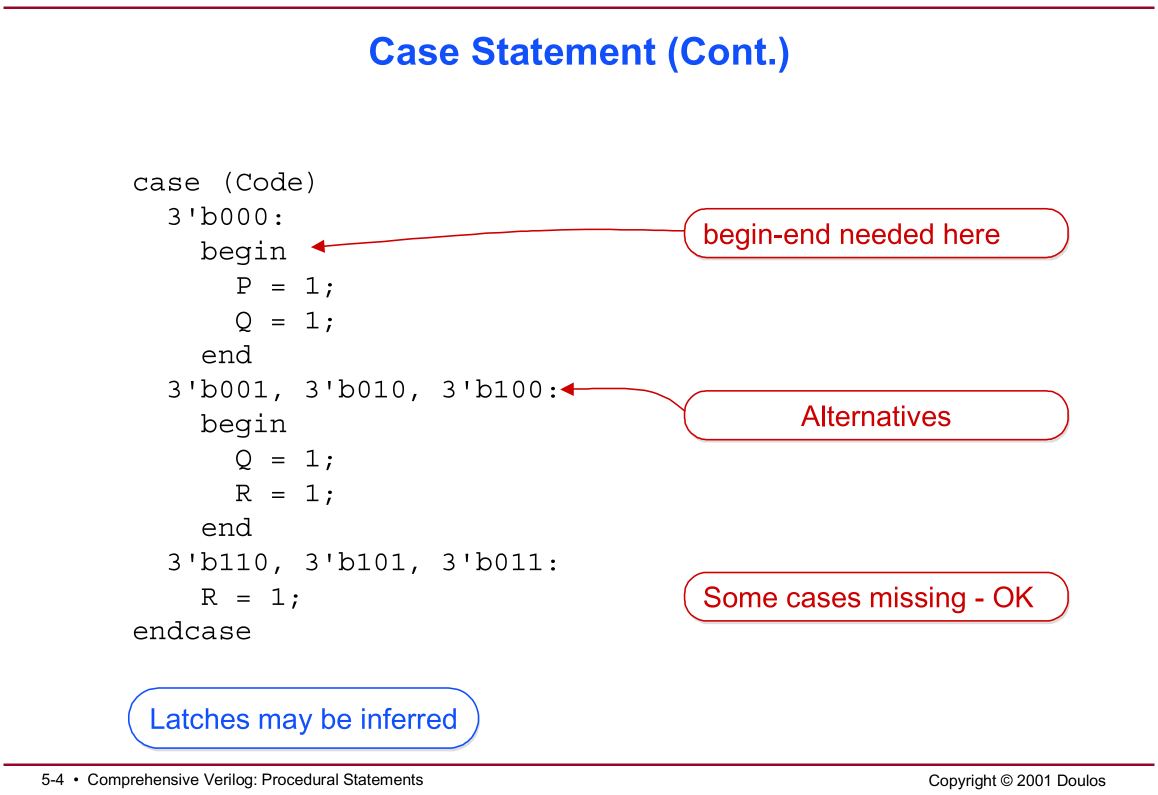
The case statement is synthesized in the same way as the if statement - by generating multiplexers for each reg assigned within the case.

The case statement provides a general and convenient way of describing the behavior of combinational logic so long as the number of inputs is fairly small.

FPGAs

It's important to remember that certain FPGA architectures are fanin limited; that is, the architecture is composed from logic blocks each of which has a small, fixed number of inputs. If the fanin limit is exceeded by the case statement, then it will be necessary to synthesis multiple logic blocks and multiple logic levels. Significant improvements in area and timing can be achieved by intelligently partitioning the logic to fit the architecture.

Case Statement (Cont.)



Case Statement (Cont.)

```
case (Code)
  3'b000:
    begin
      P = 1;
      Q = 1;
    end
  3'b001, 3'b010, 3'b100:
    begin
      Q = 1;
      R = 1;
    end
  3'b110, 3'b101, 3'b011:
    R = 1;
endcase
```

begin-end needed here

Alternatives

Some cases missing - OK

Latches may be inferred

5-4 • Comprehensive Verilog: Procedural Statements

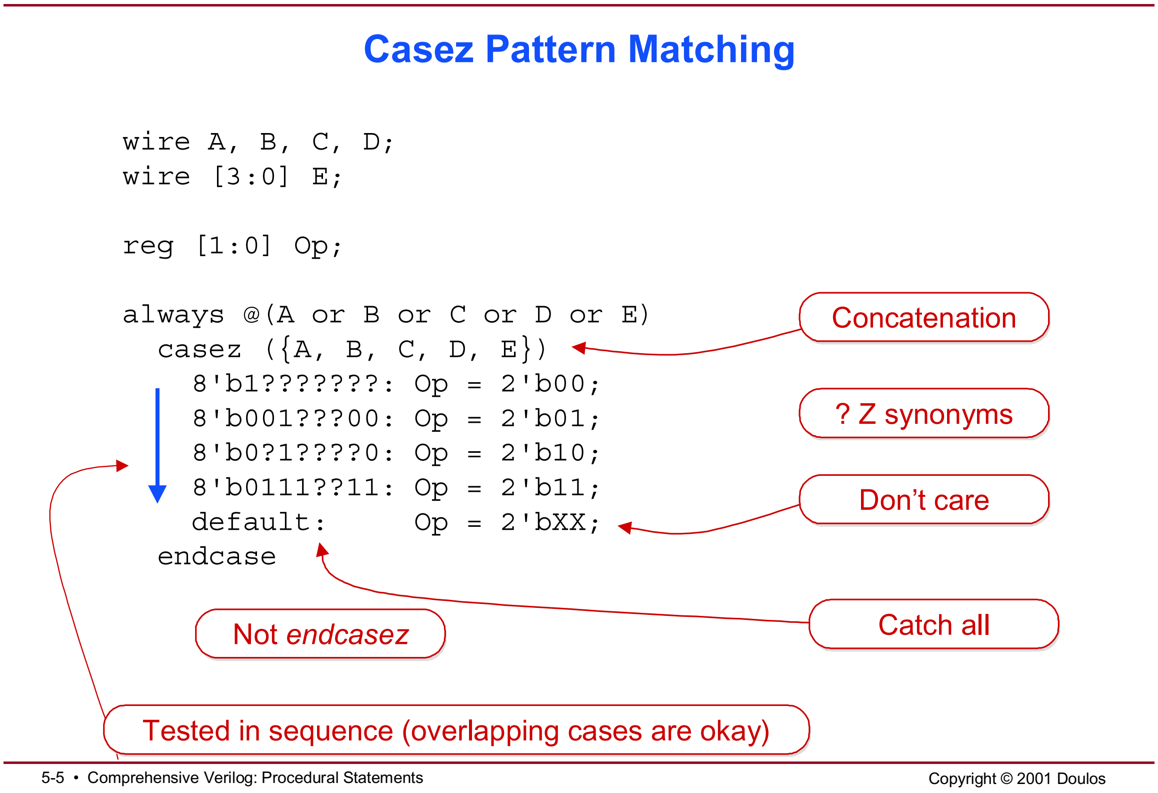
Copyright © 2001 Doulos

Notes:

This example shows some more features of the case statement. Two or more cases can be included in the same branch by separating the values with commas (3'b001, 3'b010, 3'b100 : ...). Several statements may be executed for a given value of the expression by including the statements in a begin ... end block.

As with if statements, latches will be synthesized if any register is incompletely assigned in the always block.

Casez Pattern Matching



Casez Pattern Matching

```
wire A, B, C, D;
wire [3:0] E;

reg [1:0] Op;

always @(A or B or C or D or E)
  casez ({A, B, C, D, E})
    8'b1???????: Op = 2'b00;
    8'b001???00: Op = 2'b01;
    8'b0?1????0: Op = 2'b10;
    8'b0111??11: Op = 2'b11;
    default:     Op = 2'bXX;
  endcase
```

Concatenation

? Z synonyms

Don’t care

Catch all

Not *endcasez*

Tested in sequence (overlapping cases are okay)

5-5 • Comprehensive Verilog: Procedural Statements

Notes:

A variation of the case statement is casez. The syntax and behavior are the same as those of the case statement, except that the value Z in a number represents a "don't care" value. For this reason the character ? can be used in place of Z in any number, because it more clearly shows the meaning in this context.

In the example shown, the first label matches any value of the case expression whose MSB is 1.- it matches whenever A is 1.

The second and third labels here overlap. For example, 8'b00100000 matches both 8'b001???00 and 8'b0?1????0. The first one has the priority, so Op would be set to 2'b01 in this case.

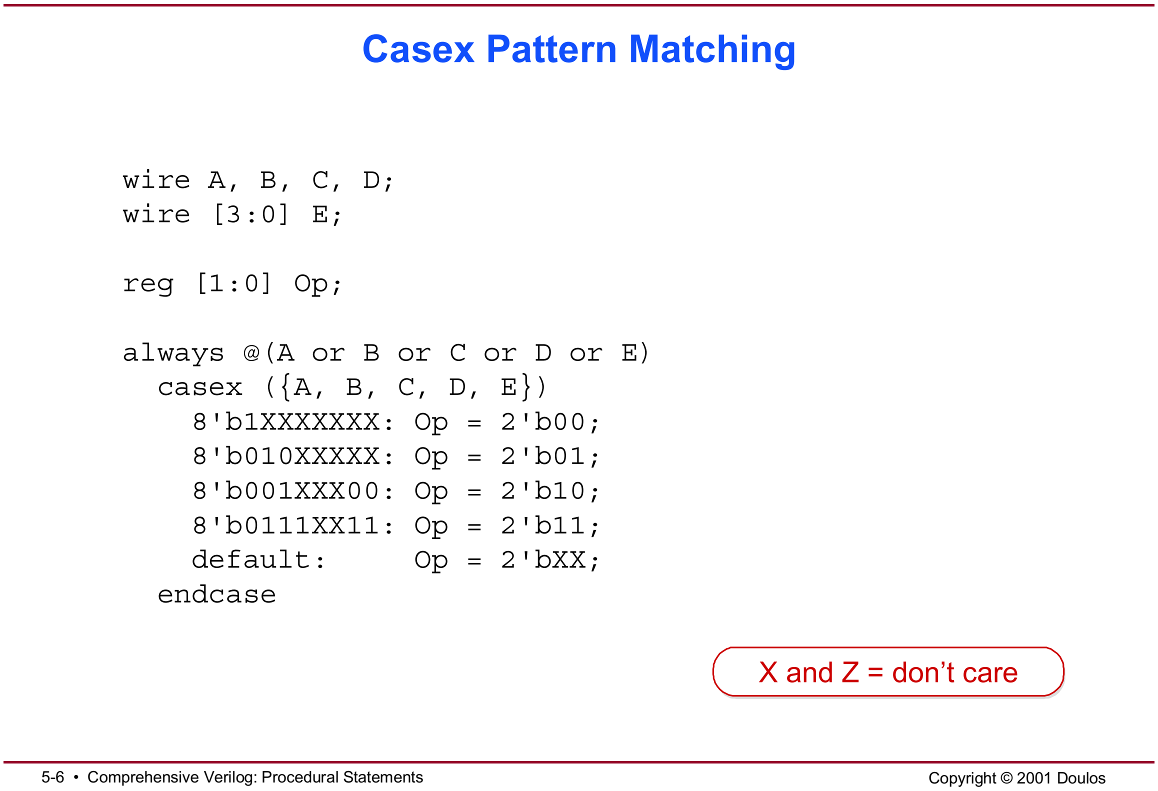
The casez statement in this example could be written using an if statement:

```
if (A)
  Op = 2'b00;
else if (~A & ~B & C & ~E[1] & ~E[0])
  Op = 2'b01;
else if (~A & C & ~E[0])
  Op = 2'b10;
else if (~A & B & C & D & E[1] & E[0])
  Op = 2'b11;
else
  Op = 2'bxx;
```

Concatenation

Note also the use of concatenation in the case expression. The case expression is a vector made up from the elements listed inside the curly brackets ({}).

Casex Pattern Matching



Casex Pattern Matching

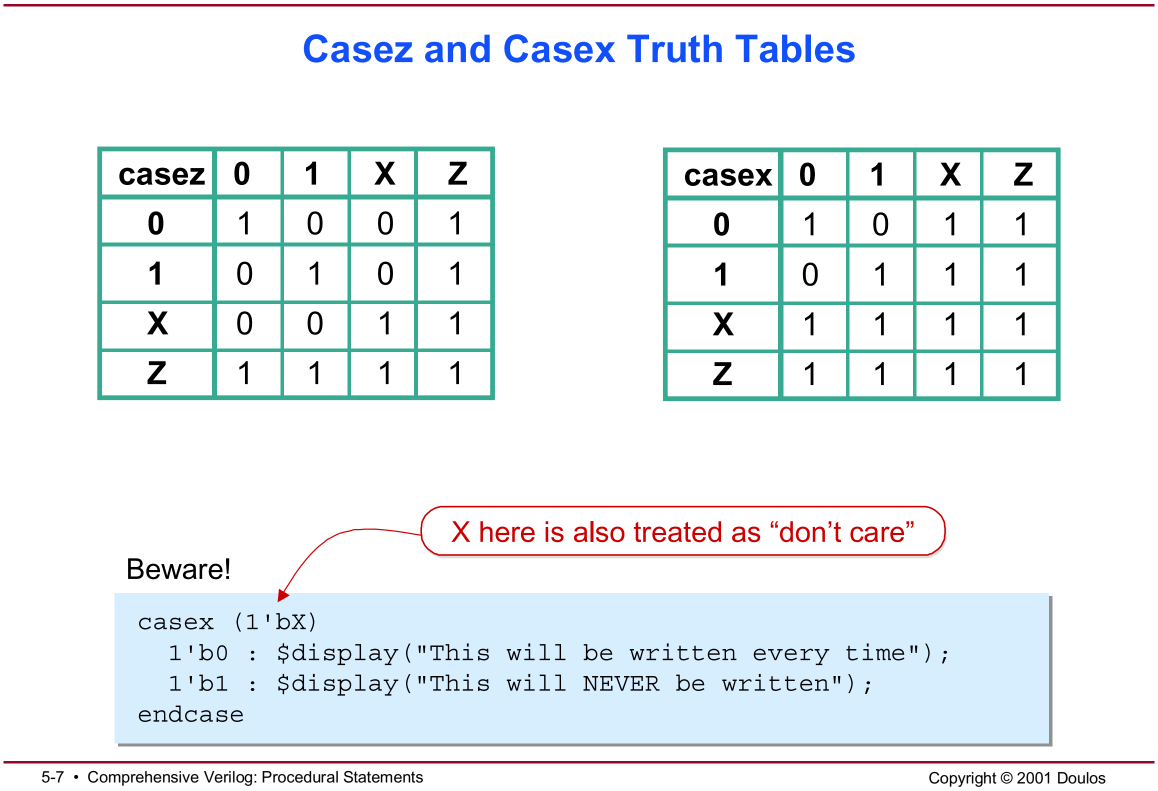
```
wire A, B, C, D;
wire [3:0] E;

reg [1:0] Op;

always @(A or B or C or D or E)
  casex ({A, B, C, D, E})
    8'b1XXXXXXX: Op = 2'b00;
    8'b010XXXXX: Op = 2'b01;
    8'b001XXX00: Op = 2'b10;
    8'b0111XX11: Op = 2'b11;
    default:     Op = 2'bXX;
  endcase
```

X and Z = don’t care

Notes:

Another variation is the casex. This is similar in concept to the casez, the difference being that both X and Z are pattern matching characters.

Casez and Casex Truth Tables



Casez and Casex Truth Tables

casez	0	1	X	Z
0	1	0	0	1
1	0	1	0	1
X	0	0	1	1
Z	1	1	1	1

casex	0	1	X	Z
0	1	0	1	1
1	0	1	1	1
X	1	1	1	1
Z	1	1	1	1

X here is also treated as “don’t care”

Beware!

```
casex (1'bX)
  1'b0 : $display("This will be written every time");
  1'b1 : $display("This will NEVER be written");
endcase
```

5-7 • Comprehensive Verilog: Procedural Statements

Copyright © 2001 Doulos

Notes:

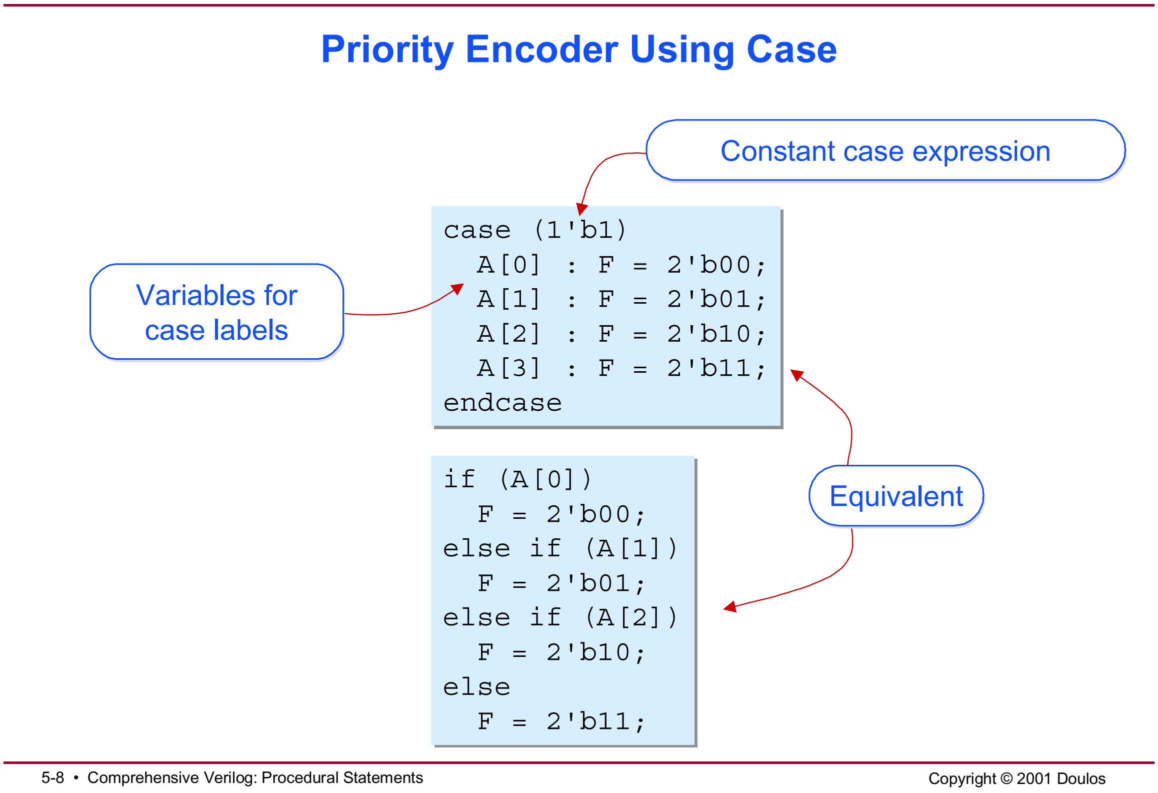
Here are the truth tables for comparing values in casez and casex statements. As has been explained, Z matches any value in a casez statement; Z and X match any value in a casex statement.

casez versus casex

Note that unlike the casez, a casex cannot be used to explicitly detect X's in an expression. This is because even any Xs and Zs in the case expression are treated as wildcards (i.e. they match any value).

For this reason casex is not recommended, because if the case expression is unknown during simulation (which is possible), it will match the first case value, and the corresponding statement will be executed. The same problem may arise with a casez statement, if the case expression contains Zs, but this is less likely to occur.

Priority Encoder Using Case



Priority Encoder Using Case

Constant case expression

Variables for case labels

```
case (1'b1)
  A[0] : F = 2'b00;
  A[1] : F = 2'b01;
  A[2] : F = 2'b10;
  A[3] : F = 2'b11;
endcase
```

Equivalent

```
if (A[0])
  F = 2'b00;
else if (A[1])
  F = 2'b01;
else if (A[2])
  F = 2'b10;
else
  F = 2'b11;
```

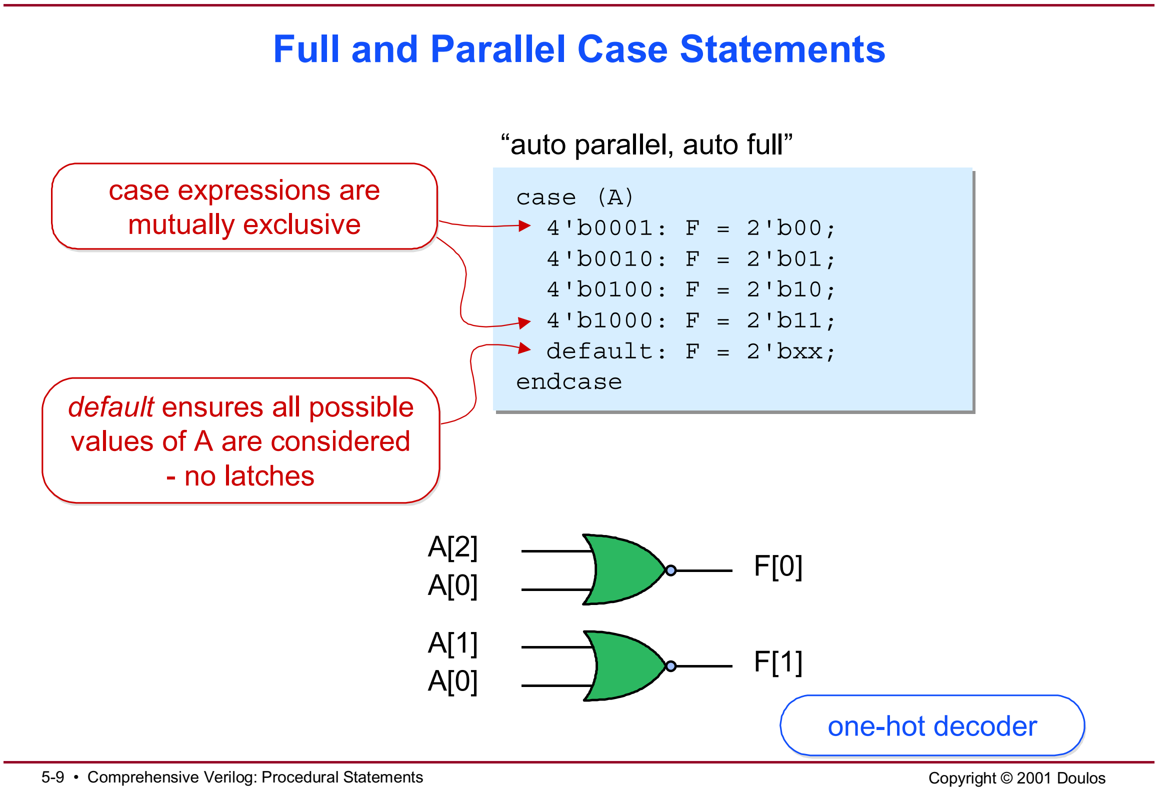
5-8 • Comprehensive Verilog: Procedural Statements

Notes:

One unusual feature of the Verilog case statement syntax is that the case expression (at the top) is allowed to be a constant, and the case values (inside) may be variables!

In this style of case statement, it is possible that the case expression may match more than one case value. If so, the first one - starting at the top - is used. In other words, the case statement is being used to describe priority, and is equivalent to a set of nested if-else statements.

Full and Parallel Case Statements



Full and Parallel Case Statements

"auto parallel, auto full"

```
case (A)
  4'b0001: F = 2'b00;
  4'b0010: F = 2'b01;
  4'b0100: F = 2'b10;
  4'b1000: F = 2'b11;
  default: F = 2'bxx;
endcase
```

case expressions are mutually exclusive

default ensures all possible values of A are considered - no latches

one-hot decoder

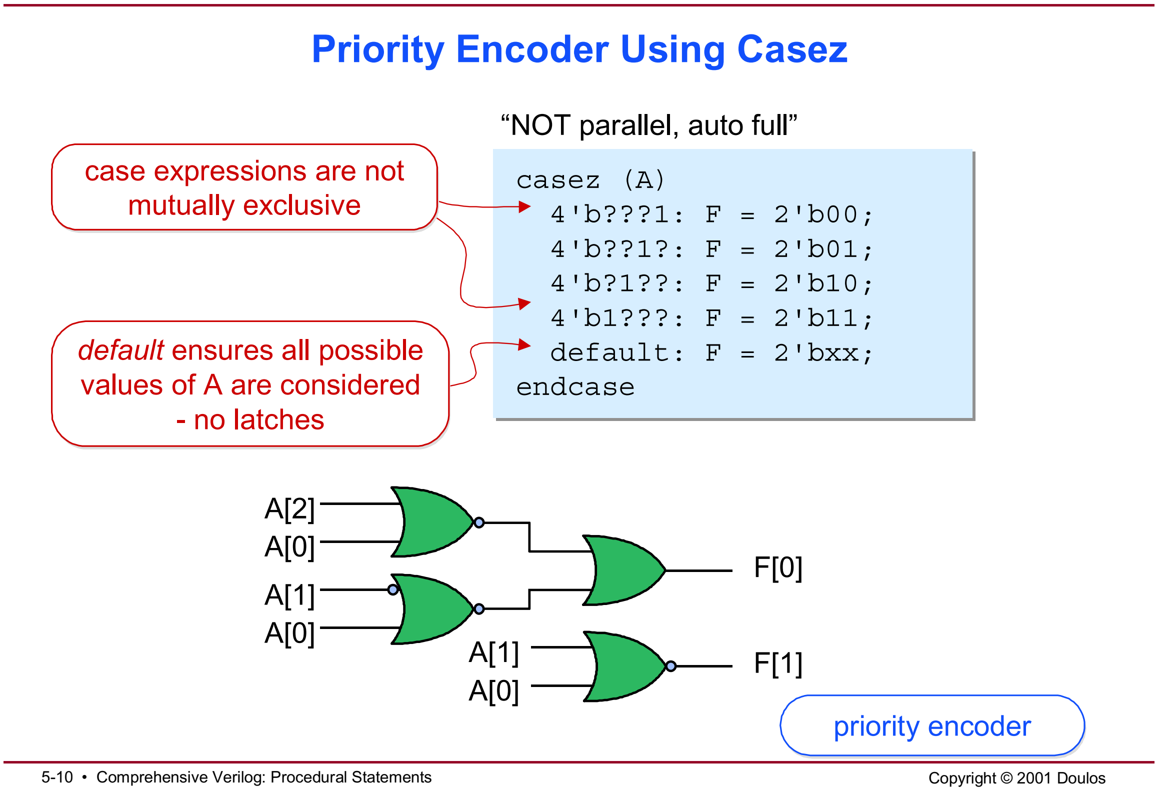
Notes:

If a case statement contains labels that are mutually exclusive, it is said to be "parallel"; if it covers all possible cases, it is said to be "full". A full and parallel case statement infers one or more multiplexers. Synthesis tools are usually able to recognize full and parallel case statements automatically.

Where the synthesis tool is not able to recognize that the case labels are mutually exclusive, a priority encoder is synthesized instead. If it is known that the labels are in fact mutually exclusive, the synthesis tool must be told that this is indeed so, so that a multiplexer will be synthesized.

Similarly, where the labels do not appear to cover every possible case, but do in fact do so in the context of the design, it is possible to tell the synthesis tool, so that unwanted default logic or latches are not synthesized.

Priority Encoder Using Casez



Priority Encoder Using Casez

"NOT parallel, auto full"

```
casez (A)
   4'b???1: F = 2'b00;
   4'b??1?: F = 2'b01;
   4'b?1??: F = 2'b10;
   4'b1???: F = 2'b11;
   default: F = 2'bxx;
endcase
```

case expressions are not mutually exclusive

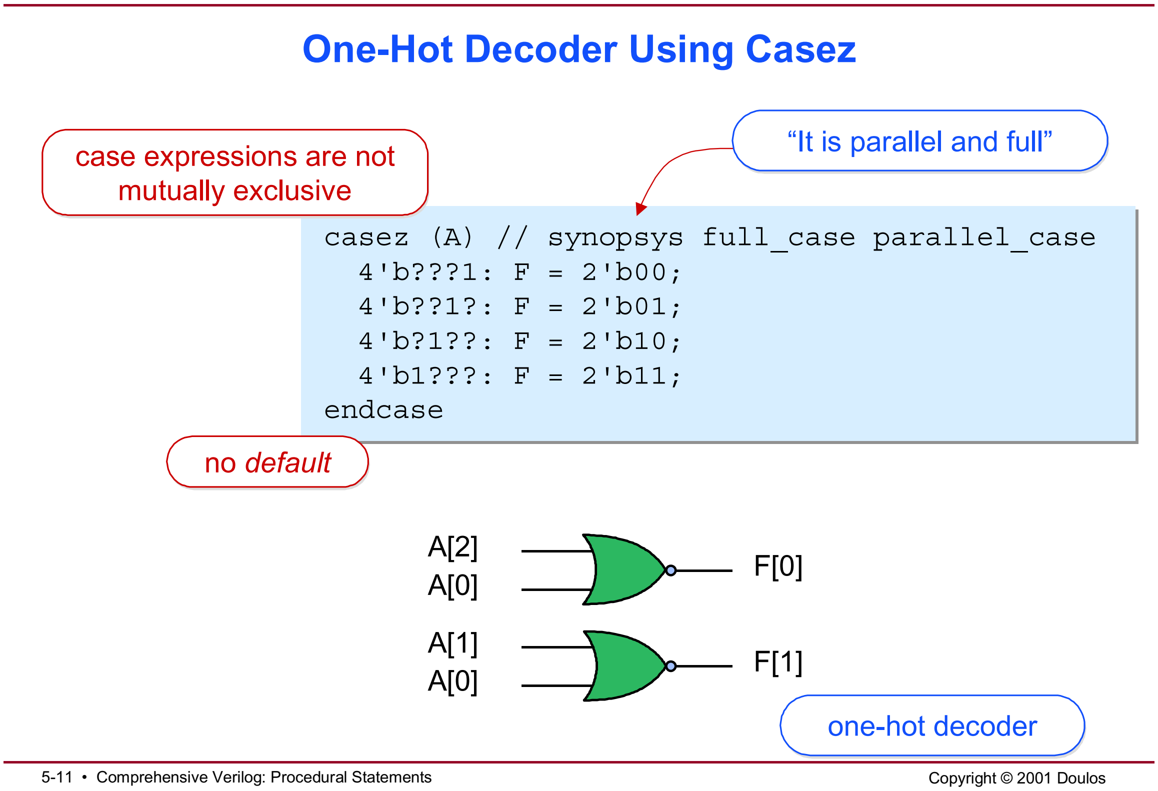
default ensures all possible values of A are considered - no latches

priority encoder

Notes:

This casez statement is full (it has a default), but it is not parallel. For example, if all the bits of A were 1, this would in fact match all the labels; the priority rules state that the first statement (only) would be executed, so a priority encoder is being described.

One-Hot Decoder Using Casez



One-Hot Decoder Using Casez

case expressions are not mutually exclusive

"It is parallel and full"

```
casez (A) // synopsys full_case parallel_case
  4'b???1: F = 2'b00;
  4'b??1?: F = 2'b01;
  4'b?1??: F = 2'b10;
  4'b1???: F = 2'b11;
endcase
```

no *default*

one-hot decoder

5-11 • Comprehensive Verilog: Procedural Statements

Notes:

Suppose that in fact the value of A is known to have a one-hot encoding, so that at any given time all the elements of A are 0, except one. The synthesis tool must be told that this is the case, otherwise unnecessary logic will be inferred.

Synthesis Directive

One way of doing this is to include a synthesis directive in the Verilog code. This is a Verilog comment that is ignored by simulators (it is just a comment...), but which is read and actioned by synthesis tools (so it's more than a comment!).

parallel_case

The parallel_case directive applies to a case statement, and tells the synthesis tool that the case statement has mutually exclusive case values.

full_case

The full_case directive also applies to a case statement. It tells the synthesis tool that the case statement does in fact cover all possible values of the case expression.

WARNING!

These directives must be used with great care, as they tell the synthesis tool and simulator to interpret the same Verilog code in different ways.

Most synthesis tools support these directives, but they all have a slightly different syntax.

// synopsys parallel_case

// synopsys full_case

// synopsys full_case parallel_case

// synthesis parallel_case

// synthesis full_case

// synthesis full_case parallel_case

// pragma parallel_case

// pragma full_case

// pragma full_case parallel_case

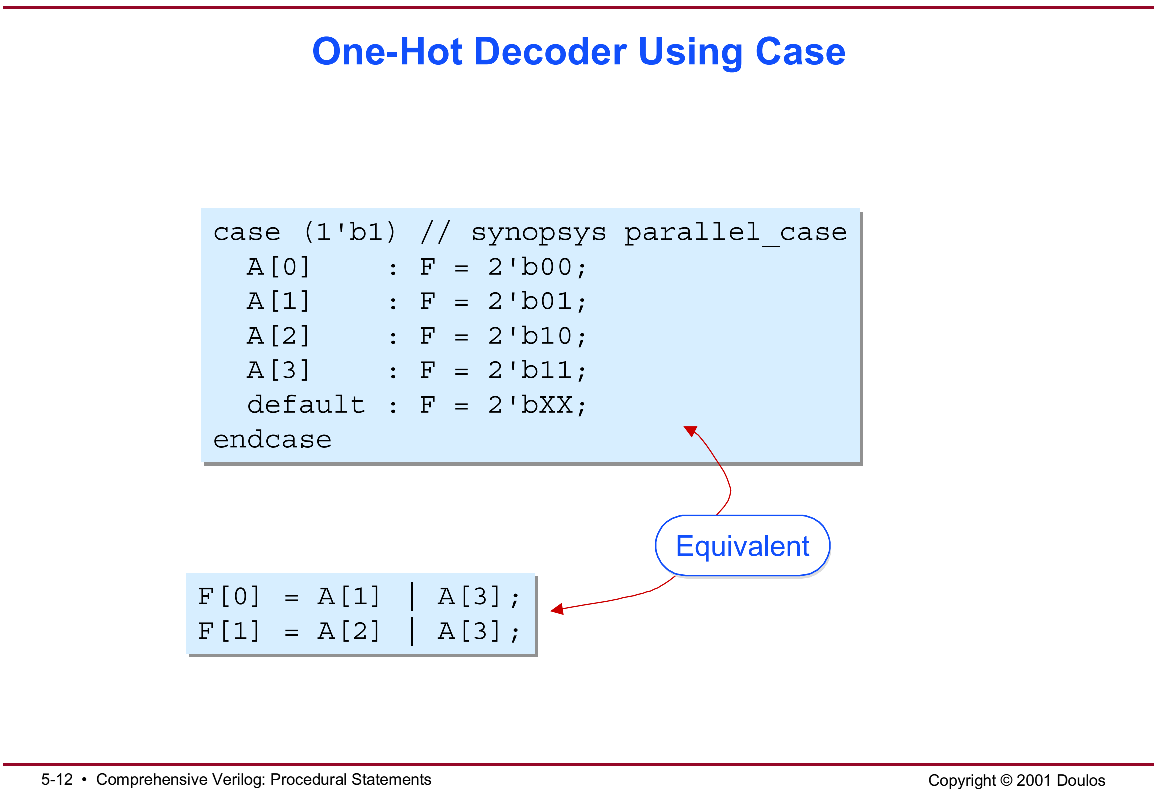
(These are the directives for Synopsys Design Compiler and FPGA Express; Synplicity Synplify and Exemplar Leonardo Spectrum respectively)

IEEE 1364.1

There is a proposal (IEEE 1364.1) to standardize these directives across different synthesis tools.

```
// rtl_synthesis parallel_case

// rtl_synthesis full_case

// rtl_synthesis full_case, parallel_case
```

One-Hot Decoder Using Case



One-Hot Decoder Using Case

```
case (1'b1) // synopsys parallel_case
  A[0]     : F = 2'b00;
  A[1]     : F = 2'b01;
  A[2]     : F = 2'b10;
  A[3]     : F = 2'b11;
  default  : F = 2'bXX;
endcase
```

Equivalent

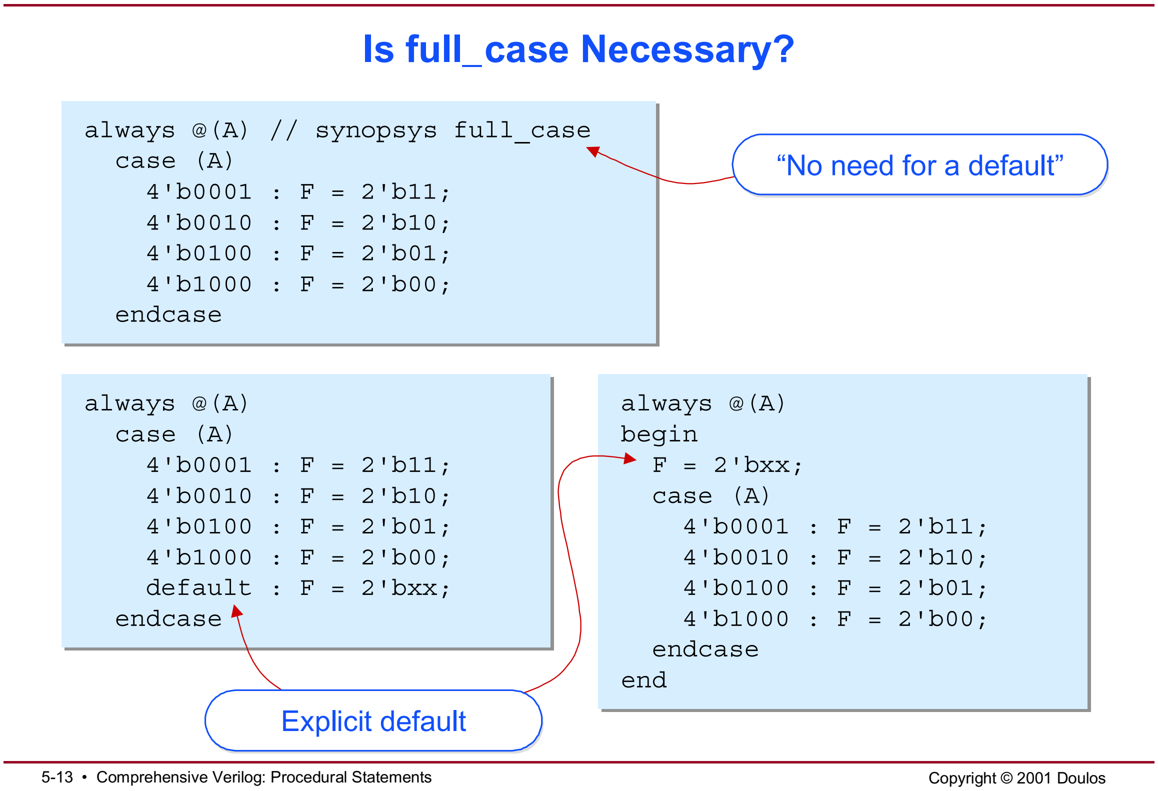
```
F[0] = A[1] | A[3];
F[1] = A[2] | A[3];
```

5-12 • Comprehensive Verilog: Procedural Statements

Notes:

This example shows a case statement with a parallel_case directive, being used to describe a one-hot decoder for synthesis.

Is full_case Necessary?



Is full_case Necessary?

```
always @(A) // synopsys full_case
  case (A)
    4'b0001 : F = 2'b11;
    4'b0010 : F = 2'b10;
    4'b0100 : F = 2'b01;
    4'b1000 : F = 2'b00;
  endcase
```

“No need for a default”

```
always @(A)
  case (A)
    4'b0001 : F = 2'b11;
    4'b0010 : F = 2'b10;
    4'b0100 : F = 2'b01;
    4'b1000 : F = 2'b00;
    default : F = 2'bxx;
  endcase
```

```
always @(A)
begin
  F = 2'bxx;
  case (A)
    4'b0001 : F = 2'b11;
    4'b0010 : F = 2'b10;
    4'b0100 : F = 2'b01;
    4'b1000 : F = 2'b00;
  endcase
end
```

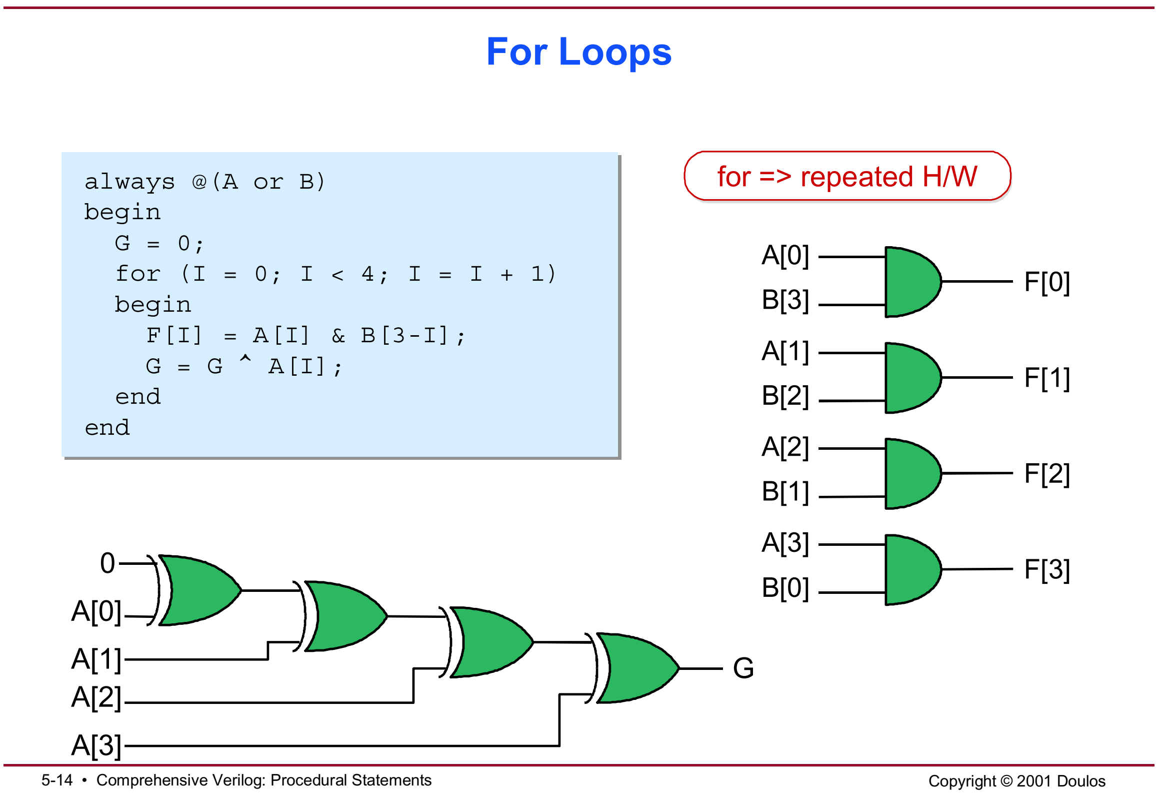
Explicit default

5-13 • Comprehensive Verilog: Procedural Statements

Notes:

No! You can achieve the same results by using "don't care" values. All three of the examples shown here result in the same logic (a one-hot decoder and no latches) being synthesized.

For Loops



For Loops

```
always @(A or B)
begin
  G = 0;
  for (I = 0; I < 4; I = I + 1)
  begin
    F[I] = A[I] & B[3-I];
    G = G ^ A[I];
  end
end
```

for => repeated H/W

5-14 • Comprehensive Verilog: Procedural Statements

Notes:

For loops are sequential statements that are often used in conjunction with vectors. The statements inside the loop are repeated for a number of values described by the iteration scheme at the top of the loop. In this example, the loop is executed 4 times with the loop parameter I taking the values 0, 1, 2 and 3 respectively.

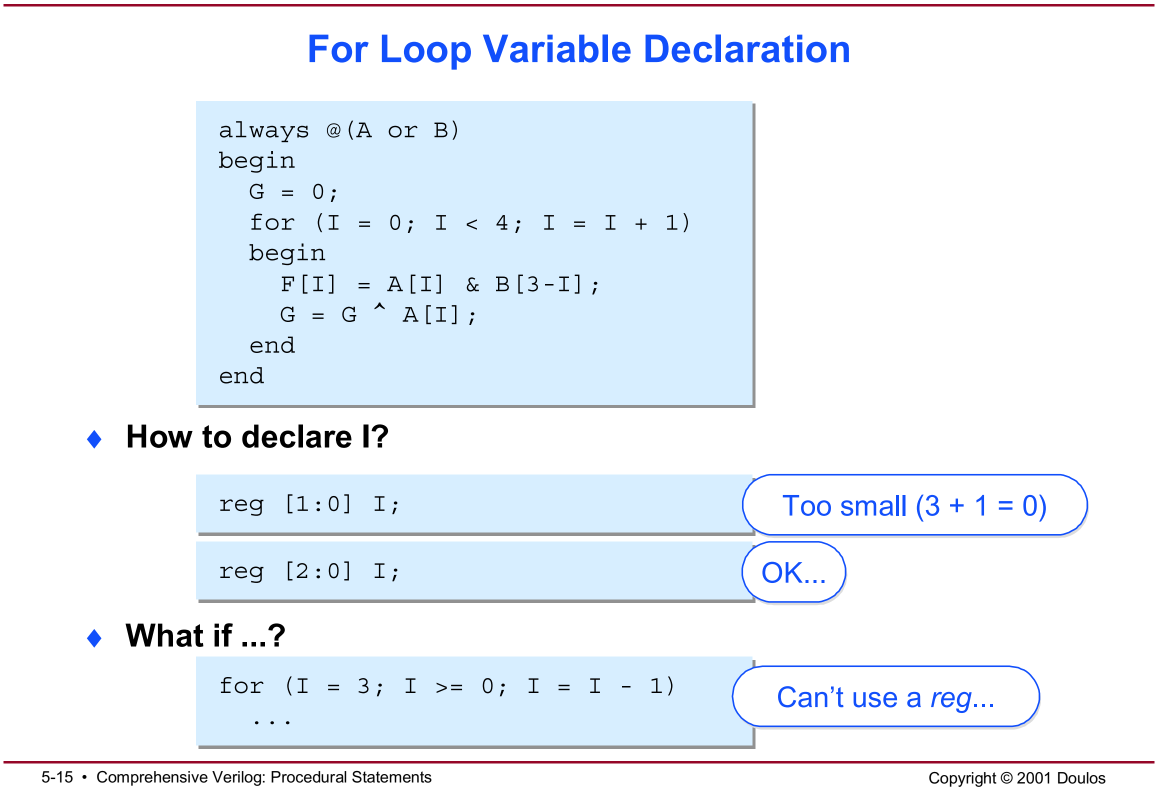
There is an important rule governing for loops. The value of the loop parameter can be read inside the loop, but should not be assigned, so you should not alter the execution of a loop by changing the value of the loop parameter inside the loop! (The IEEE Verilog standard does not enforce this rule, but you are strongly recommended to follow it.)

Synthesis

For loops are synthesized to repeated hardware structures, providing the bounds of the loop are fixed. In the example, the first assignment in the loop would result in the synthesis of 4 AND gates. The second assignment synthesizes to a cascaded structure of XOR gates, because the value assigned in one pass through the loop is consumed in the subsequent pass. In this case, the optimizer would be able to restructure the XORs into a tree to balance the delay paths. However, in more complex cases the optimizer may not be able to restructure such a cascaded logic structure, particularly if any arithmetic operations are included, so such loops must be written with care.

Synthesis is unable to handle loops with variable bounds, as the synthesizer is unable to determine how many copies of the loop contents to generate.

For Loop Variable Declaration



For Loop Variable Declaration

```
always @(A or B)
begin
  G = 0;
  for (I = 0; I < 4; I = I + 1)
  begin
    F[I] = A[I] & B[3-I];
    G = G ^ A[I];
  end
end
```

- **How to declare I?**

```
reg [1:0] I;
```

Too small (3 + 1 = 0)

```
reg [2:0] I;
```

OK...

- **What if ...?**

```
for (I = 3; I >= 0; I = I - 1)
  ...
```

Can’t use a *reg*...

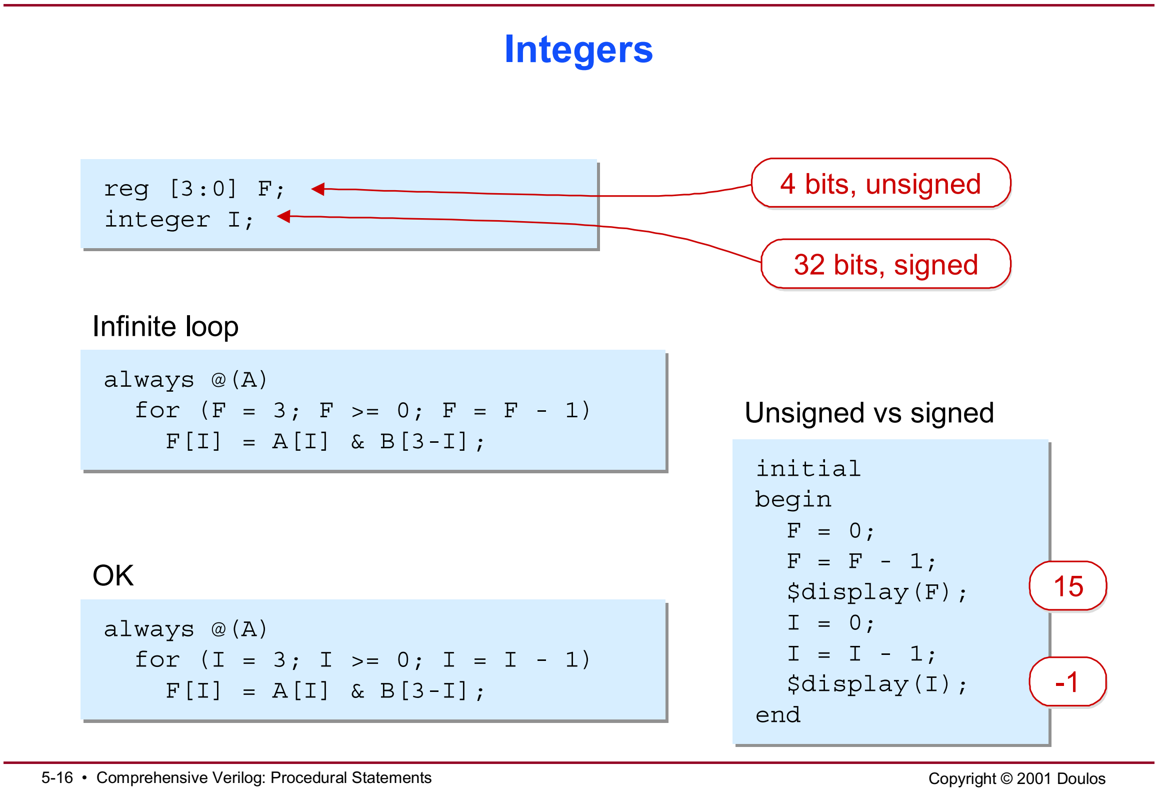
Notes:

A for loop variable such as I must be declared as a register. Here, it is tempting to declare I as reg [1:0], because I is being used to count from 0 to 3. This would not work, because regs are considered to have unsigned values, and two bits gives a maximum value of 3. The assignment I = I + 1 would cause the value of I to "roll over" when I has this maximum value. (2'b11 + 1 = 2'b00) So, the value of I would always be less than 4, and the loop would not terminate when simulated.

The problem can be solved for this example, by making sure that the width of I is sufficiently large. However, this would not work in a loop where the loop variable was decremented instead of being incremented: the value of a reg, being unsigned, is always greater than zero.

Fortunately, there is a solution to this problem, which is described in the next slide.

Integers



Integers

```
reg [3:0] F;
integer I;
```

4 bits, unsigned

32 bits, signed

Infinite loop

```
always @(A)
  for (F = 3; F >= 0; F = F - 1)
    F[I] = A[I] & B[3-I];
```

OK

```
always @(A)
  for (I = 3; I >= 0; I = I - 1)
    F[I] = A[I] & B[3-I];
```

Unsigned vs signed

```
initial
begin
  F = 0;
  F = F - 1;
  $display(F);
  I = 0;
  I = I - 1;
  $display(I);
end
```

15

-1

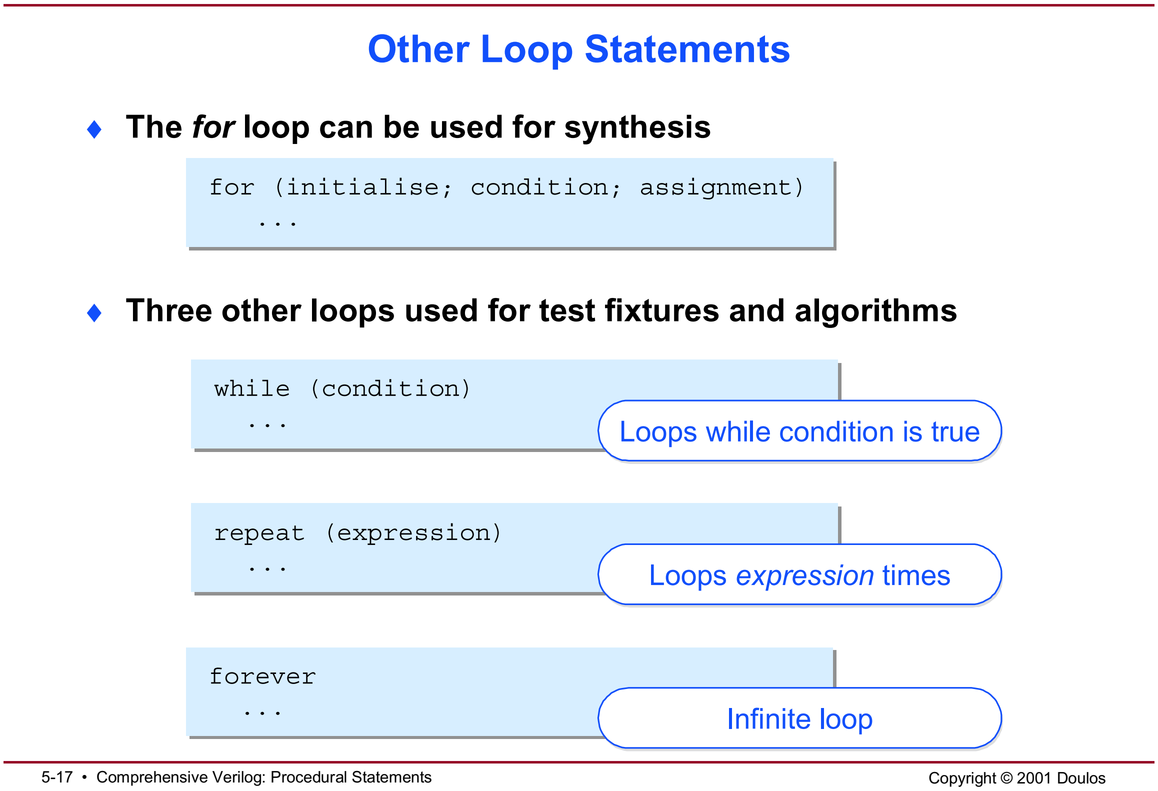
5-16 • Comprehensive Verilog: Procedural Statements

Notes:

Verilog has a special type of register used to represent a mathematical integer. The value stored in a Verilog data type integer is exactly the same as the value stored in the data type reg [31:0] , but the values are treated differently in any context where numbers are required; integers are interpreted as two's complement signed numbers, whereas regs are interpreted as unsigned binary numbers.

The data type reg should be used to represent hardware, whereas the data type integer should be used for loop variables and wherever a pure mathematical number is required (e.g. a counter in a test fixture).

Other Loop Statements



Other Loop Statements

- The *for* loop can be used for synthesis

```
for (initialise; condition; assignment)
   ...
```

- Three other loops used for test fixtures and algorithms

```
while (condition)
   ...
```

Loops while condition is true

```
repeat (expression)
   ...
```

Loops *expression* times

```
forever
   ...
```

Infinite loop

5-17 • Comprehensive Verilog: Procedural Statements

Notes:

In addition to for loops, Verilog includes three other loop statements: repeat loops, while loops and forever loops.

Unlike the for loop, these loop statements should be considered not to be synthesizable. They are useful in test fixtures and high-level behavioral models.

while

A while loop executes as long as the condition is true (i.e. not zero).

repeat

A repeat loop executes a fixed number of times, depending on the value of the repeat expression. Repeat loops may be synthesizable in the same way as for loops, although not all tools support this.

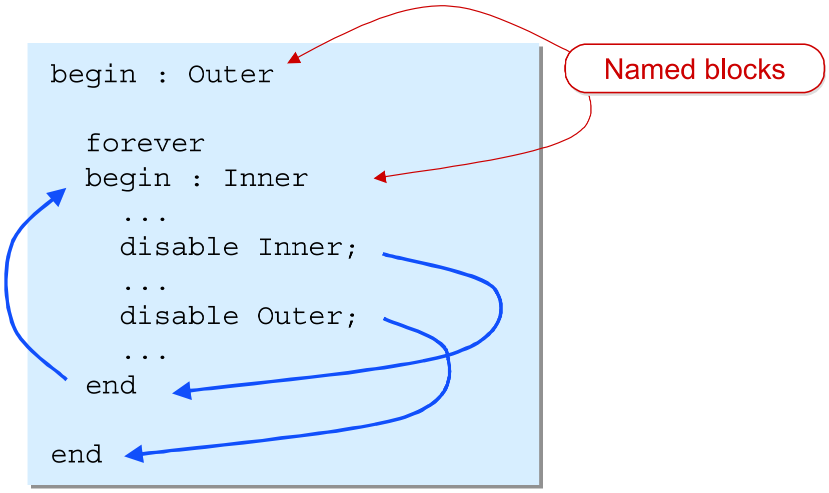
forever

A forever loop keeps executing unconditionally. It may only be interrupted by using the disable statement, as described on the next page.

Disable

Disable

- The *disable* statement can be used to jump out of a block



```
begin : Outer

  forever
  begin : Inner
    ...
    disable Inner;
    ...
    disable Outer;
    ...
  end

end
```

Notes:

The disable statement causes a begin-end block to stop executing. This has the effect that when the disable is executed, the next statement to be executed will be the statement immediately following the block being disabled.

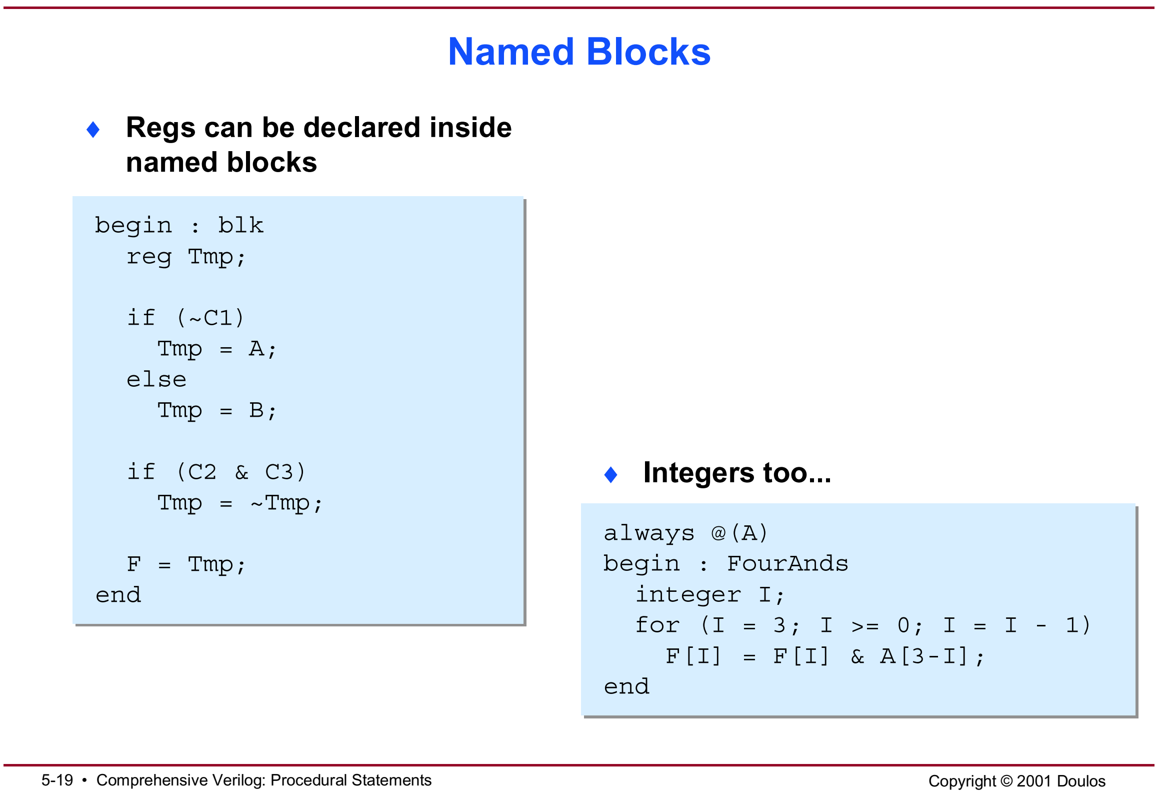
Named Block

To be disabled, a begin-end block must have a name. A begin-end block is named by following the word begin with a colon and the name.

Disable provides a means of breaking out of loops, either to continue with the next iteration of the loop, or to stop executing the loop altogether. (C.f. the continue and break statements in C, or the next and exit statements in VHDL.) This is illustrated opposite where disable Inner in effect jumps to the next loop iteration, and disable Outer jumps out of the loop altogether.

Disable may also be used as a return statement from a task. This will be described later in the course.

Named Blocks



Named Blocks

- Regs can be declared inside named blocks

```
begin : blk
  reg Tmp;

  if (~C1)
    Tmp = A;
  else
    Tmp = B;

  if (C2 & C3)
    Tmp = ~Tmp;

  F = Tmp;
end
```

- Integers too...

```
always @(A)
begin : FourAnds
  integer I;
  for (I = 3; I >= 0; I = I - 1)
    F[I] = F[I] & A[3-I];
end
```

5-19 • Comprehensive Verilog: Procedural Statements

Copyright © 2001 Doulos

Notes:

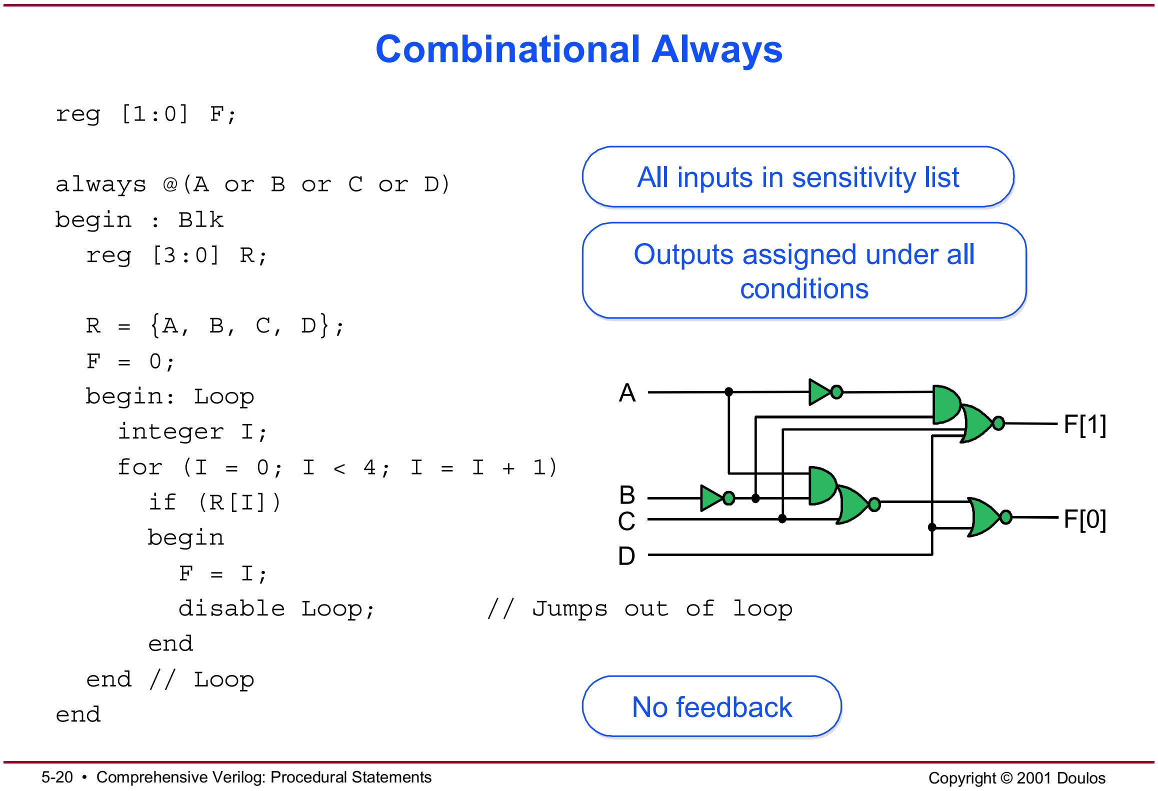
If a begin-end block has a name, it is possible to declare a register, such as a reg or an integer, inside it. This indicates that the register so declared can only be used inside that block.

Scope

In software engineering, the term scope is used to describe this concept. The scope of an identifier (such as a reg) is the region of the program text within which the identifier's characteristics are understood. The scope of the reg Tmp in this example extends from the its declaration to the end of the named begin-end block. Two or more identifiers can be declared with the same name, provided that the declarations are in different scopes. It is illegal to declare the same name more than once in the same scope.

Verilog defines four regions of scope: modules, named blocks, tasks and functions. Tasks and functions will be described in a later section.

Combinational Always



Combinational Always

```
reg [1:0] F;

always @(A or B or C or D)
begin : Blk
  reg [3:0] R;

  R = {A, B, C, D};
  F = 0;
  begin: Loop
    integer I;
    for (I = 0; I < 4; I = I + 1)
      if (R[I])
      begin
        F = I;
        disable Loop;       // Jumps out of loop
      end
  end // Loop
end
```

All inputs in sensitivity list

Outputs assigned under all conditions

No feedback

5-20 • Comprehensive Verilog: Procedural Statements

Notes:

The example above shows an always statement being used to infer combinational logic for synthesis. Note the use of a wide range of Verilog statements, including vectors, concatenation, a for loop and a disable statement.

The diagram shows the synthesized gates, after optimization.

An always statement is synthesized to combinational logic if the following conditions are true:

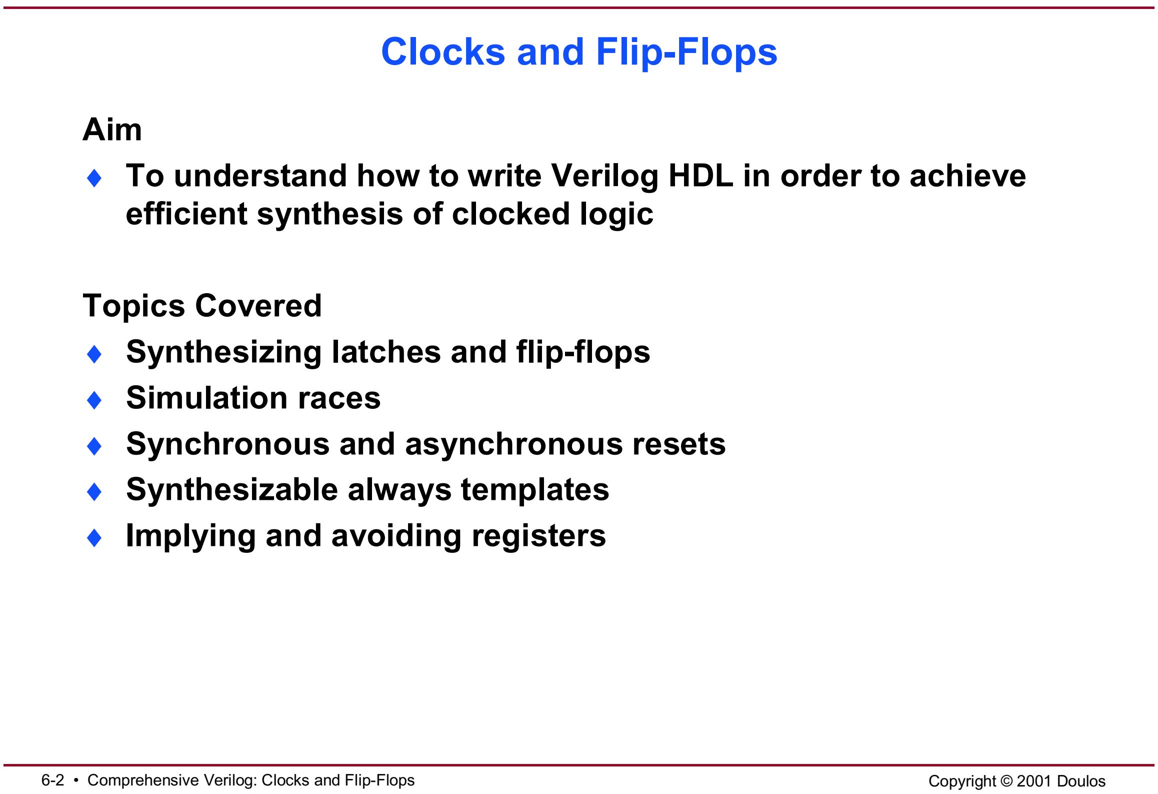
- The sensitivity list is complete. In other words all the combinational inputs must be listed after the @ timing control at the top of the always statement..

- Outputs are completely assigned. Every time the always block is executed (i.e. whenever one of the inputs changes value), all the outputs must be assigned a value. If this is not true, then it is not combinational but sequential logic that is being described and unwanted latches may be synthesized.

- There is no combinational feedback. Again, this would result in asynchronous sequential behavior.

Module 6
Clocks and Flip-Flops

Clocks and Flip-Flops



Clocks and Flip-Flops

Aim

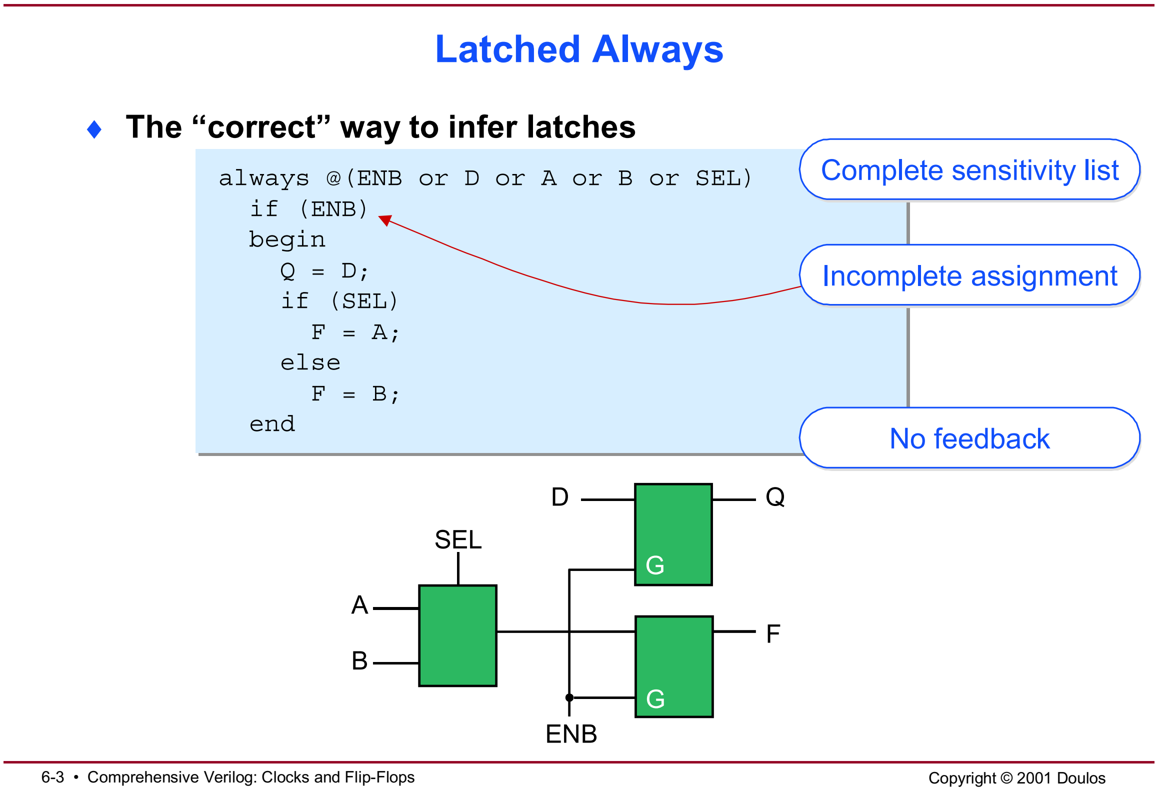
- To understand how to write Verilog HDL in order to achieve efficient synthesis of clocked logic

Topics Covered

- Synthesizing latches and flip-flops
- Simulation races
- Synchronous and asynchronous resets
- Synthesizable always templates
- Implying and avoiding registers

Notes:

Latched Always



Latched Always

- **The “correct” way to infer latches**

```
always @(ENB or D or A or B or SEL)
  if (ENB)
  begin
    Q = D;
    if (SEL)
      F = A;
    else
      F = B;
  end
```

Complete sensitivity list

Incomplete assignment

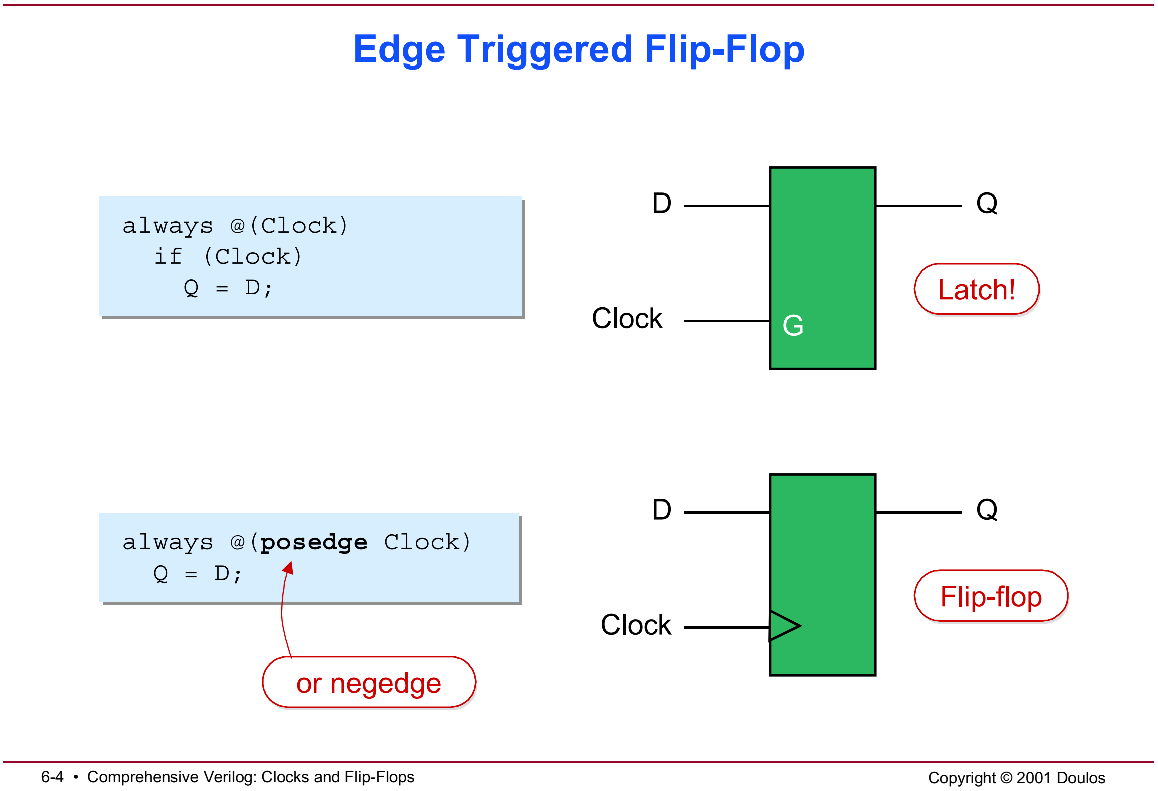
No feedback

Notes:

This always statement shows how to infer latches. The only difference between this and the style used to infer combinational logic is that the value of the output is deliberately not specified for every possible input change.

In order to infer latches predictably, use an if statement that is conditional on the latch enable, as shown here.

Edge Triggered Flip-Flop



Edge Triggered Flip-Flop

```
always @(Clock)
  if (Clock)
    Q = D;
```

```
always @(posedge Clock)
  Q = D;
```

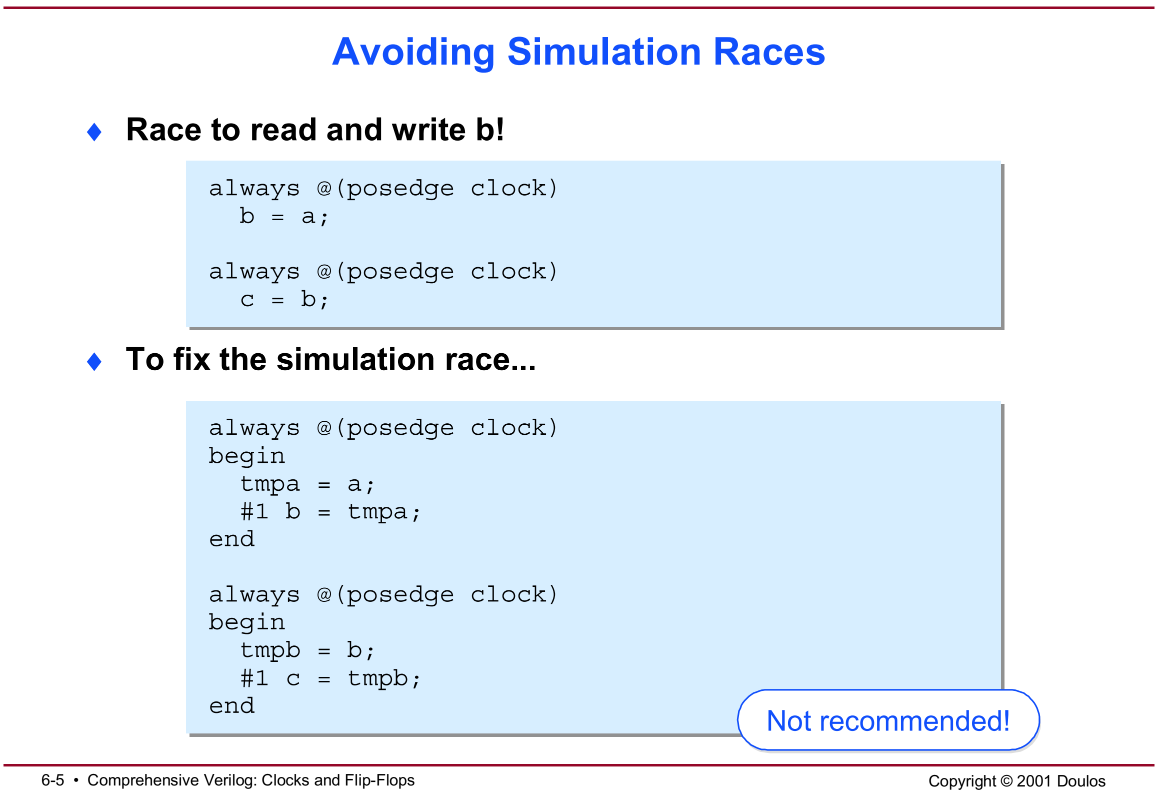
Notes:

Although the top example will simulate correctly as a D type flip-flop, it will be synthesized as a transparent latch (if it is synthesized at all). This is because synthesis tools will think the event control is incomplete, since D is missing. To infer a flip-flop requires that the synthesis tool is given an even bigger hint than correct functionality! The "hint" is the reserved word posedge or negedge.

It is if (and only if) an always statement contains the reserved word posedge or negedge in its event control, that flip-flops will be inferred.

The second example shows the simplest possible clocked always statement. It describes a single edge triggered (D type) flip-flop, clocked on the rising edge of Clock.

Avoiding Simulation Races



Avoiding Simulation Races

- Race to read and write b!

```
always @(posedge clock)
  b = a;

always @(posedge clock)
  c = b;
```

- To fix the simulation race...

```
always @(posedge clock)
begin
  tmpa = a;
  #1 b = tmpa;
end

always @(posedge clock)
begin
  tmpb = b;
  #1 c = tmpb;
end
```

Not recommended!

6-5 • Comprehensive Verilog: Clocks and Flip-Flops

Notes:

Non-determinism

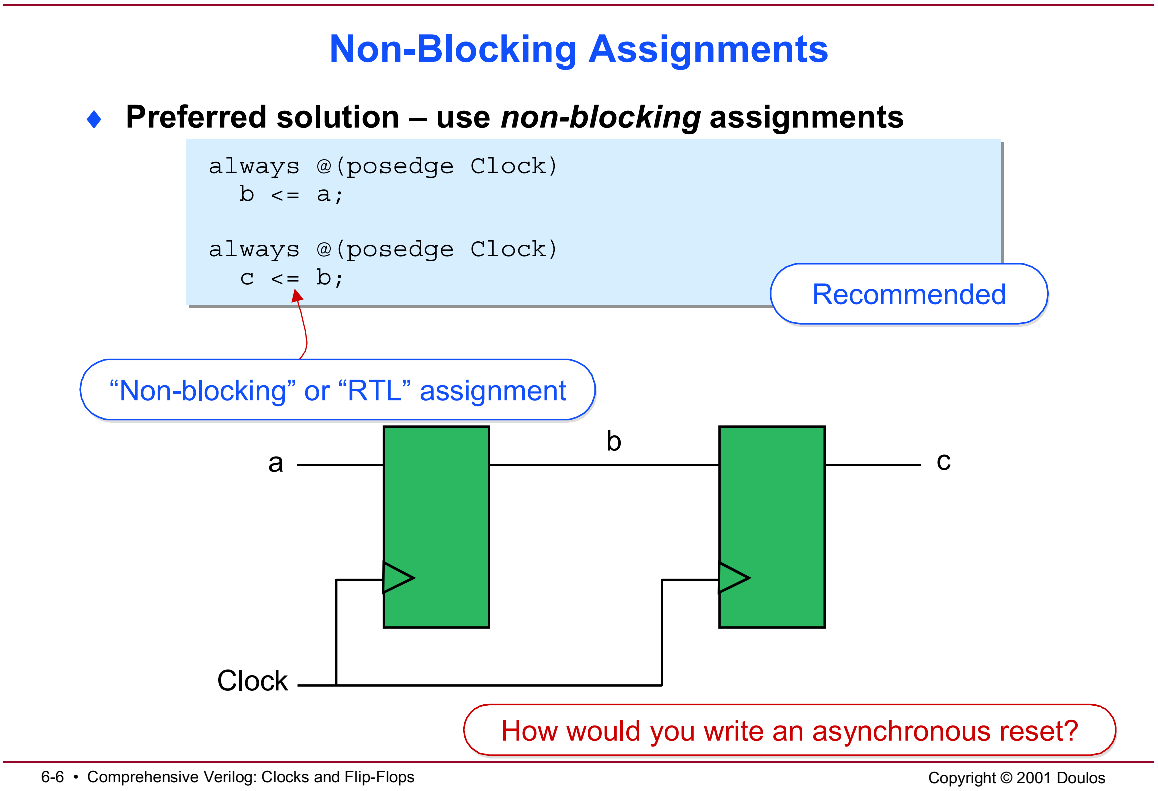
One characteristic of Verilog is that the language is non-deterministic. In other words, it is possible to write Verilog code that simulates differently on different simulators. This is not a fault with one of the simulators; the Verilog language has this feature.

An example of non-determinism occurs when describing flip-flops in separate always statements, as shown here. When a positive clock edge occurs in the simulation, both always statements will be executed. It is not possible to tell which one the simulator will execute first. If the statements are executed in the order shown, then b will be updated first, and the new value of b will be used to update c. So b and c will both have the same value. On the other hand, if the always statements are executed in reverse order (and there is nothing in the definition of Verilog to say that they can't be), then c will be given the old value of b, before b is updated with the value of a.

For synthesis, there is no ambiguity, and two flip-flops will be inferred. There may therefore be differences between the simulation results of the RTL description and the synthesized gates. In effect there is a hold time violation in the RTL model.

We could avoid this problem by using temporary regs and adding delays in the always statements, as shown. This would work (synthesis ignores delays, but they make the simulation correct), but it is clumsy and inelegant.

Non-Blocking Assignments



Non-Blocking Assignments

- Preferred solution – use *non-blocking* assignments

```
always @(posedge Clock)
  b <= a;

always @(posedge Clock)
  c <= b;
```

Recommended

“Non-blocking” or “RTL” assignment

How would you write an asynchronous reset?

Notes:

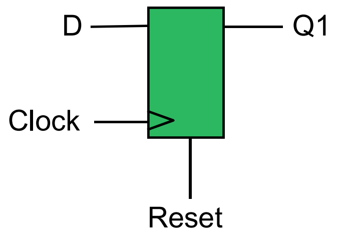
Simulation races can be avoided by using the RTL (or "non-blocking") assignment operator (<=) as shown above. (The ordinary assignment operator, =, is properly called the "blocking" assignment operator. The terms "blocking" and "non-blocking" will be explained later in the course.)

Unlike an ordinary or blocking assignment, a non-blocking or RTL assignment doesn't immediately update the reg on the left-hand side. Instead it schedules an event on the left hand side reg, which happens only at the end of the simulation time step. So in this example, b and c don't get their new values until after both the assignments have been executed.

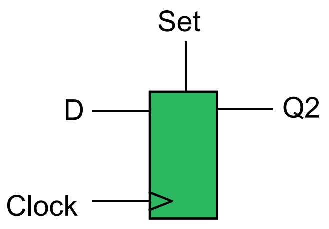
Asynchronous Set or Reset

Asynchronous Set or Reset

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Q1 <= 0;
  else
    Q1 <= D;
```



```
always @(posedge Clock or posedge Set)
  if (Set)
    Q2 <= 1;
  else
    Q2 <= D;
```

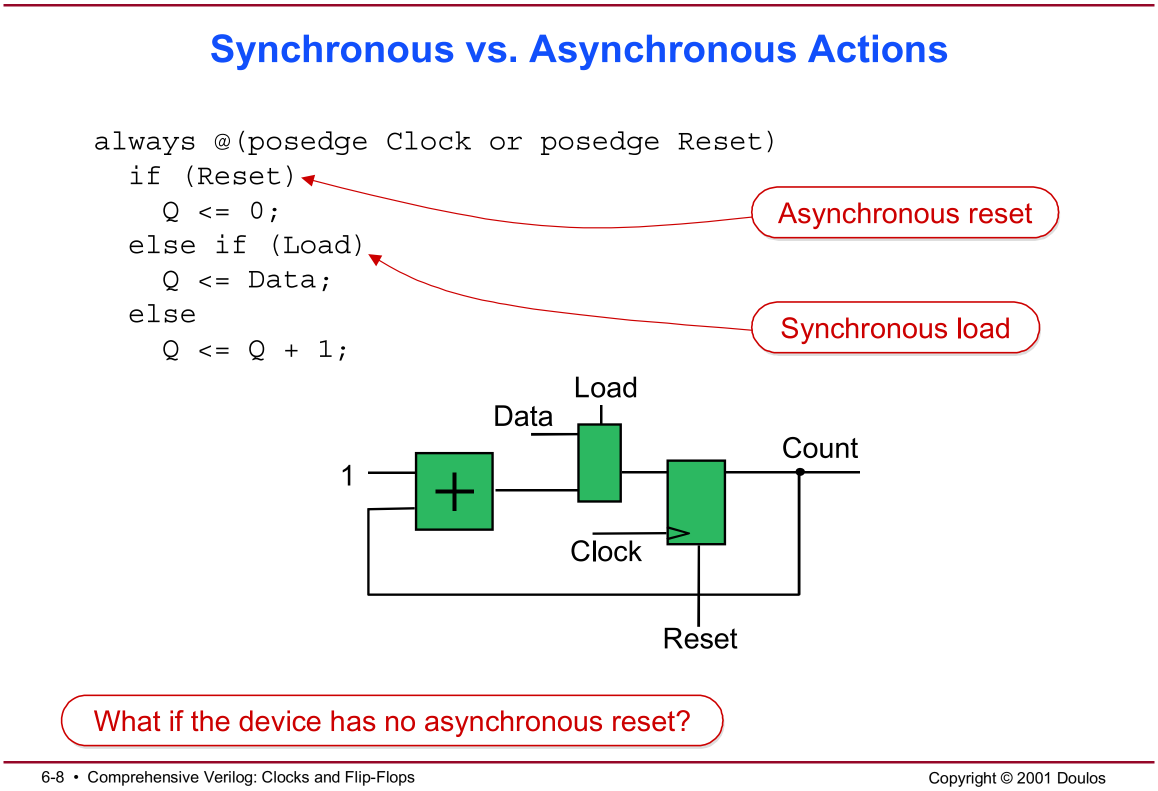


Notes:

Here are two examples showing how to describe flip-flops with asynchronous inputs.

Note that the clock and the asynchronous set or reset must appear in the event control, and both must be qualified by posedge or negedge as appropriate. This is true even though the set or reset is level-sensitive. If the asynchronous set or reset is not so qualified, the model would not simulate correctly.

Synchronous vs. Asynchronous Actions



Synchronous vs. Asynchronous Actions

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Q <= 0;
  else if (Load)
    Q <= Data;
  else
    Q <= Q + 1;
```

Asynchronous reset

Synchronous load

What if the device has no asynchronous reset?

6-8 • Comprehensive Verilog: Clocks and Flip-Flops

Copyright © 2001 Doulos

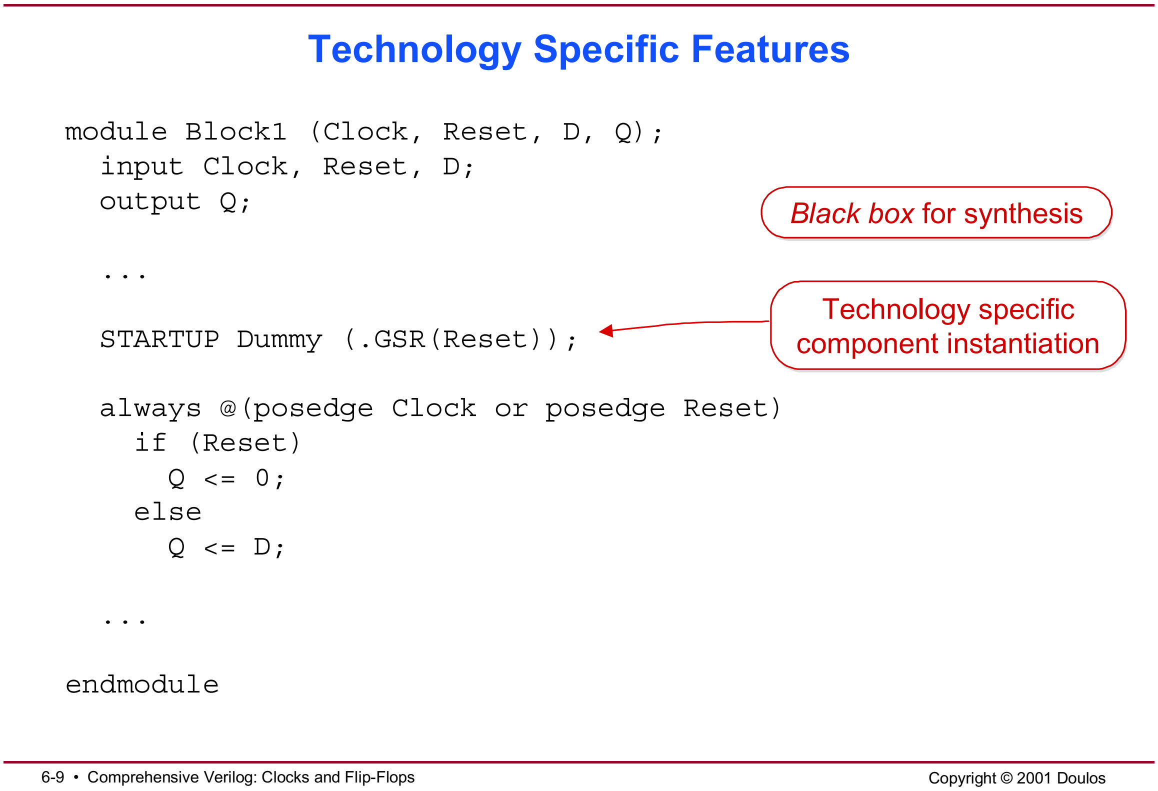
Notes:

This example shows some more complex synchronous logic with an asynchronous reset. Again, the active edge of the reset is included in the event control together with the clock edge.

Any asynchronous inputs must be tested before the synchronous behavior is described. The pattern is:

```
always @(active_clock_edge or active_async_edge)
  if ( async_is_active )
    perform_async_operation
  else
    perform_synchronous_operation
```

Technology Specific Features



Technology Specific Features

```
module Block1 (Clock, Reset, D, Q);
  input Clock, Reset, D;
  output Q;

  ...

  STARTUP Dummy (.GSR(Reset));

  always @(posedge Clock or posedge Reset)
    if (Reset)
      Q <= 0;
    else
      Q <= D;

  ...

endmodule
```

Black box for synthesis

Technology specific component instantiation

6-9 • Comprehensive Verilog: Clocks and Flip-Flops

Copyright © 2001 Doulos

Notes:

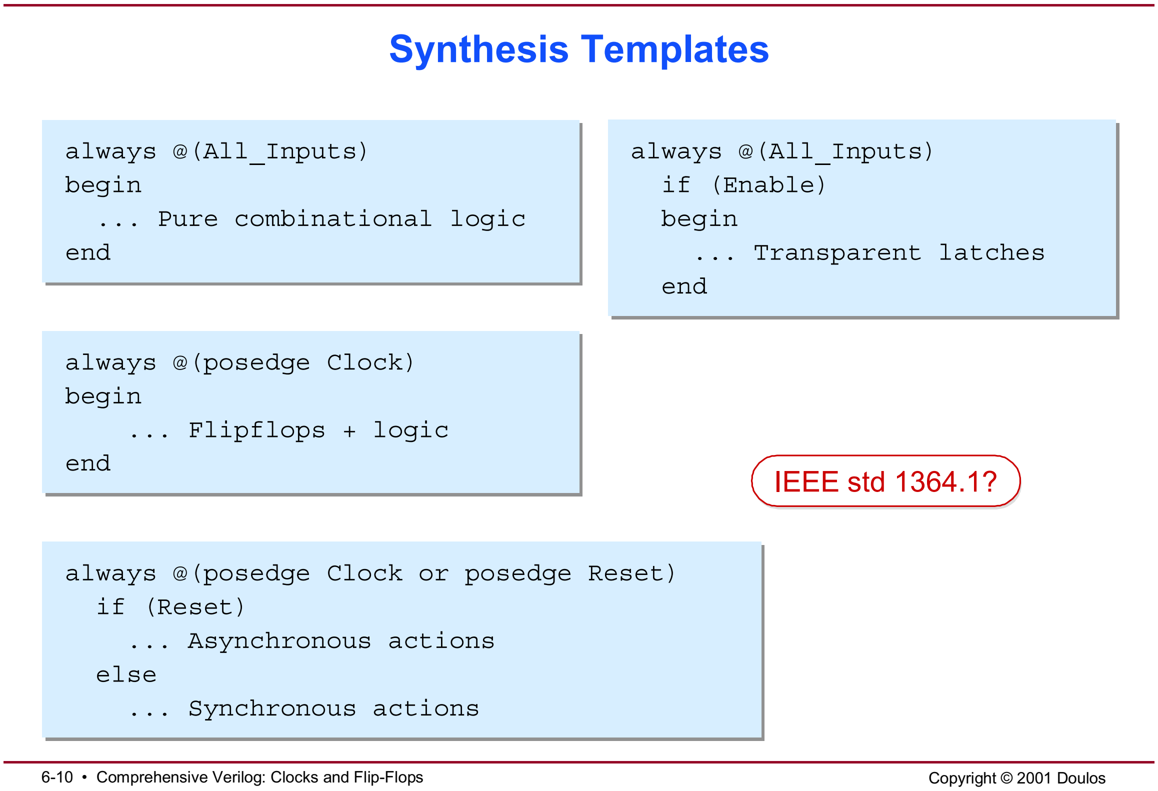
If you are designing an FPGA, you will need to bring architecture specific features of the device into your Verilog source code to achieve optimal results.

For example, Xilinx FPGA devices have a STARTUP block, which controls a dedicated global set/reset net (GSR). When asserted, the GSR sets or resets all the flip-flops in the device. The STARTUP block can be configured so that the GSR can be used in normal device operation. FPGA synthesis tools can usually infer the use of the GSR automatically. However, there are situations where automatic inference isn't the best option, and the global reset needs to be instantiated explicitly.

The Verilog code opposite shows an instance of the Xilinx STARTUP module, with the first (GSR) input connected to Reset. This plays no part in simulation, but tells the Xilinx place and route tools to use the global GSR net to route the reset.

Unfortunately, this means that the Verilog is no longer technology independent.

Synthesis Templates



Synthesis Templates

```
always @(All_Inputs)
begin
  ... Pure combinational logic
end
```

```
always @(All_Inputs)
  if (Enable)
  begin
    ... Transparent latches
  end
```

```
always @(posedge Clock)
begin
    ... Flipflops + logic
end
```

IEEE std 1364.1?

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    ... Asynchronous actions
  else
    ... Synchronous actions
```

6-10 • Comprehensive Verilog: Clocks and Flip-Flops

Copyright © 2001 Doulos

Notes:

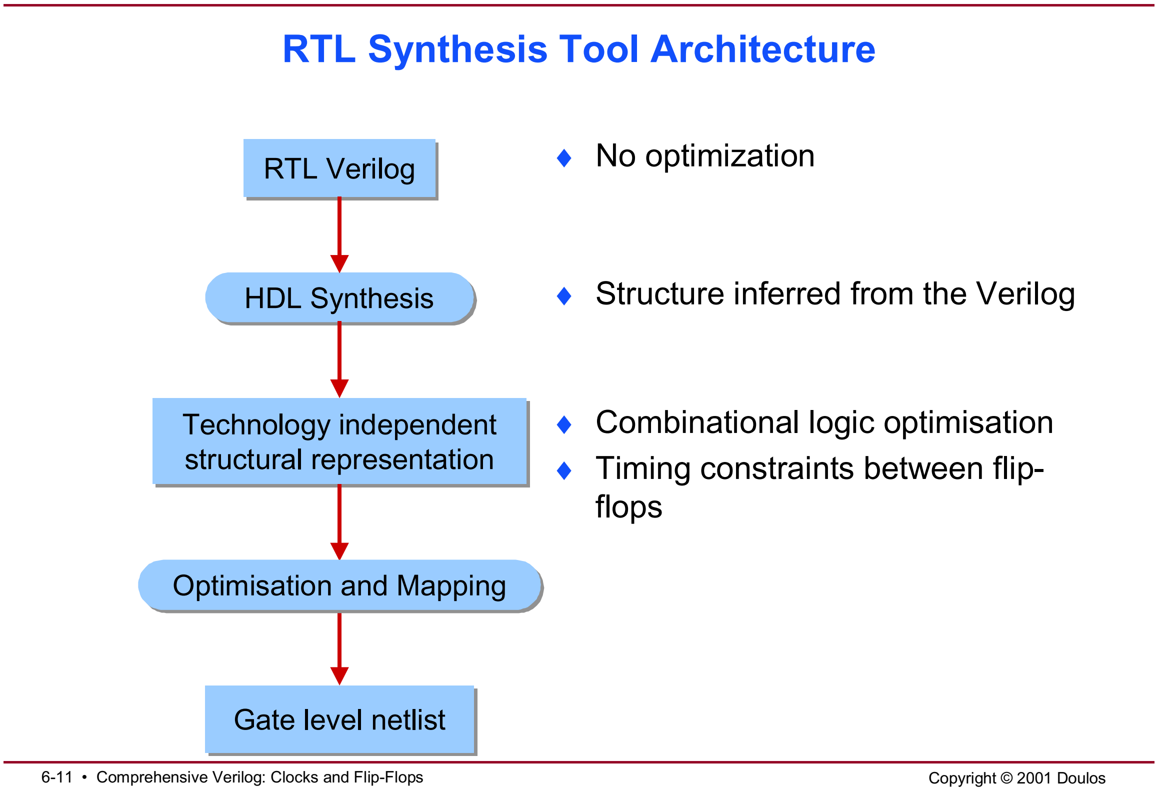
These templates show how to infer combinational logic, transparent latches and flip-flops from behavioral Verilog code.

When writing Verilog for synthesis, all always statements should follow one of these templates.

The proposed IEEE 1364.1 RTL synthesis standard defines what templates synthesis tools should support.

Note that for combinational logic, continuous assignments may also be used.

RTL Synthesis Tool Architecture



RTL Synthesis Tool Architecture

- No optimization
- Structure inferred from the Verilog
- Combinational logic optimisation
- Timing constraints between flip-flops

6-11 • Comprehensive Verilog: Clocks and Flip-Flops

Copyright © 2001 Doulos

Notes:

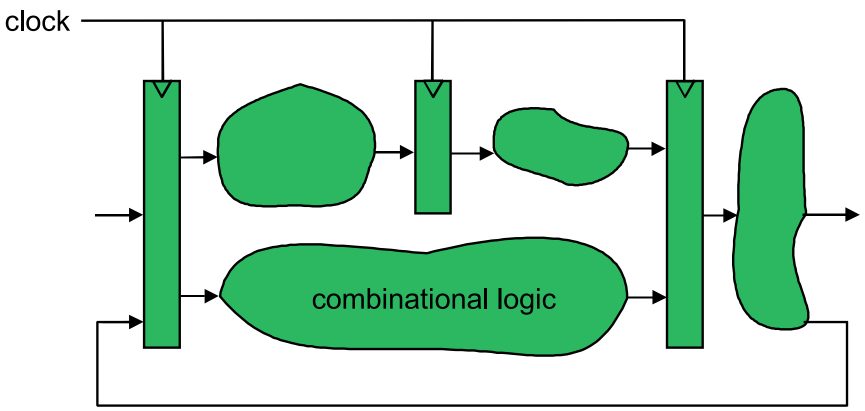
RTL synthesis tools typically work in two stages, shown in the diagram. The first stage, sometimes called HDL synthesis or translation, reads the Verilog HDL source code and converts it into a technology independent structural representation. This representation usually takes the form of a hierarchical network of technology independent boolean primitives such as NAND gates, D-type flip-flops, adders, multiplexers etc. The second stage converts this intermediate representation into an optimized network of primitives taken from the library of a specific cell based ASIC technology (the most popular to date being CMOS gate arrays and FPGAs).

The second stage performs both optimization and mapping. Optimization is often based on the technique of multi-level boolean minimization, where a multi-level network of boolean equations is restructured to minimize area (by sharing common boolean terms) and meet timing constraints (by reducing the number of logic levels on the critical path and swapping in faster, high-drive cells). A more specialized technique, state encoding, is used to optimize finite state machines by changing the encoding in the state register.

RTL Synthesis

RTL Synthesis

- Flip-flops are inferred by synchronizing assignments to clocks



- RTL synthesis optimizes combinational logic
- RTL synthesis does not add, delete or move flip-flops

6-12 • Comprehensive Verilog: Clocks and Flip-Flops

Notes:

A Register Transfer Level description (in Verilog HDL) must explicitly synchronize all operations with a clock - that is the working definition of RTL. The essence of RTL is that you define the registers and the transfers between those registers that occur on each clock tick (i.e. the combinational logic). RTL synthesis tools infer the existence of registers directly from the Verilog HDL source following a few simple rules.

RTL optimization

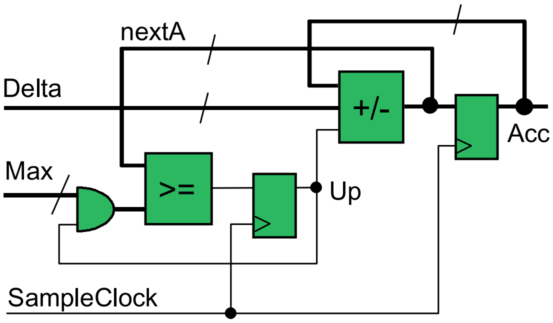
RTL synthesis tools keep the RTL structure defined in the Verilog HDL source code. In other words, RTL synthesis does not add registers, delete registers, merge registers, move registers, or optimize registers. RTL synthesis does optimize the combinational logic between the registers.

Some tools do perform more than the pure RTL synthesis described here. For example, many (but not all) tools merge equivalent flip-flops, as we shall see later.

Inferring Flip-Flops

Inferring Flip-Flops

```
reg Up;
reg [11:0] Acc, nextA;

always @(posedge SampleClock)
begin
  if (Up)
  begin
    nextA = Acc + Delta;
    if (nextA >= Max)
      Up <= 0;
  end
  else
  begin
    nextA = Acc - Delta;
    if (nextA[11] == 1'b1) // nextA < 0
      Up <= 1;
  end
  Acc <= nextA;
end
```

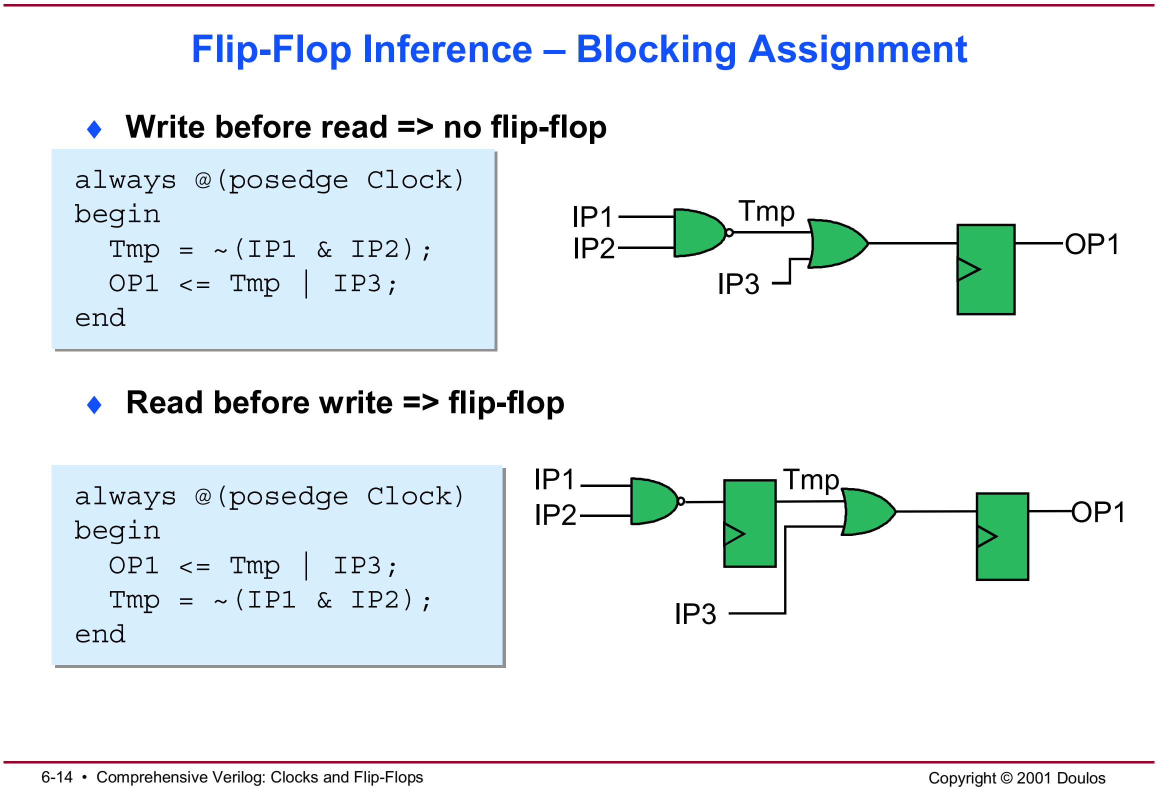


Notes:

regs assigned in a procedural block that is synchronized to a clock are candidates to become flip-flops. Whether a reg is synthesized as a flip-flop or a wire depends on how it is used. *regs* that are assigned a new value before being read in each clock cycle will become wires (nextA in the example opposite). *Regs* that are read before being assigned in any clock cycle become flip-flops (Up in the example opposite). Finally, *regs* that are assigned with non-blocking assignments become flip-flops unconditionally.

These rules are explained more fully in the following pages.

Flip-Flop Inference – Blocking Assignment



Flip-Flop Inference – Blocking Assignment

- Write before read => no flip-flop

```
always @(posedge Clock)
begin
  Tmp = ~(IP1 & IP2);
  OP1 <= Tmp | IP3;
end
```

- Read before write => flip-flop

```
always @(posedge Clock)
begin
  OP1 <= Tmp | IP3;
  Tmp = ~(IP1 & IP2);
end
```

Notes:

Whether or not a flip-flop is inferred from a blocking assignment depends on whether or not the value of the reg being assigned needs to be remembered from one clock edge to the next.

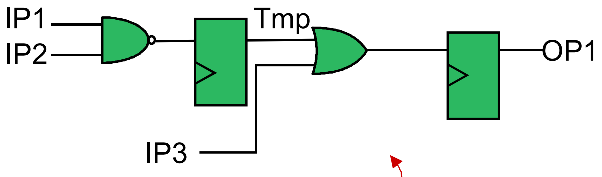
- If, on an active clock edge, the reg is assigned a value before its value is used, then no flip-flop is required.
- If the value of the reg is used before a new value is assigned to it, then the value that is used will be the value that was assigned on a previous clock. Therefore a flip-flop is required.

Flip-Flop Inference – Non-Blocking

Flip-Flop Inference – Non-Blocking

- **Non-blocking assignments always infer flip-flops**

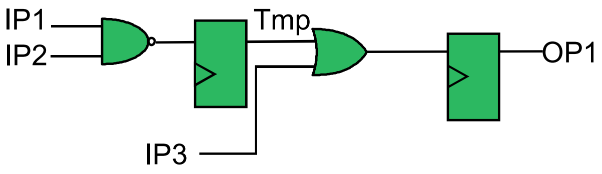
```
always @(posedge Clock)
begin
  Tmp <= ~(IP1 & IP2);
  OP1 <= Tmp | IP3;
end
```



- **Order of assignments doesn't matter**

Same!

```
always @(posedge Clock)
begin
  OP1 <= Tmp | IP3;
  Tmp <= ~(IP1 & IP2);
end
```



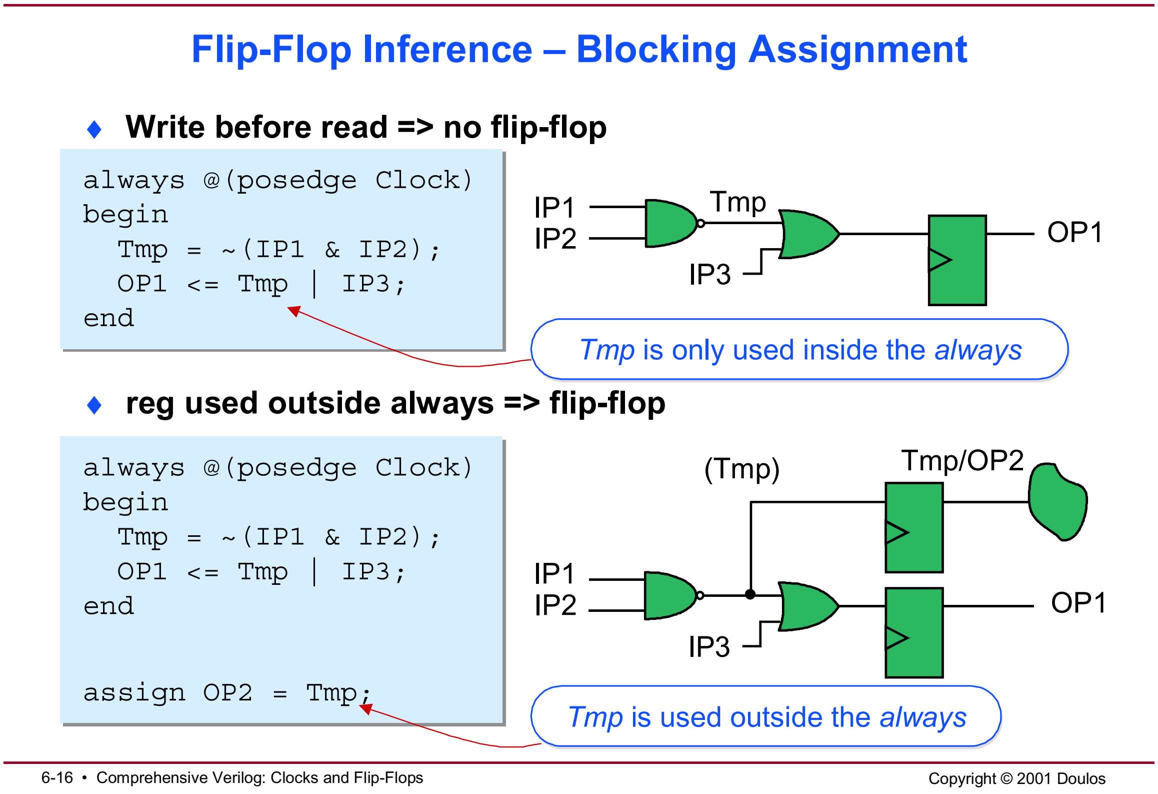
Notes:

The rule for flip-flop inference from non-blocking assignments is straightforward: a flip-flop is always inferred!

Because a flip-flop is always inferred when a non-blocking assignment is used, it is generally recommended that a non-blocking assignments are used whenever flip-flops are required.

Note that when non-blocking assignments are used, the order of the assignments does not matter. This is because regs assigned with a non-blocking assignment are not updated immediately.

Flip-Flop Inference – Blocking Assignment



Flip-Flop Inference – Blocking Assignment

- Write before read => no flip-flop

```
always @(posedge Clock)
begin
  Tmp = ~(IP1 & IP2);
  OP1 <= Tmp | IP3;
end
```

Tmp is only used inside the *always*

- reg used outside always => flip-flop

```
always @(posedge Clock)
begin
  Tmp = ~(IP1 & IP2);
  OP1 <= Tmp | IP3;
end


assign OP2 = Tmp;
```

Tmp is used outside the *always*

6-16 • Comprehensive Verilog: Clocks and Flip-Flops

Notes:

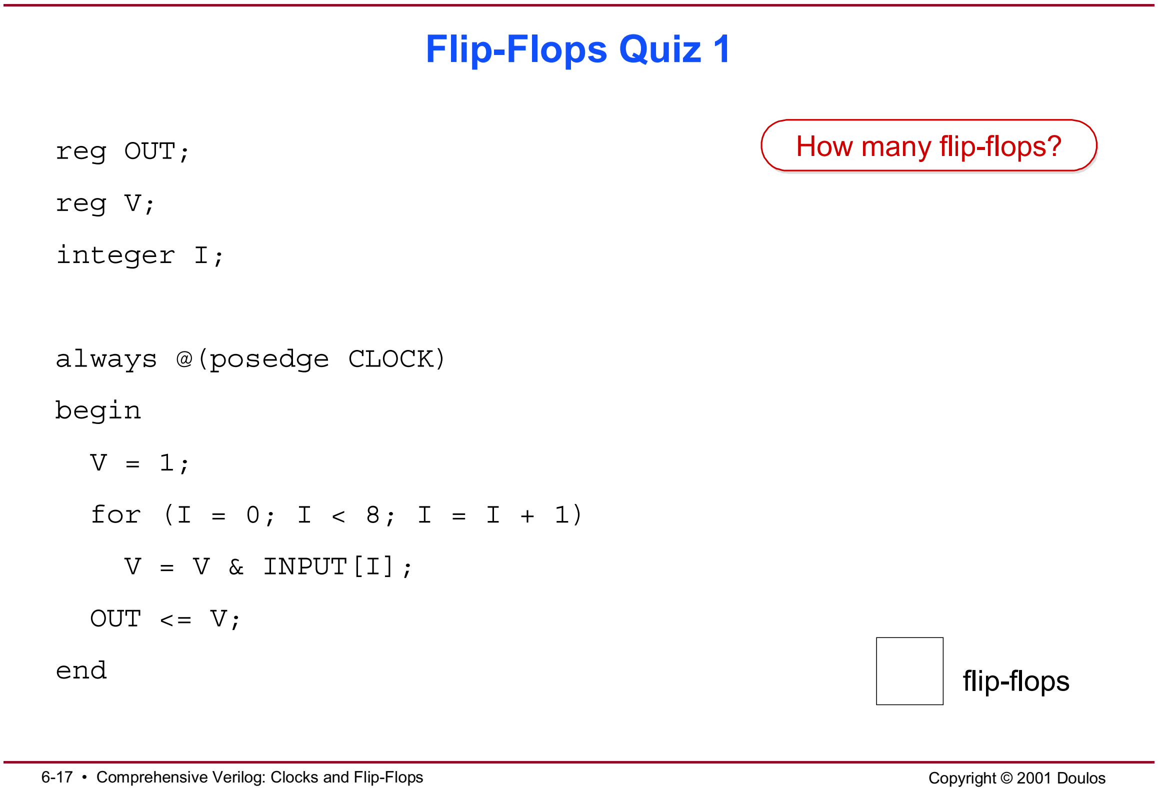
If a reg is assigned values with blocking assignments, and the reg is used outside the always block in which it is assigned, then a flip-flop will be inferred, even if the reg is not required to store a value from one clock cycle to the next.

In the example at the bottom of the slide, Tmp is used in the continuous assignment to OP2, and so a flip-flop is inferred.

However, it is not good practice to infer flip-flops like this, because of the possibility of simulation races, as described earlier in this section. Rewriting the bottom example as follows is to be preferred, because (i) it is clear that a flip-flop will be inferred, and (ii) there will not be a simulation race:

```
always @(posedge Clock)
begin
  Tmp <= ~(IP1 & IP2);
  OP1 <= Tmp | IP3;
end

assign OP2 = Tmp;
```

Flip-Flops Quiz 1



Flip-Flops Quiz 1

How many flip-flops?

```
reg OUT;
reg V;
integer I;

always @(posedge CLOCK)
begin
  V = 1;
  for (I = 0; I < 8; I = I + 1)
    V = V & INPUT[I];
  OUT <= V;
end
```

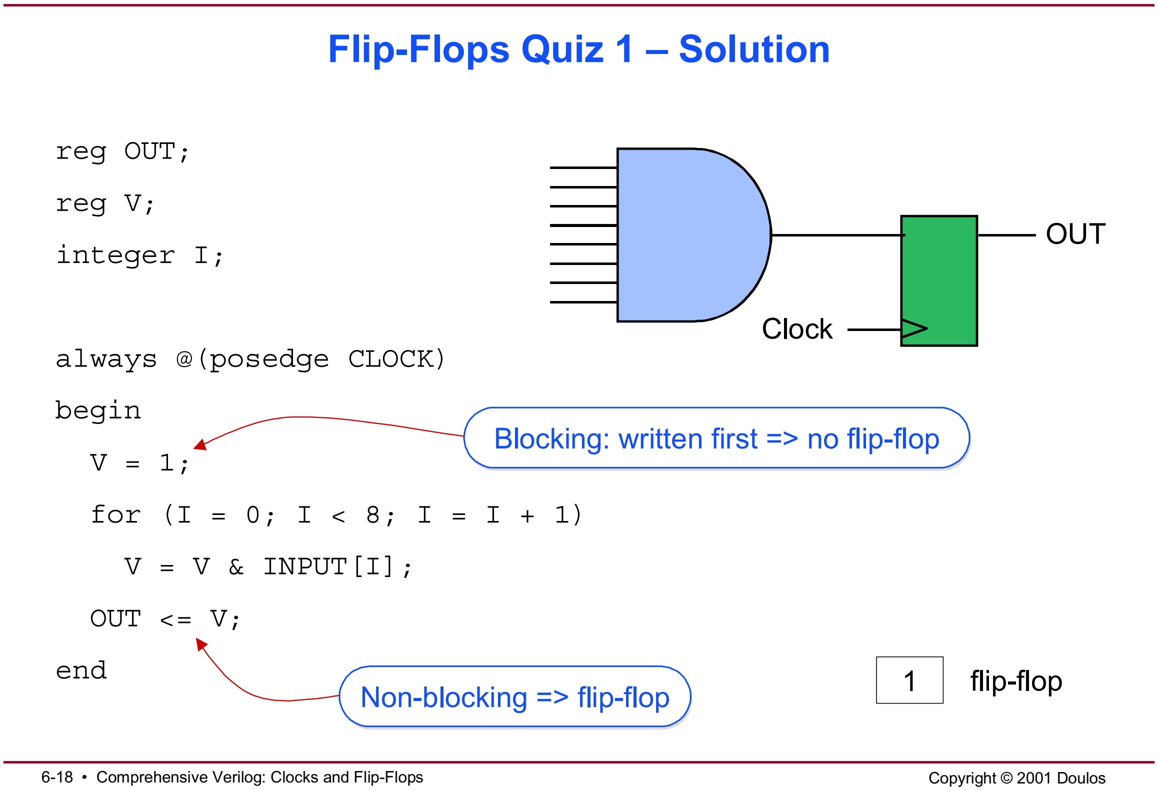
□ flip-flops

6-17 • Comprehensive Verilog: Clocks and Flip-Flops

Notes:

How many flip-flops will be inferred from this always block?

Flip-Flops Quiz 1 – Solution



Flip-Flops Quiz 1 – Solution

```
reg OUT;
reg V;
integer I;

always @(posedge CLOCK)
begin
  V = 1;
  for (I = 0; I < 8; I = I + 1)
    V = V & INPUT[I];
  OUT <= V;
end
```

Blocking: written first => no flip-flop

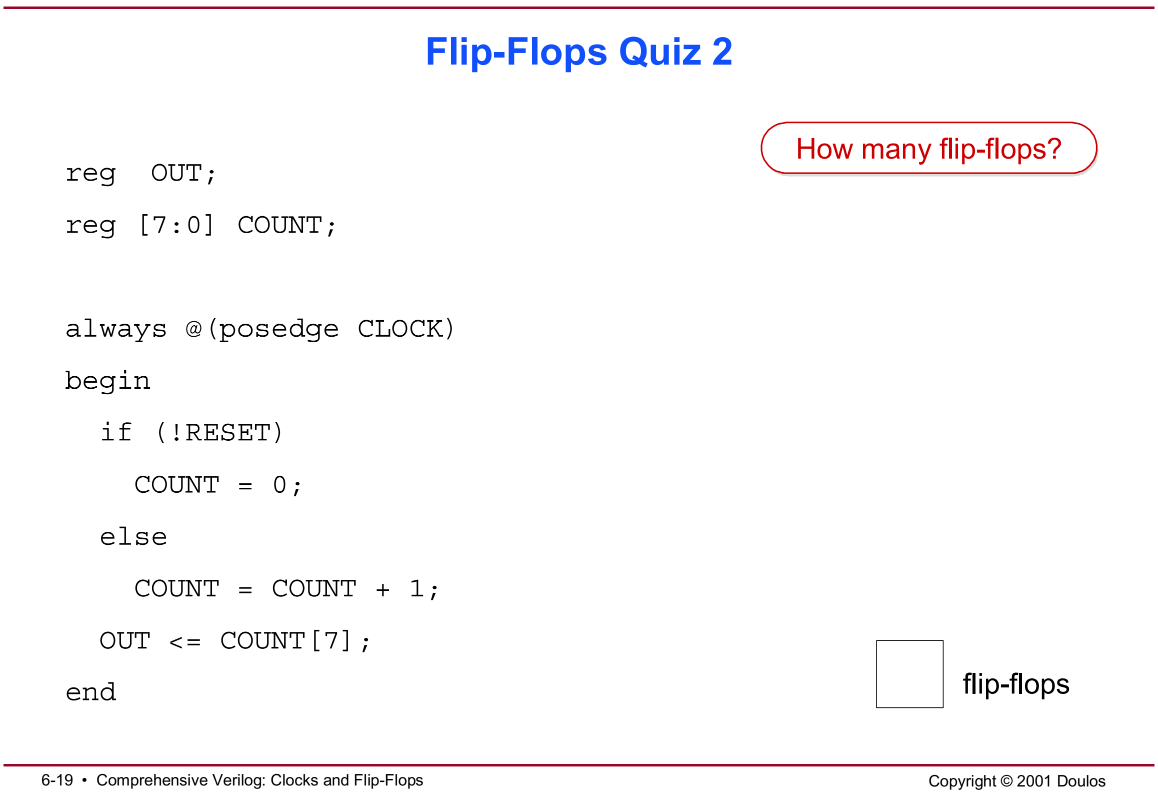
Non-blocking => flip-flop

1	flip-flop

6-18 • Comprehensive Verilog: Clocks and Flip-Flops

Notes:

Flip-Flops Quiz 2



Flip-Flops Quiz 2

How many flip-flops?

```
reg  OUT;
reg [7:0] COUNT;

always @(posedge CLOCK)
begin
  if (!RESET)
    COUNT = 0;
  else
    COUNT = COUNT + 1;
  OUT <= COUNT[7];
end
```

□ flip-flops

6-19 • Comprehensive Verilog: Clocks and Flip-Flops

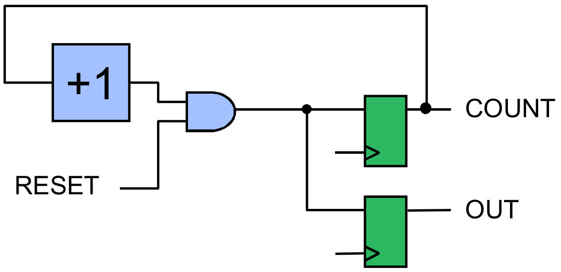
Notes:

How many flip-flops will be inferred from this always block?

Flip-Flops Quiz 2 – Solution

Flip-Flops Quiz 2 – Solution

```
reg  OUT;
reg [7:0] COUNT;

always @(posedge CLOCK)
begin
  if (!RESET)
    COUNT = 0;
  else
    COUNT = COUNT + 1;
  OUT <= COUNT[7];
end
```



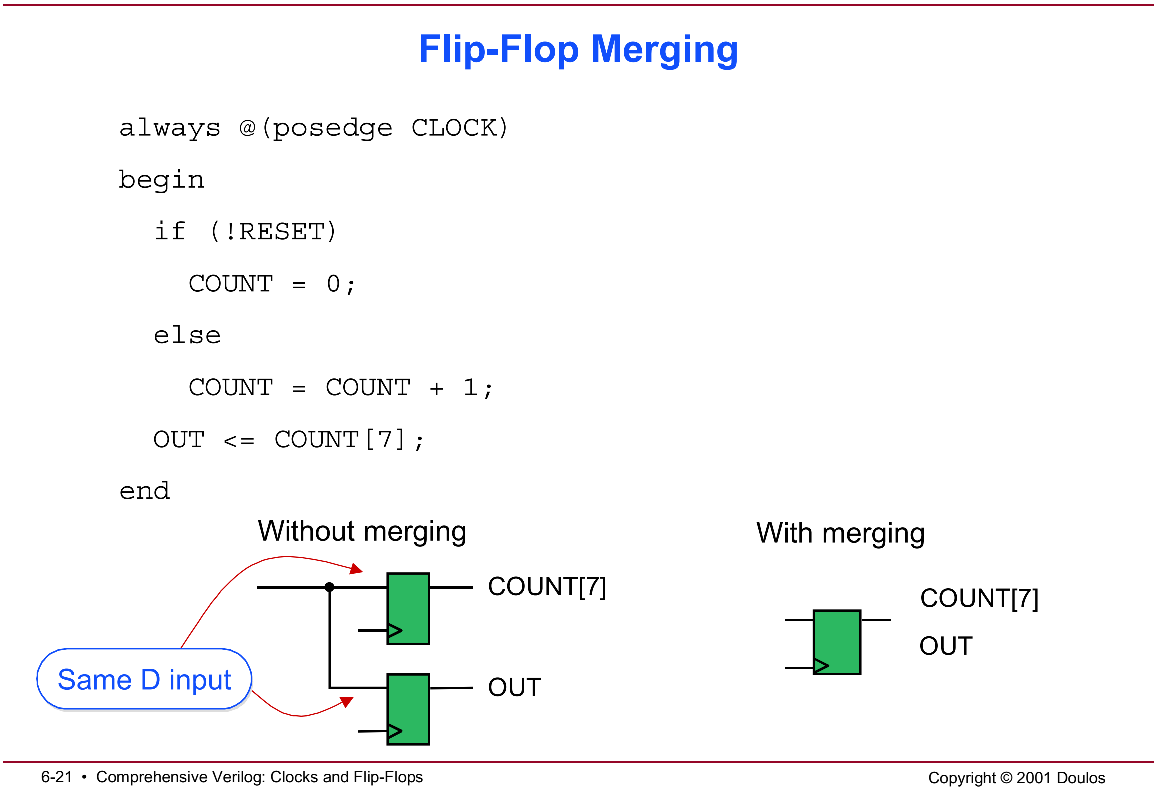
Blocking: read first => 8 flip-flops

Non-blocking => flip-flop

□9 flip-flops

Notes:

Flip-Flop Merging



Flip-Flop Merging

```
always @(posedge CLOCK)
begin
  if (!RESET)
    COUNT = 0;
  else
    COUNT = COUNT + 1;
  OUT <= COUNT[7];
end
```

6-21 • Comprehensive Verilog: Clocks and Flip-Flops

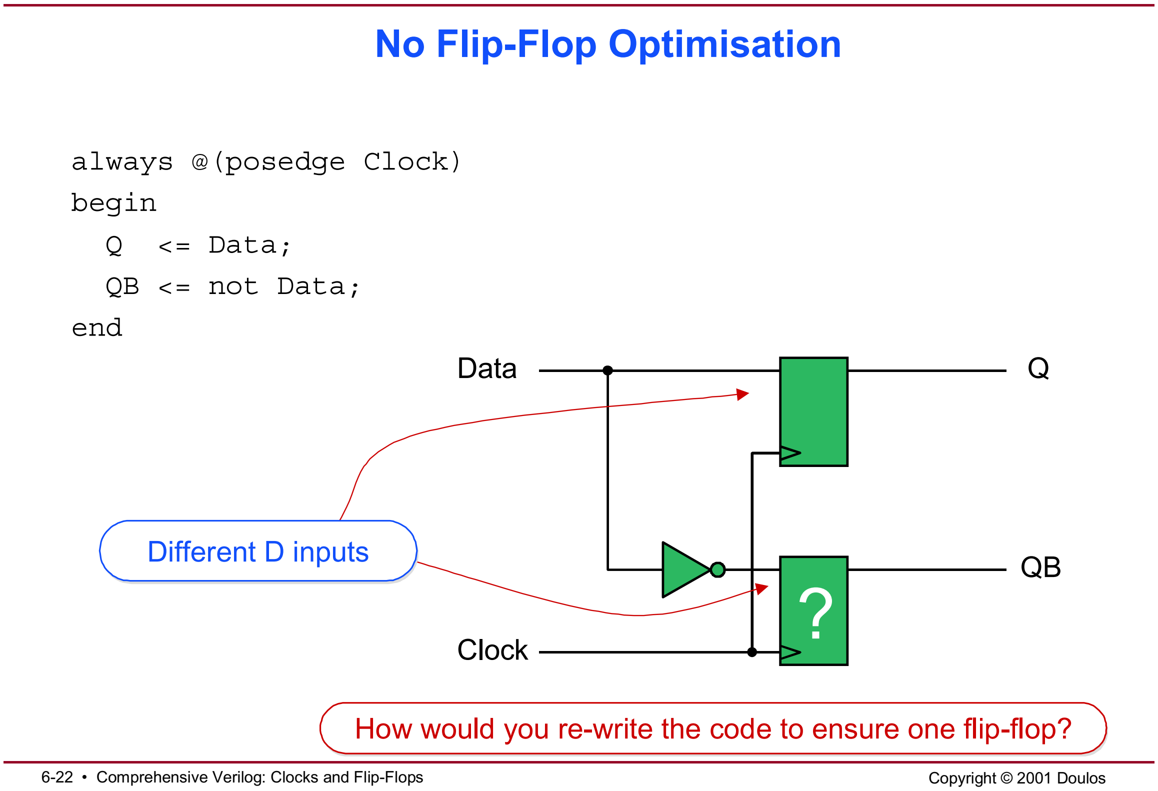
Notes:

Following the rules for synthesizing flip-flops, we will get 8 flip-flops for the reg COUNT, and one flip-flop for the reg OUT.

Note, however, that COUNT[7] and OUT always have the same value, so their respective flip-flops have their D inputs connected together.

Many synthesis tools will allow two flip-flops to have their D inputs tied, and will actually give 9 flip-flops in this situation. Other synthesis tools will automatically merge flip-flops with their D inputs connected, giving just 8 flip-flops.

No Flip-Flop Optimization



No Flip-Flop Optimisation

```
always @(posedge Clock)
begin
  Q  <= Data;
  QB <= not Data;
end
```

Different D inputs

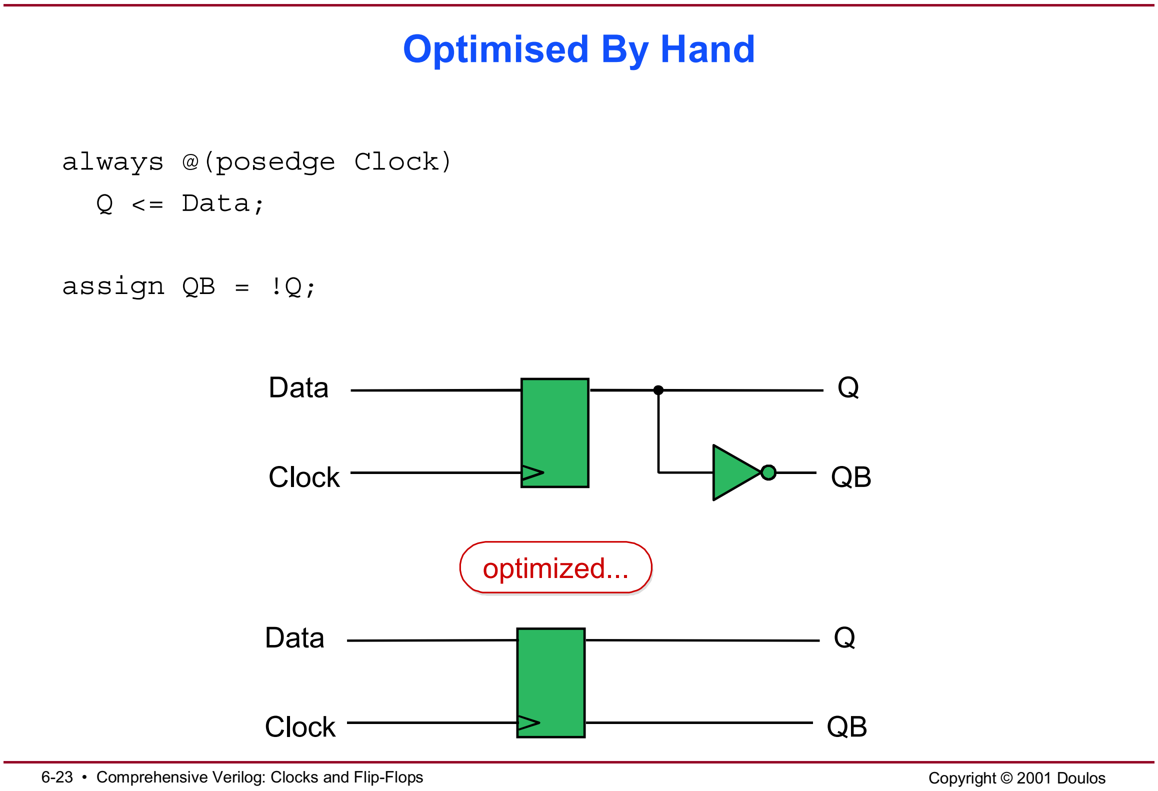
How would you re-write the code to ensure one flip-flop?

6-22 • Comprehensive Verilog: Clocks and Flip-Flops

Notes:

In the example above, both Q and QB will become flip-flops! Care must be taken when writing RTL Verilog in order to avoid unwanted flip-flops. Synthesis is very simple minded when it comes to flip-flops. You are responsible for deciding how many flip-flops are needed and how they are organized, then writing the Verilog accordingly.

Optimized By Hand



Optimised By Hand

```
always @(posedge Clock)
  Q <= Data;

assign QB = !Q;
```

6-23 • Comprehensive Verilog: Clocks and Flip-Flops

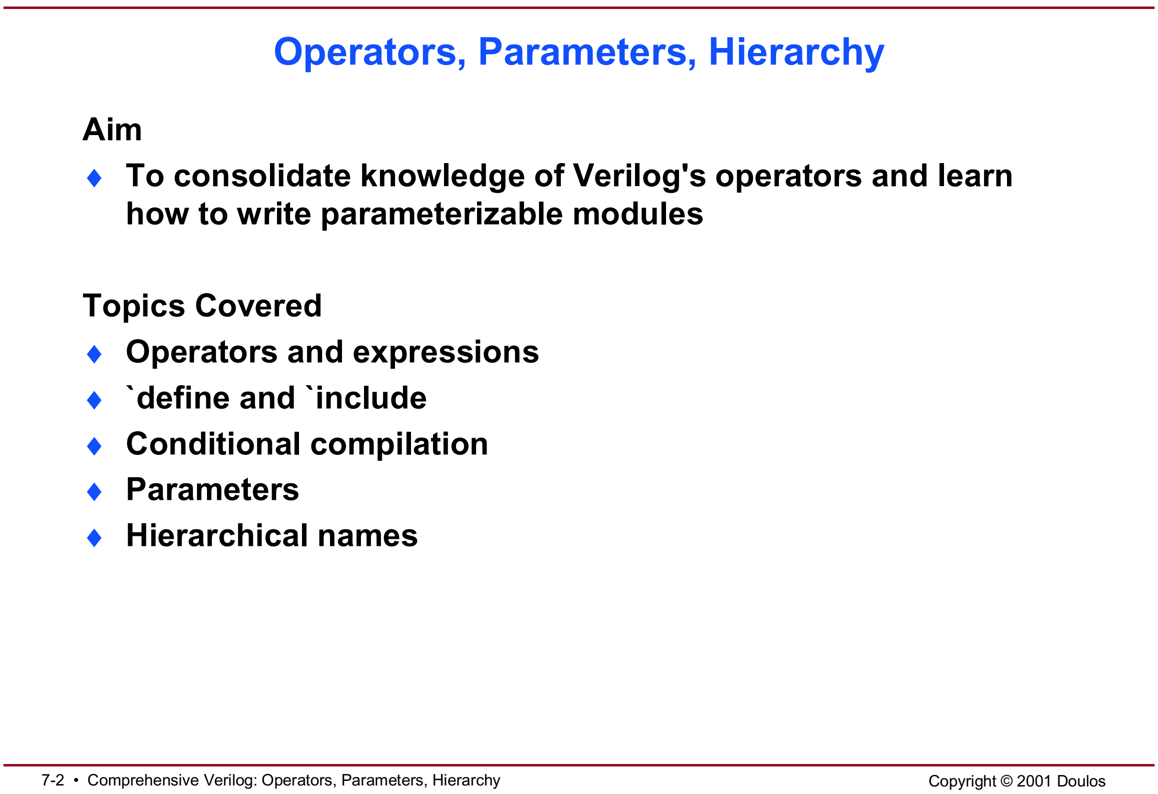
Notes:

To avoid unwanted flip-flops in this situation, only one of Q and QB can be assigned within the clocked always block. The solution is to take the assignment to QB outside the always block and make it a continuous assignment, thus avoiding a flip-flop for QB.

If flip-flops with Q and QB outputs are available in the target technology, the synthesis tool should be able to use these. Note that the flip-flops in most CPLD and FPGA devices only have a Q output, so this optimization will not be made.

Module 7
Operators, Parameters, Hierarchy

Operators, Parameters, and Hierarchy



Operators, Parameters, Hierarchy

Aim

- To consolidate knowledge of Verilog's operators and learn how to write parameterizable modules

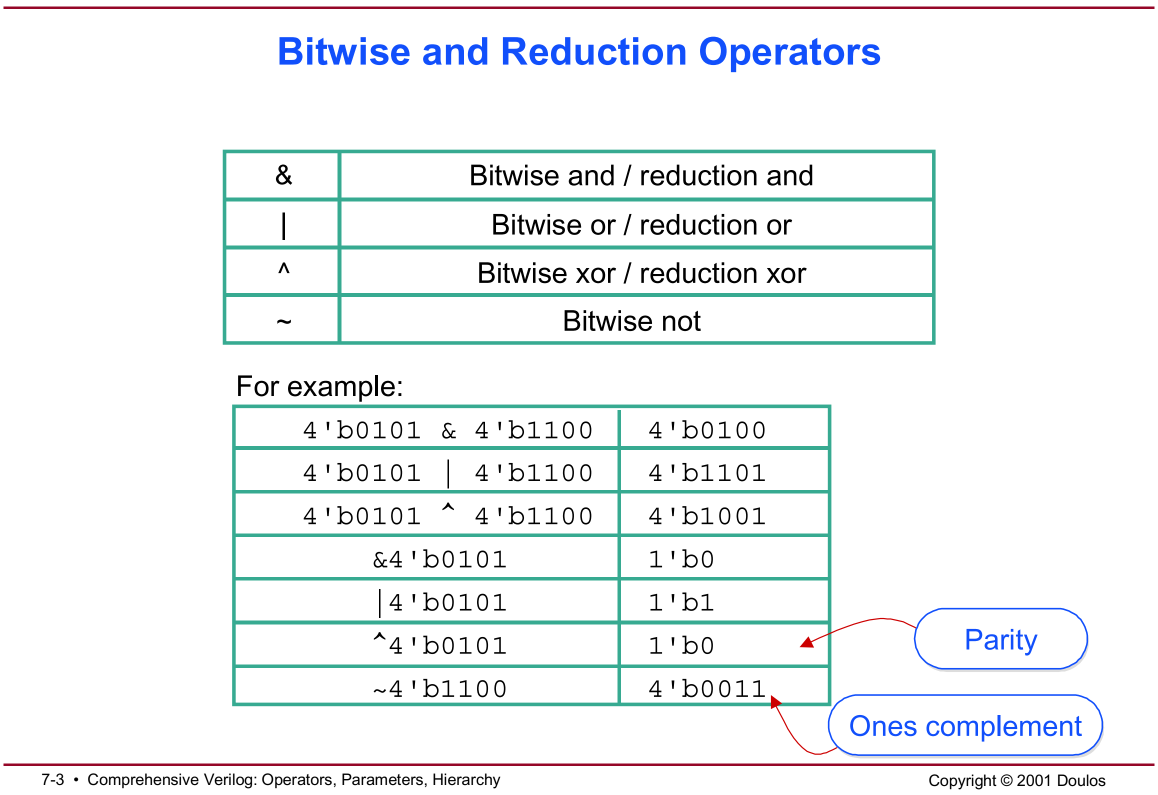
Topics Covered

- Operators and expressions
- `define and `include
- Conditional compilation
- Parameters
- Hierarchical names

7-2 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Notes:

Bitwise and Reduction Operators



Bitwise and Reduction Operators

&	Bitwise and / reduction and
|	Bitwise or / reduction or
^	Bitwise xor / reduction xor
~	Bitwise not

For example:

`4'b0101 & 4'b1100`	`4'b0100`
`4'b0101 | 4'b1100`	`4'b1101`
`4'b0101 ^ 4'b1100`	`4'b1001`
`&4'b0101`	`1'b0`
`|4'b0101`	`1'b1`
`^4'b0101`	`1'b0`
`~4'b1100`	`4'b0011`

7-3 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Notes:

We have already met some of the bitwise operators, but only with scalar operands. For vectors, a bitwise operation treats the individual bits of vector operands separately.

For example, a bitwise and (&) of two vectors produces a vector result, with the each bit of the result being a logical and of the corresponding bits of the two operands.

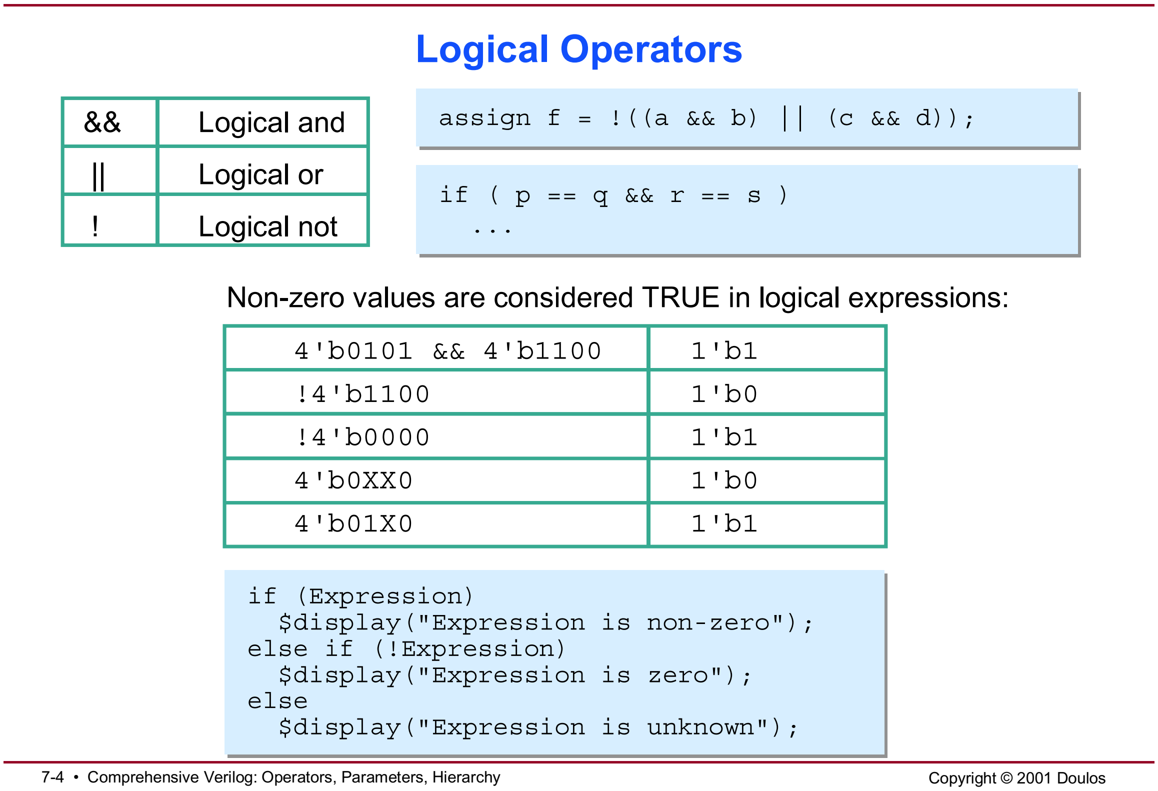
Likewise, a bitwise inversion (~) of a vector simply inverts all the bits of that vector. In other words it produces the one's complement.

Reduction operators

Confusingly, &, | and ^ can also be used as unary reduction operators. A reduction operator takes a single operand that is a vector and produces a one bit result. For example, the reduction and (&) ands together the bits of the vector. The result is always 1'b0, unless all the bits are '1'.

The only really useful reduction operator is ^, which generates a parity bit.

Logical Operators



Logical Operators

&&	Logical and
||	Logical or
!	Logical not

```
assign f = !((a && b) || (c && d));
```

```
if ( p == q && r == s )
  ...
```

Non-zero values are considered TRUE in logical expressions:

`4'b0101 && 4'b1100`	`1'b1`
`!4'b1100`	`1'b0`
`!4'b0000`	`1'b1`
`4'b0XX0`	`1'b0`
`4'b01X0`	`1'b1`

```
if (Expression)
  $display("Expression is non-zero");
else if (!Expression)
  $display("Expression is zero");
else
  $display("Expression is unknown");
```

Notes:

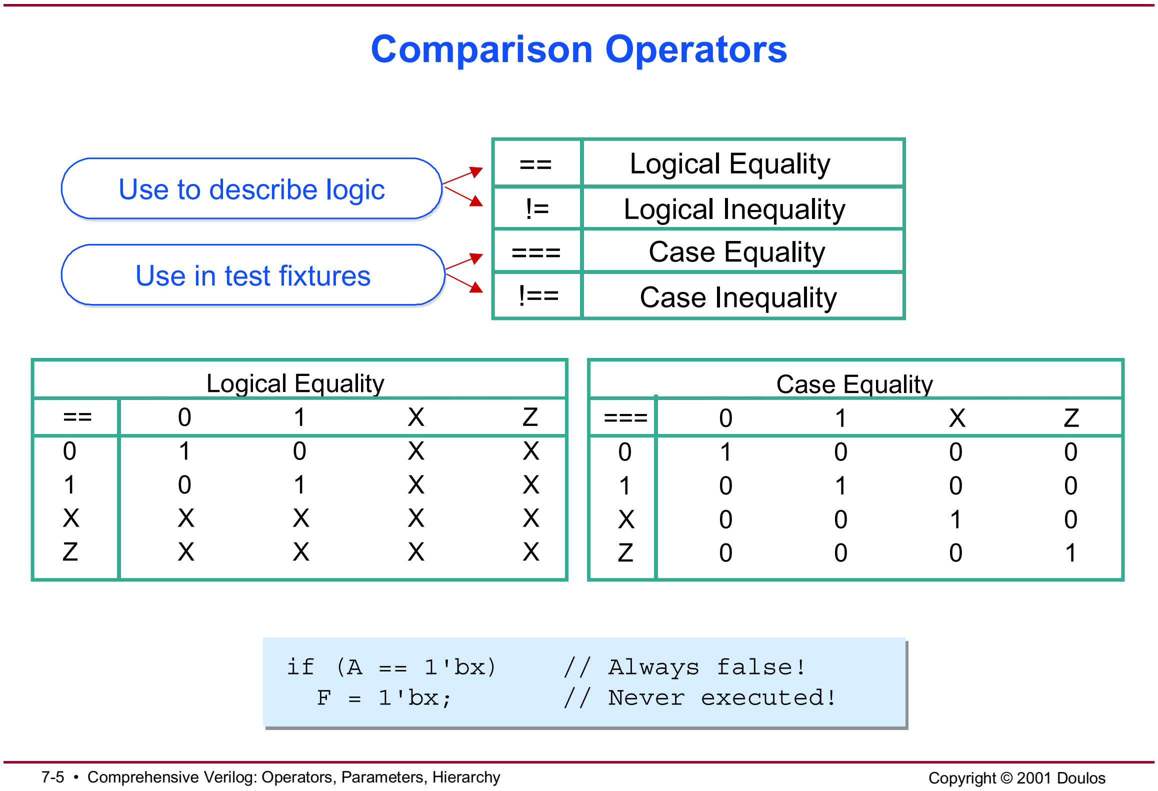
In contrast to the bitwise operators, the logical operators treat their operands as logical (Boolean) quantities. A scalar or vector is considered to be TRUE when it is not zero, and FALSE when it is zero. Xs and Zs are considered to be unknown (neither TRUE nor FALSE).

An expression that uses logical operators evaluates to 1'b1 (TRUE), 1'b0 (FALSE) or 1'bX (unknown). When an unknown value is tested, in an if statement for example, it is considered to be FALSE.

The not operator (!) is useful for testing to see whether or not an expression is zero.

Note that for single-bit quantities, &&, || and ! are equivalent to &, | and ~ respectively.

Comparison Operators



Comparison Operators

Use to describe logic

Use in test fixtures

==	Logical Equality
!=	Logical Inequality
===	Case Equality
!==	Case Inequality

Logical Equality				
==	0	1	X	Z
0	1	0	X	X
1	0	1	X	X
X	X	X	X	X
Z	X	X	X	X

Case Equality				
===	0	1	X	Z
0	1	0	0	0
1	0	1	0	0
X	0	0	1	0
Z	0	0	0	1

```
if (A == 1'bx)      // Always false!
  F = 1'bx;         // Never executed!
```

7-5 • Comprehensive Verilog: Operators, Parameters, Hierarchy

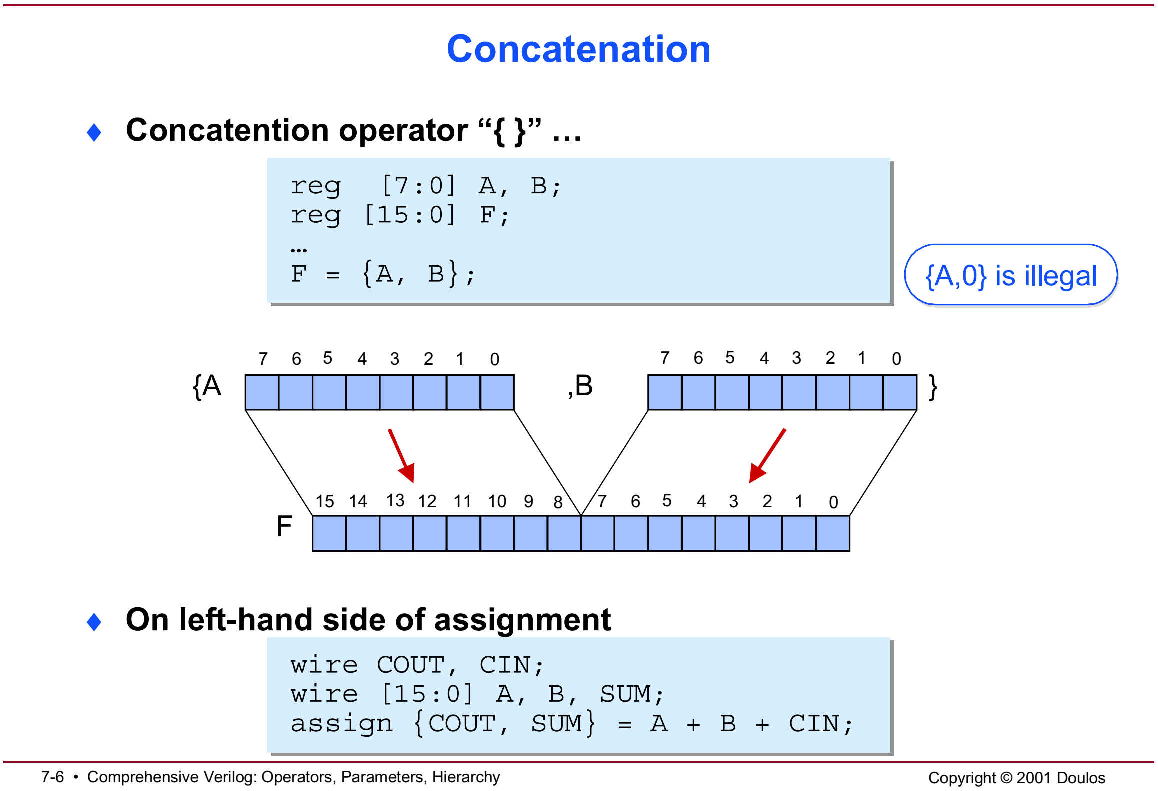
Notes:

There are two sorts of equality operator in Verilog. == and != are logical equality operators, which compare the values of two vectors. The result of such a comparison is TRUE (if the values are the same), FALSE (if they are not) or unknown (if either vector contains an X or Z).

=== and !== are the case equality operators. Two vectors are equal if and only if their bits are identical, including Xs and Zs.

== gives the same result as ===, and != as !== if the vectors being compared do not contain Xs or Zs. Otherwise == and != may give unknown results, whereas === and !== give a definite result.

Concatenation



Concatenation

- Concatention operator "{ }" …

```
reg  [7:0] A, B;
reg [15:0] F;
…
F = {A, B};
```

{A,0} is illegal

- On left-hand side of assignment

```
wire COUT, CIN;
wire [15:0] A, B, SUM;
assign {COUT, SUM} = A + B + CIN;
```

7-6 • Comprehensive Verilog: Operators, Parameters, Hierarchy

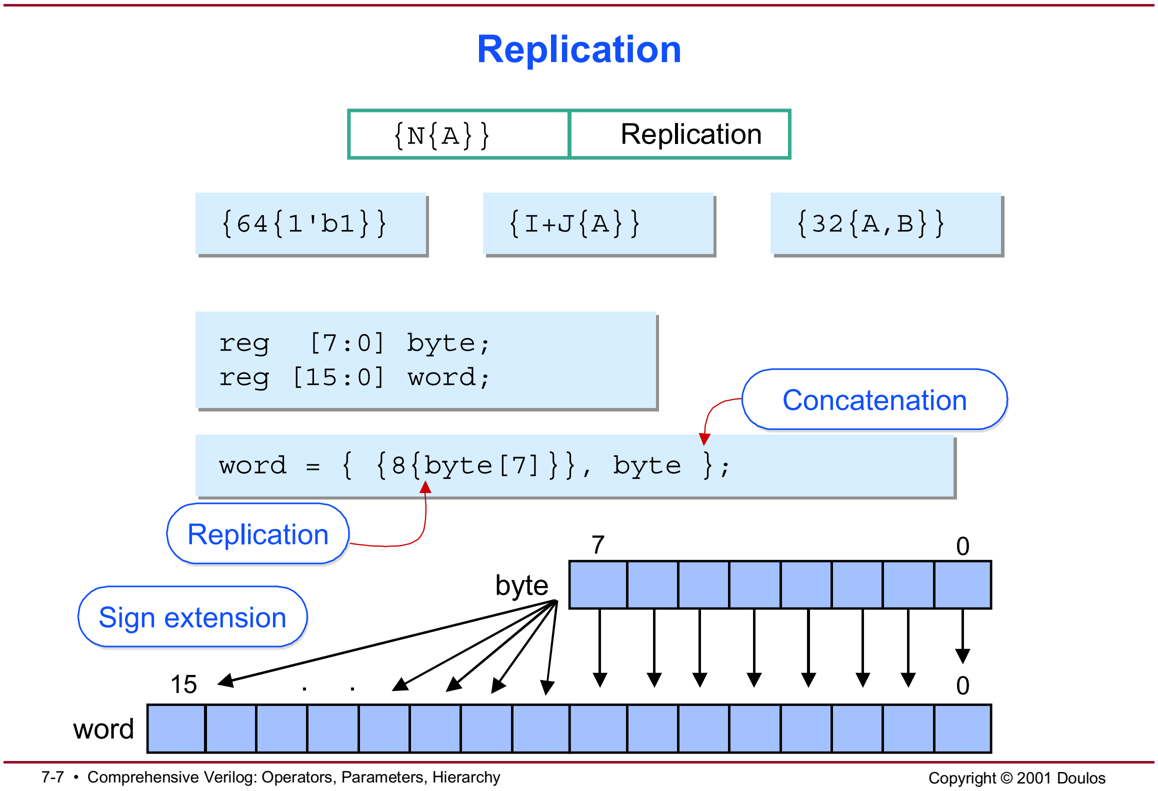
Notes:

The concatenation operator {} is used to join together two vectors (or single bit values) to form a single longer vector. The two concatenated vectors are placed end-to-end, as shown opposite in the assignment F = {A, B};, where A is copied into the leftmost 8 bits of F, and B is copied into the rightmost 8 bits of F.

Note that it is an error to include an unsized integer value in a concatenation.

The second example shows concatenation being used on the left-hand side of a continuous assignment. The wire COUT will receive the carry bit generated from the sum in the expression on the right-hand side of the assignment. This is because the width of the addition (A + B + CIN) is taken from the width of the left-hand side of the assignment ({COUT, SUM}).

Replication



Replication

`{N{A}}`	Replication

`{64{1'b1}}`   `{I+J{A}}`   `{32{A,B}}`

```
reg  [7:0] byte;
reg [15:0] word;
```

```
word = { {8{byte[7]}}, byte };
```

7-7 • Comprehensive Verilog: Operators, Parameters, Hierarchy

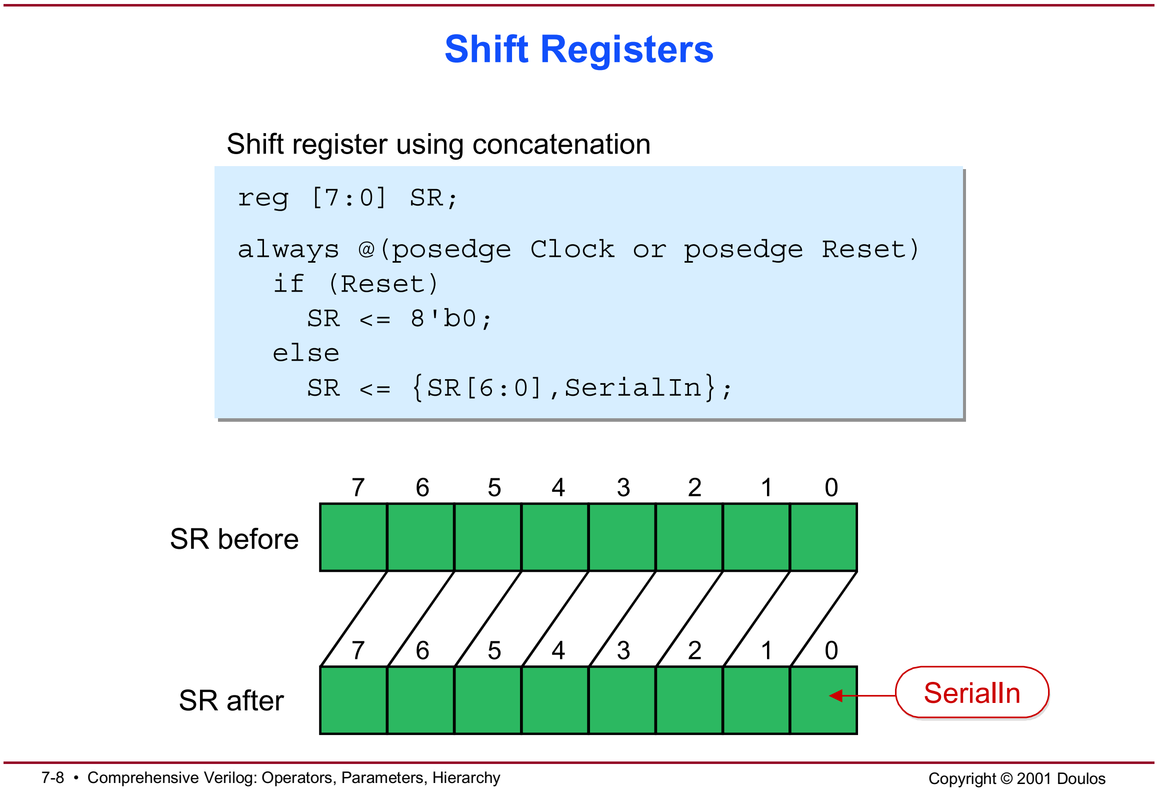
Notes:

The replication operator is an extension of the concatenation operator. It also uses curly brackets - two sets, in fact. The inner concatenation is repeated by the number of times given after the first curly bracket.

Note that the innermost curly brackets do enclose a concatenation: expressions such as {10{A,B}} are legal.

The example in the bottom half of the slide shows how to describe the sign extension of an eight bit reg to sixteen bits. Sign extension ensures that the two's complement value of the smaller reg is maintained. Explicit sign extension is often required in Verilog models that use two's complement (signed) values. Note that the expression being assigned to word is a concatenation of a replication ({8{byte[7]}} with a simple expression (byte).

Shift Registers



Shift Registers

Shift register using concatenation

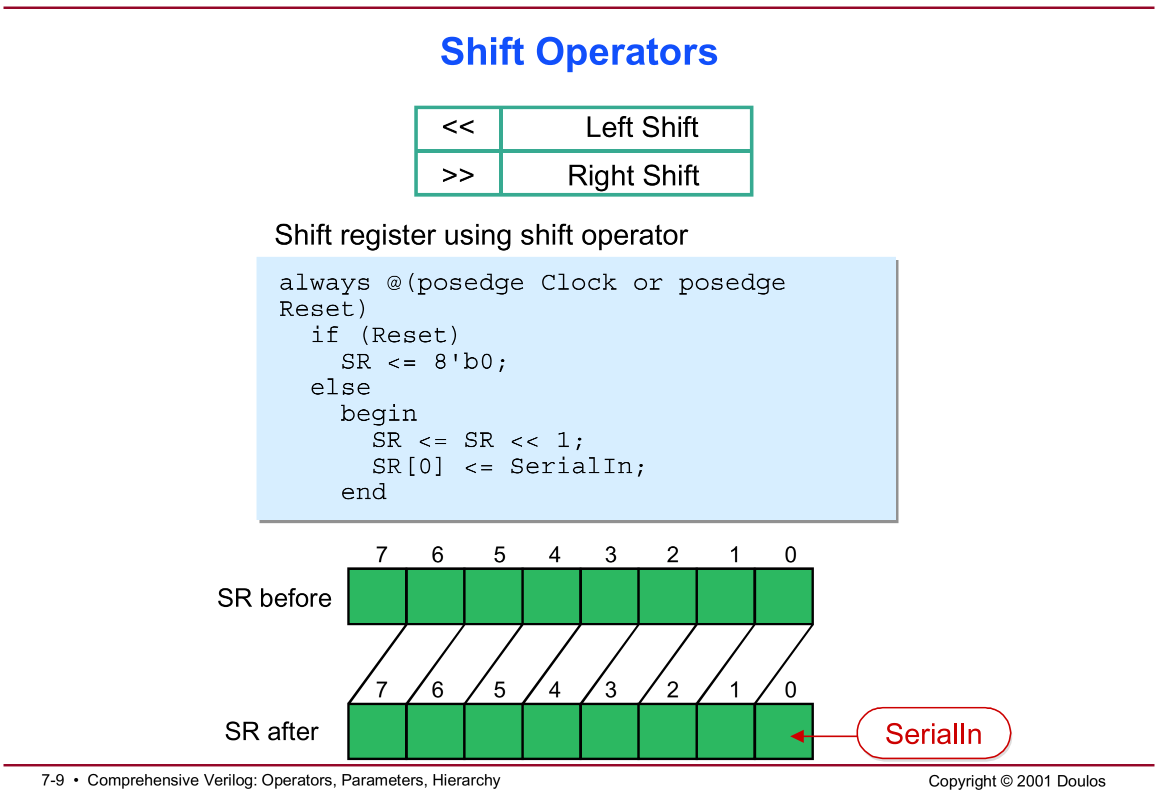
```
reg [7:0] SR;

always @(posedge Clock or posedge Reset)
  if (Reset)
    SR <= 8'b0;
  else
    SR <= {SR[6:0],SerialIn};
```

7
6
5
4
3
2
1
0
SR before
7
6
5
4
3
2
1
0
SR after
SerialIn

Notes:

Shift registers can be described using concatenation, as shown. The right-hand side of the assignment is an eight-bit vector, with SR[6] as the MSB and SerialIn as the LSB. This is the new value for the left-hand side of the assignment (SR). Thus SR[6] is copied to SR[7], SR[5] to SR[6] etc. and SerialIn to SR[0].

Shift Operators



Shift Operators

<<	Left Shift
>>	Right Shift

Shift register using shift operator

```
always @(posedge Clock or posedge
Reset)
  if (Reset)
    SR <= 8'b0;
  else
    begin
      SR <= SR << 1;
      SR[0] <= SerialIn;
    end
```

7-9 • Comprehensive Verilog: Operators, Parameters, Hierarchy

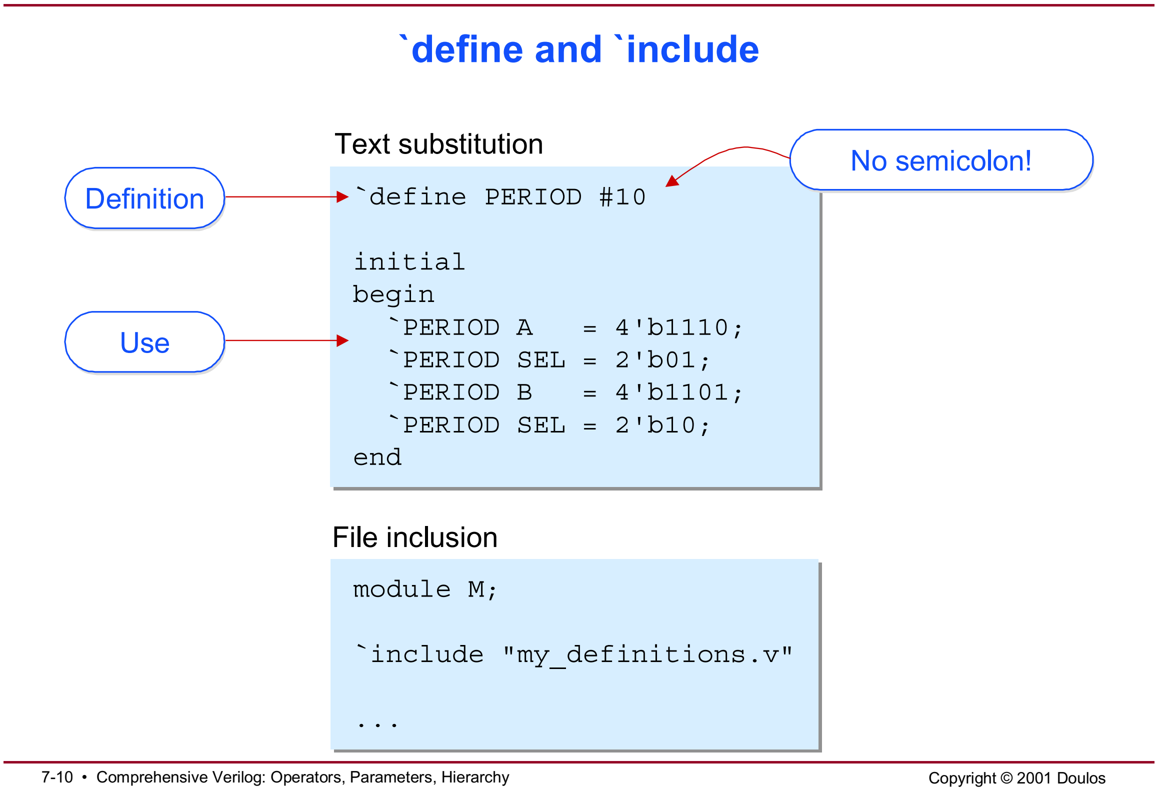
Copyright © 2001 Doulos

Notes:

Verilog has two shift operators. << shifts to the left and >> to the right. The left operand is shifted by the number of bits in the right operand. Any vacated bits are filled with zeroes.

Note that for this example, the shift can be described in one statement using the concatenation operator, but the shift operator requires two statements to be used.

‘define and ‘include



`define and `include

Text substitution

Definition

Use

No semicolon!

```
`define PERIOD #10

initial
begin
   `PERIOD A    = 4'b1110;
   `PERIOD SEL = 2'b01;
   `PERIOD B    = 4'b1101;
   `PERIOD SEL = 2'b10;
end
```

File inclusion

```
module M;

`include "my_definitions.v"

...
```

7-10 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Copyright © 2001 Doulos

Notes:

Compiler directives are not, strictly speaking, part of the Verilog language, but are instructions to the Verilog compiler. A compiler directive is preceded by the backwards apostrophe ("back-tick"), or grave accent character (`). Each compiler directive starts with the character ` and is on a line by itself. Be careful to distinguish this from the normal apostrophe or inverted comma ('), which is used in numbers such as 8'bx

A compiler directive applies throughout a Verilog source file (and in some tools, even across source file boundaries) until overruled by another compiler directive.

`define

`define declares a text macro. A text macro is simply a string of characters. When the macro name, which also must be preceded by the apostrophe (`), appears in the Verilog source code, the compiler substitutes the corresponding macro string.

Note that there is no semicolon (;) at the end of the macro string. If one were included it would be considered part of the string, and substituted accordingly!

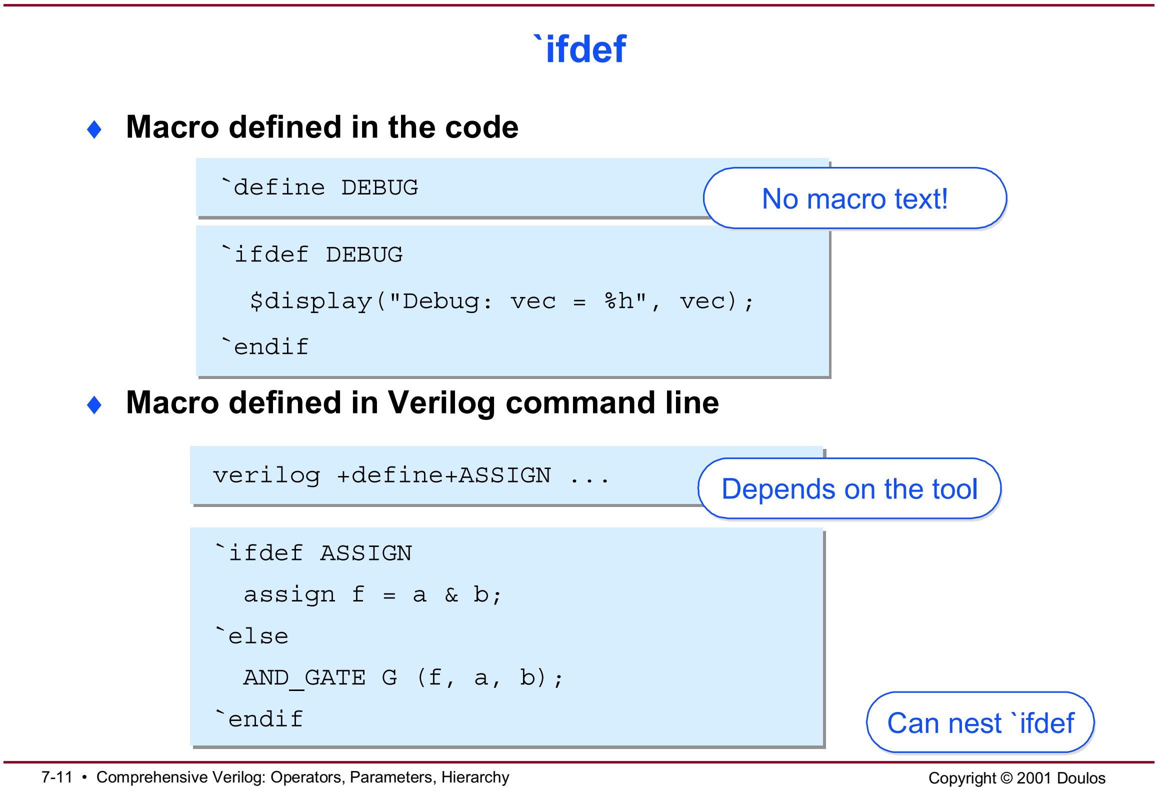
When used appropriately, `define can be used to make the Verilog code easier to read and maintain. It is also used in conjunction with the `ifdef directive, which is described on the next slide.

`include

`include is followed by a file name in double quotes. The compiler simply substitutes the contents of the file in place of the `include statement. It is as if the contents of the file had been inserted using a text editor.

One common use of `include is to enable commonly used definitions to be made once, and then included and used as required.

‘ifdef



`ifdef

- Macro defined in the code

```
`define DEBUG
```

No macro text!

```
`ifdef DEBUG
  $display("Debug: vec = %h", vec);
`endif
```

- Macro defined in Verilog command line

```
verilog +define+ASSIGN ...
```

Depends on the tool

```
`ifdef ASSIGN
  assign f = a & b;
`else
  AND_GATE G (f, a, b);
`endif
```

Can nest `ifdef

7-11 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Copyright © 2001 Doulos

Notes:

`ifdef

`ifdef enables conditional compilation, where certain lines of Verilog source code are only compiled if certain conditions are met.

One application of `ifdef is to enable diagnostic code to be included or excluded from a particular simulation.

`ifdef is followed by the name of a text macro. (Note that when used like this as part of an `ifdef statement, the back-tick is not required before the macro name.) If the macro has been defined, the following statements up to the next `endif directive are compiled, irrespective of the actual value of the macro.

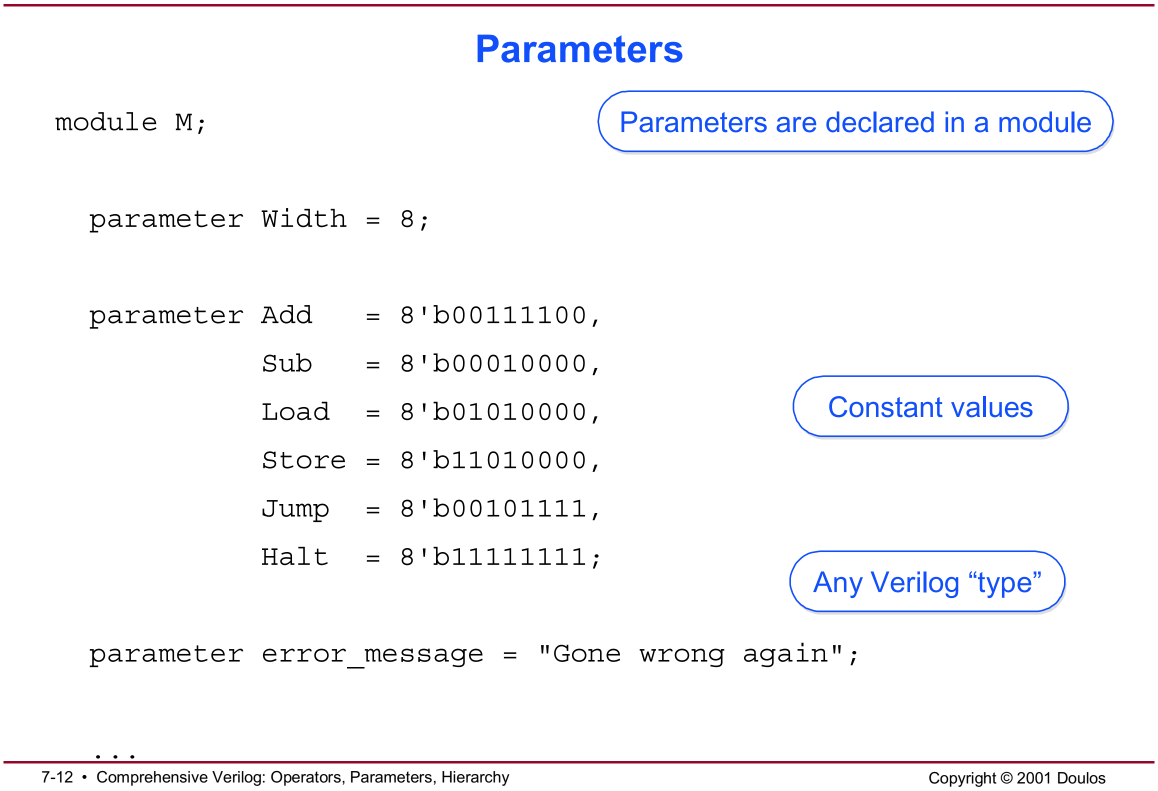
`else

When an `else directive is included, the source lines between `ifdef and `else are compiled if the macro is defined, otherwise the lines between `else and `endif are compiled.

`ifdef may be nested.

A `define macro can be defined and (optionally) given a value on the Verilog command line as shown. The details depend on which Verilog simulator you are using. The example opposite is for Cadence's Verilog-XL simulator.

Parameters



Parameters

```
module M;

  parameter Width = 8;

  parameter Add   = 8'b00111100,
            Sub   = 8'b00010000,
            Load  = 8'b01010000,
            Store = 8'b11010000,
            Jump  = 8'b00101111,
            Halt  = 8'b11111111;

  parameter error_message = "Gone wrong again";

  ...
```

Parameters are declared in a module
Constant values
Any Verilog “type”

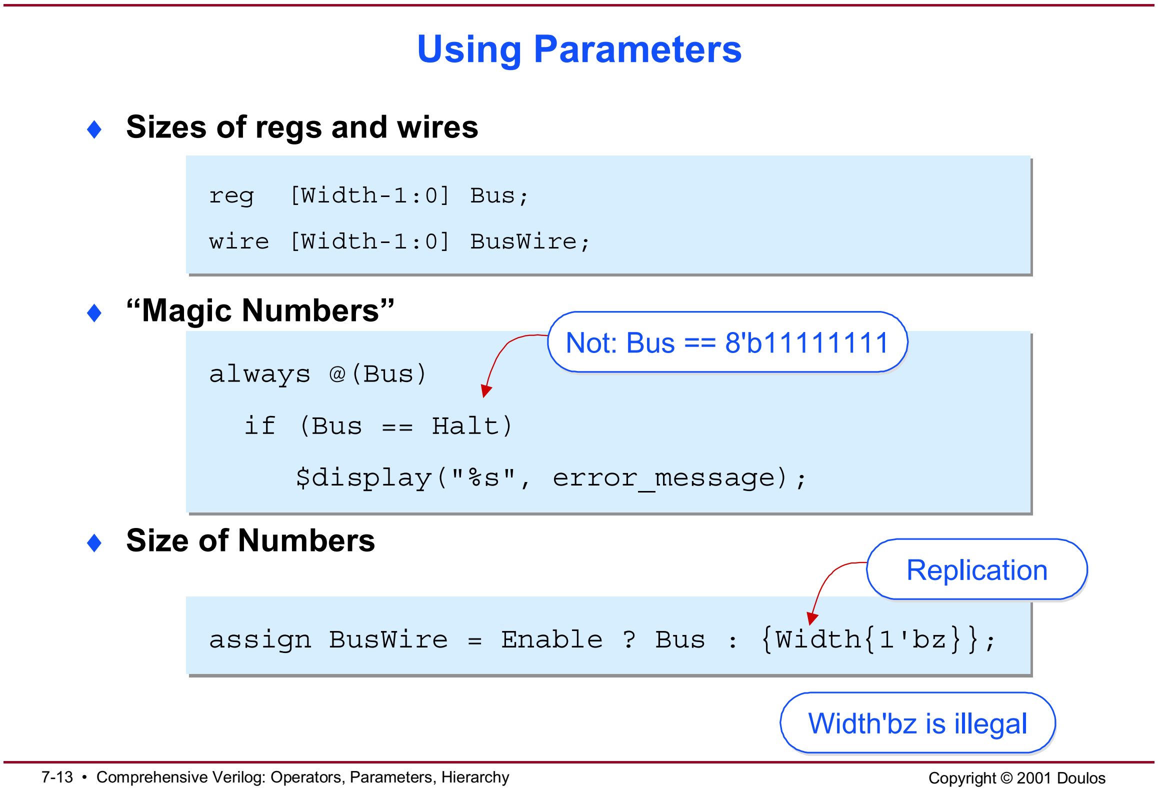
7-12 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Notes:

Parameters are constants and are declared in modules. A parameter maintains its value throughout simulation. Parameters can be used in declarations and expressions in the place of literal constants.

Parameters and `define macros are often interchangeable. Parameters look neater and are local to a module. `define macros can be global. Parameter values can be overwritten as described later in this section.

Using Parameters



Using Parameters

- **Sizes of regs and wires**

```
reg  [Width-1:0] Bus;
wire [Width-1:0] BusWire;
```

- **“Magic Numbers”**

```
always @(Bus)
  if (Bus == Halt)
    $display("%s", error_message);
```

Not: Bus == 8'b11111111

- **Size of Numbers**

```
assign BusWire = Enable ? Bus : {Width{1'bz}};
```

Replication

Width'bz is illegal

7-13 • Comprehensive Verilog: Operators, Parameters, Hierarchy

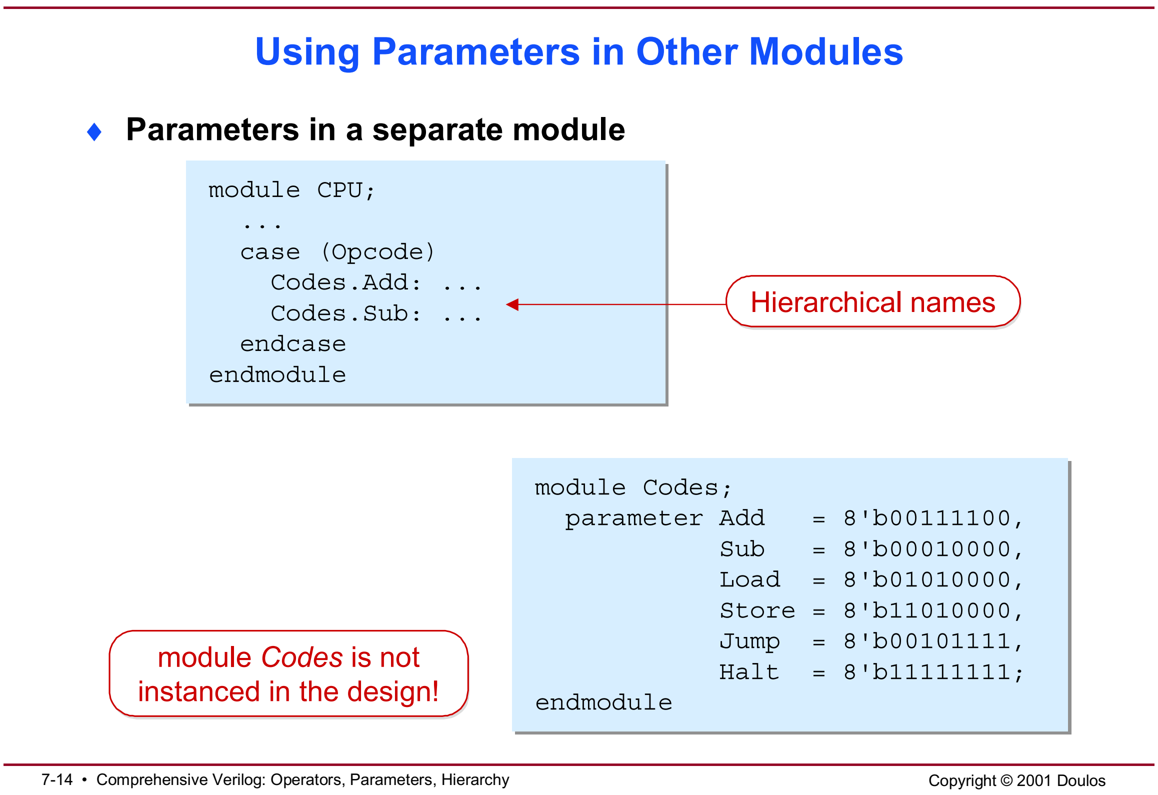
Notes:

Here are some examples where parameters are used in place of literal values.

It is good practice to give names to any meaningful literal values (sometimes called "magic numbers") in a design. This makes the Verilog code easier to understand and to maintain - if the value changes it only needs to be edited in one place. Both parameters and `define macros can be used in this way.

Note the use of replication to assign a high impedance value to BusWire. A parameter name cannot be used as the size of a number; instead, use replication, as shown.

Using Parameters in Other Modules



Using Parameters in Other Modules

- **Parameters in a separate module**

```
module CPU;
  ...
  case (Opcode)
    Codes.Add: ...
    Codes.Sub: ...
  endcase
endmodule
```

Hierarchical names

```
module Codes;
  parameter Add   = 8'b00111100,
            Sub   = 8'b00010000,
            Load  = 8'b01010000,
            Store = 8'b11010000,
            Jump  = 8'b00101111,
            Halt  = 8'b11111111;
endmodule
```

module *Codes* is not instanced in the design!

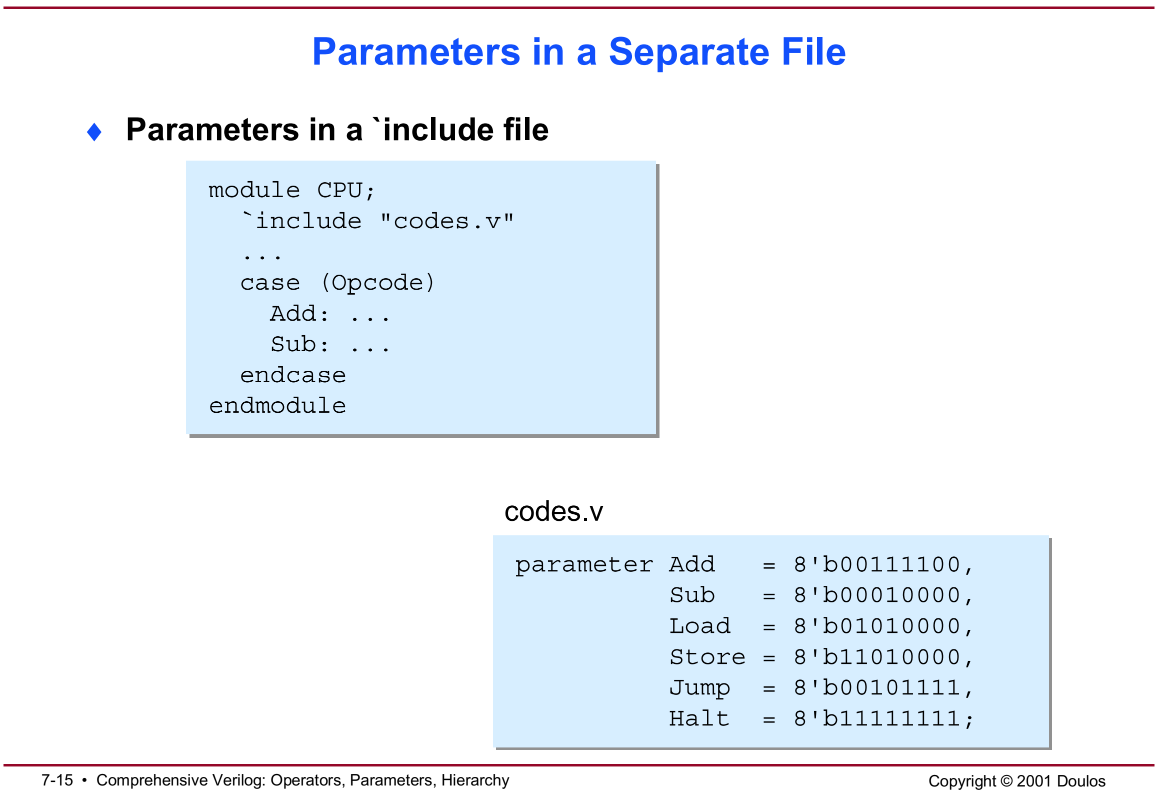
7-14 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Notes:

Modules are normally used to represent blocks of hardware or test fixtures. However, modules can be used simply to group together a set of related definitions (e.g. parameters) which can then be referenced using hierarchical names. The hierarchical names are formed by preceding the names of the parameters with the name of the module containing the parameters (Codes) and a full stop. For example, Codes. Add refers to the parameter Add in the module Codes. This module is then simulated alongside the rest of the design: it does not need to be instanced anywhere. See later in this section for more details.

Note that synthesis tools do not generally support hierarchical names. The alternative approach is to use the `include directive as shown next.

Parameters in a Separate File



Parameters in a Separate File

- Parameters in a `include file

```
module CPU;
  `include "codes.v"
  ...
  case (Opcode)
    Add: ...
    Sub: ...
  endcase
endmodule
```

codes.v

```
parameter Add   = 8'b00111100,
          Sub   = 8'b00010000,
          Load  = 8'b01010000,
          Store = 8'b11010000,
          Jump  = 8'b00101111,
          Halt  = 8'b11111111;
```

7-15 • Comprehensive Verilog: Operators, Parameters, Hierarchy

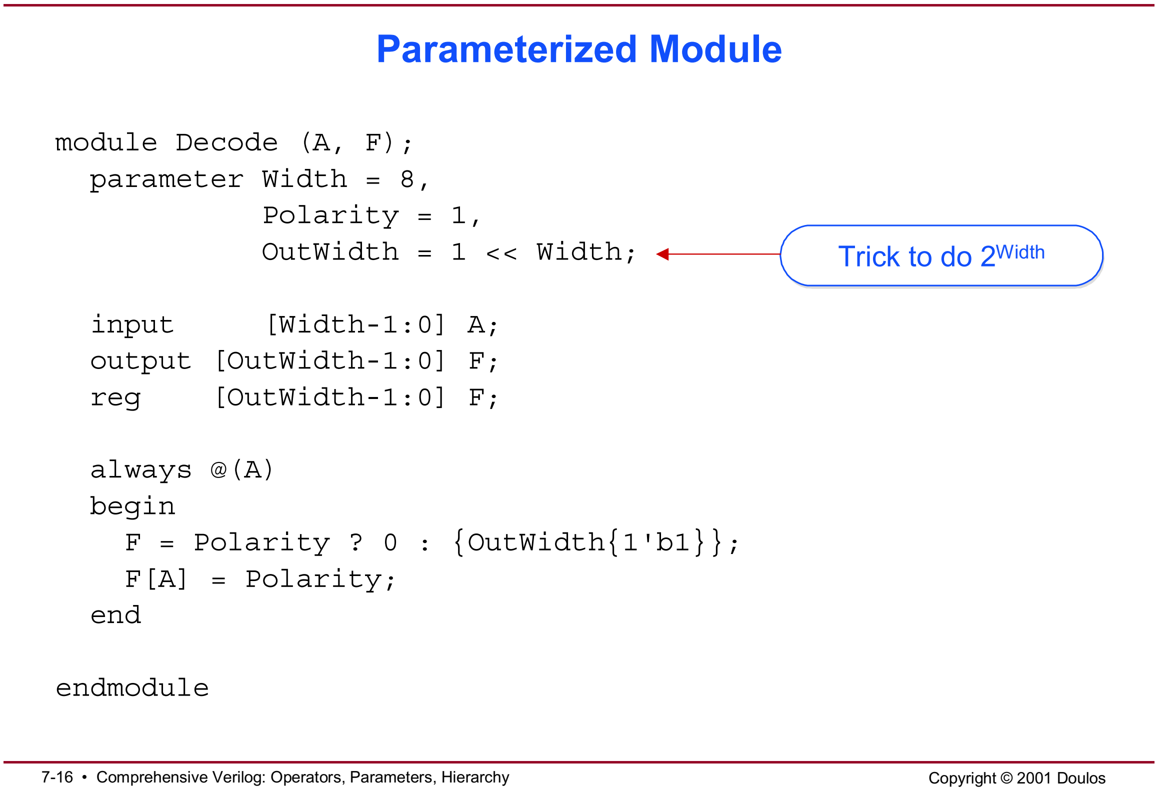
Copyright © 2001 Doulos

Notes:

Parameter definitions can be placed in a separate file and included where required using the `include directive.

Note that (unlike #define in C) `include statements may be included both outside and inside modules.

Parameterized Module



Parameterized Module

```
module Decode (A, F);
  parameter Width = 8,
            Polarity = 1,
            OutWidth = 1 << Width;

  input      [Width-1:0] A;
  output  [OutWidth-1:0] F;
  reg     [OutWidth-1:0] F;

  always @(A)
  begin
    F = Polarity ? 0 : {OutWidth{1'b1}};
    F[A] = Polarity;
  end

endmodule
```

Trick to do 2^{Width}

7-16 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Copyright © 2001 Doulos

Notes:

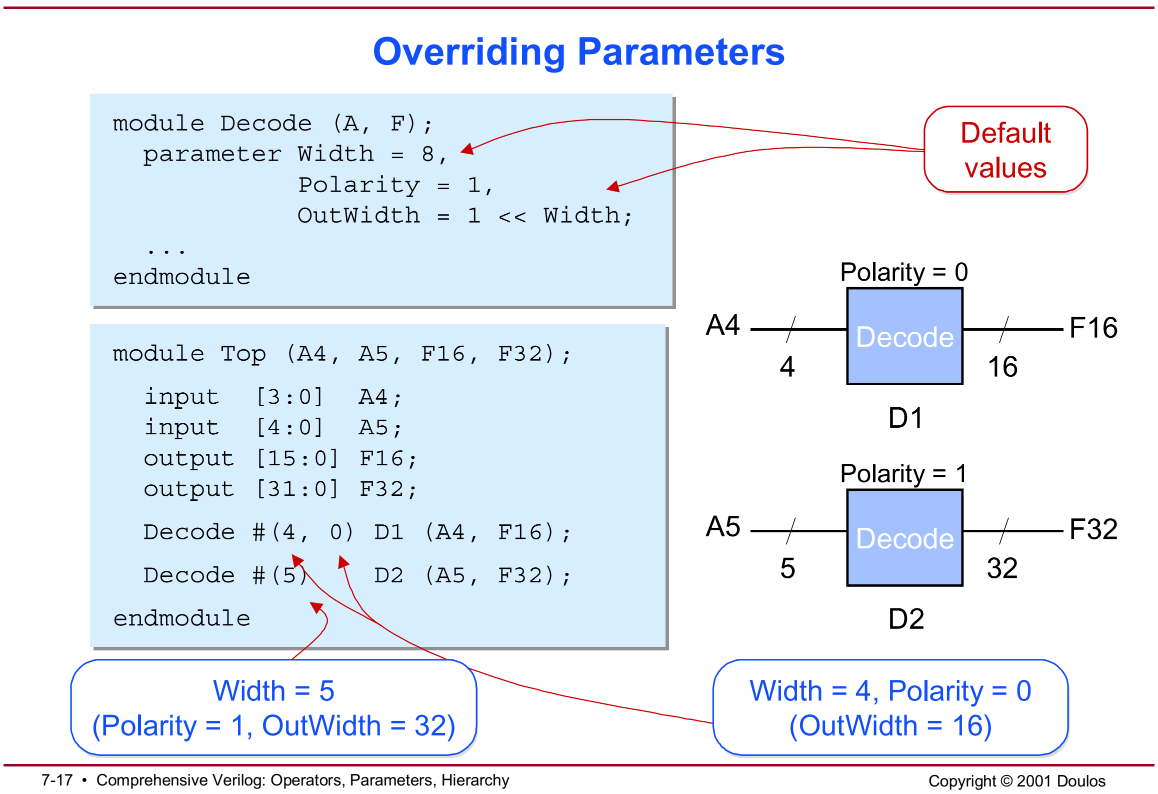
This module uses parameters for the widths of the input, A, and the output, F, of a module Decode. The parameter OutWidth is derived from the value of Width and is declared after the parameters Width and polarity for reasons that will become clear in the slide following this one. There is also a parameter that defines the polarity of the decoder: if Polarity is 1, the module describes a one-hot decoder (one bit is set to 1 and the others are all zero); if Polarity is 0, a one-cold decoder is being described.

Note the use of replication to assign all the bits of F to 1 when Polarity is 0.

Using parameters in this way makes it easy to modify the details of the module - you simply change the values of the parameters. This can be done with a text editor, or as described on the next slide.

Another point to notice here is that although Verilog does not have an exponentiation ("power of") operator, powers of 2 can be described using the left shift operator as shown.

Overriding Parameters



Overriding Parameters

```
module Decode (A, F);
  parameter Width = 8,
            Polarity = 1,
            OutWidth = 1 << Width;
  ...
endmodule
```

```
module Top (A4, A5, F16, F32);
  input  [3:0]  A4;
  input  [4:0]  A5;
  output [15:0] F16;
  output [31:0] F32;
  Decode #(4, 0) D1 (A4, F16);
  Decode #(5)    D2 (A5, F32);
endmodule
```

Default values

Width = 5
(Polarity = 1, OutWidth = 32)

Width = 4, Polarity = 0
(OutWidth = 16)

7-17 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Notes:

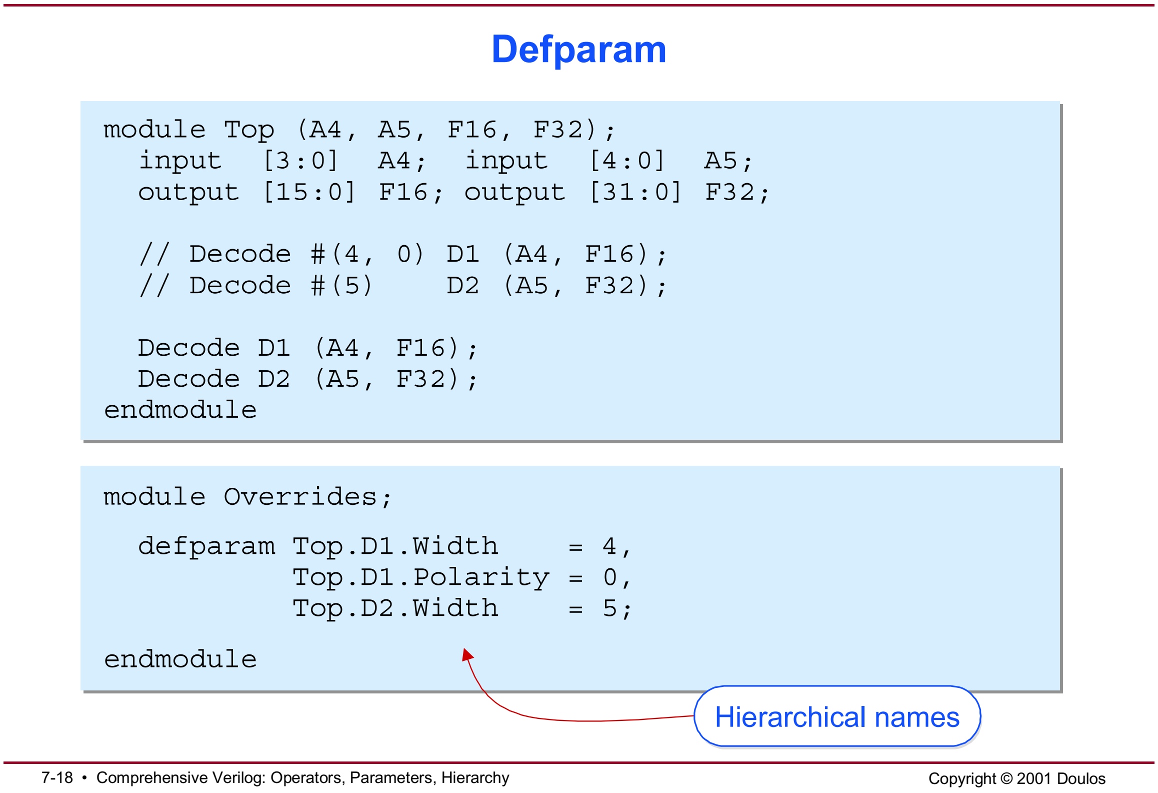
When a module that contains parameters is instanced, the parameters' values can be overridden using the parameter override (#).

In the example shown above #(4, 0) means that for the instance D1, Width (the first parameter declared in the module Decode) will be given the value 4 instead of 1. The second parameter, Polarity, will be given the value 0. Other instances of Decode are not affected.

Be careful not to override the values of derived parameters such as OutWidth: the instanced module may not work correctly! This explains why derived parameters should be declared after those that may be overridden.

Named mapping (which can be used when connecting wires and regs to the ports of a module instance) is not allowed when overriding parameters.

Defparam



Defparam

```
module Top (A4, A5, F16, F32);
  input  [3:0]  A4;  input  [4:0]  A5;
  output [15:0] F16; output [31:0] F32;

  // Decode #(4, 0) D1 (A4, F16);
  // Decode #(5)    D2 (A5, F32);

  Decode D1 (A4, F16);
  Decode D2 (A5, F32);
endmodule
```

```
module Overrides;

  defparam Top.D1.Width    = 4,
           Top.D1.Polarity = 0,
           Top.D2.Width    = 5;

endmodule
```

Hierarchical names

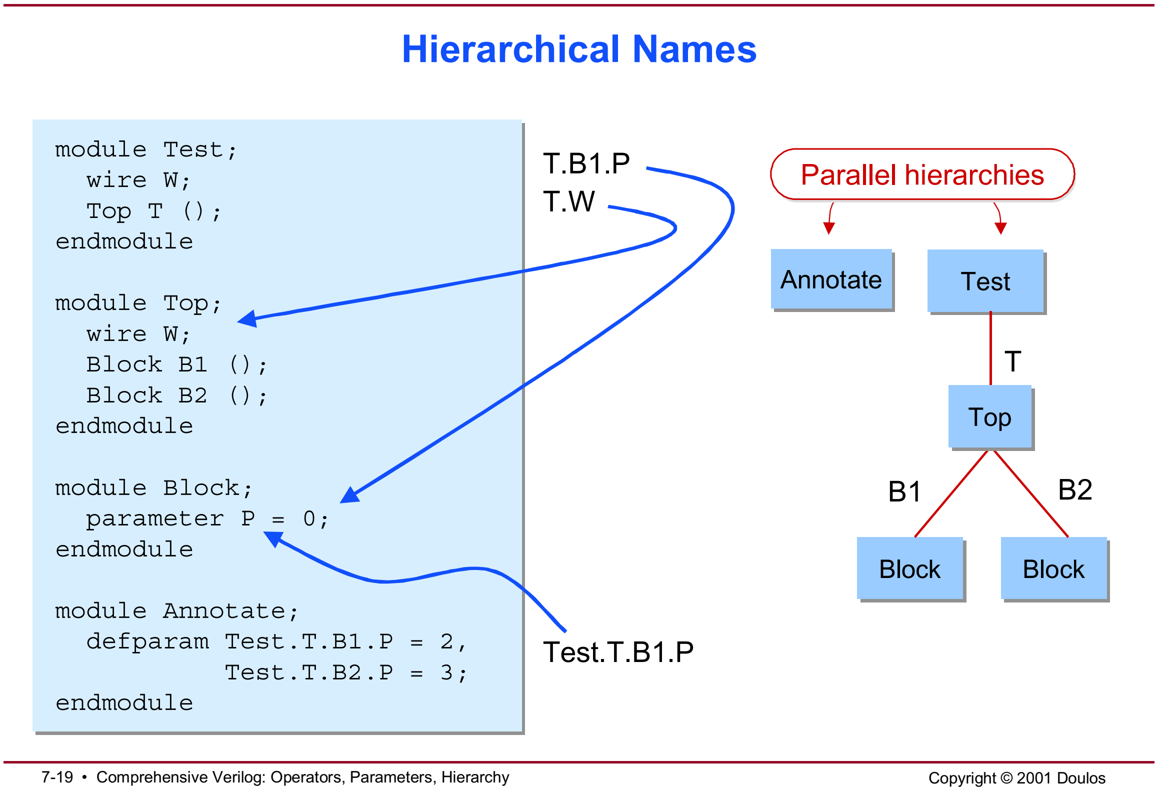
7-18 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Notes:

defparam

Parameters may also be overridden using defparam. Defparam statements usually use the hierarchical names of parameters.

Hierarchical Names



Hierarchical Names

```
module Test;
  wire W;
  Top T ();
endmodule

module Top;
  wire W;
  Block B1 ();
  Block B2 ();
endmodule

module Block;
  parameter P = 0;
endmodule

module Annotate;
  defparam Test.T.B1.P = 2,
           Test.T.B2.P = 3;
endmodule
```

7-19 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Notes:

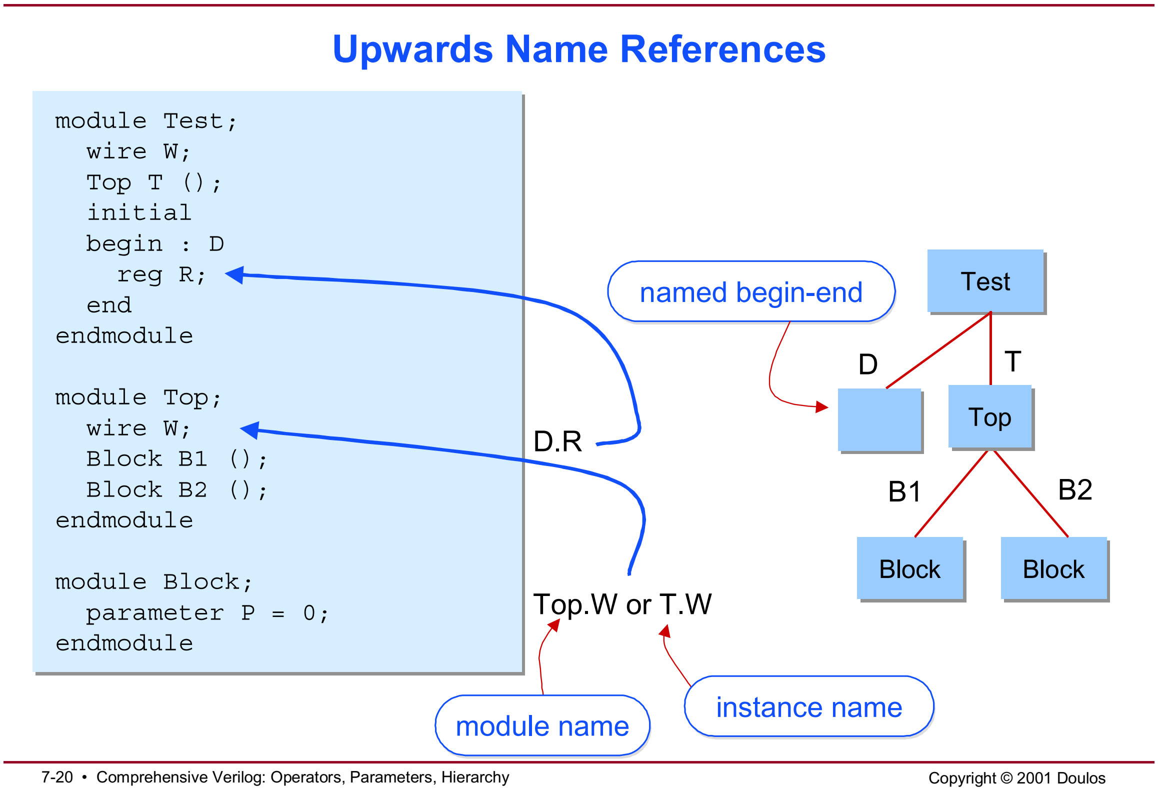
We have already seen that parameters can sometimes be referenced using a hierarchical name. In fact, any Verilog item that has a name can be referenced from anywhere in the module hierarchy using a hierarchical name! This makes it possible to "burgle" Verilog modules and named blocks, i.e. read and write items declared within a block from outside without using ports or arguments. This is very bad practice except in test fixtures, where it is very useful!

The full hierarchical name of any item starts with the name of a top level module, then contains module instance names and block names down through the hierarchy.

A hierarchical name can also be formed starting with the instance name of a module or the name of a named block. For example, a $display statement in the test fixture could reference items in the hierarchy of the design being simulated using such hierarchical names.

A Verilog simulation can have more than one top level module! The two hierarchies can communicate using hierarchical names. Here, the names in the module Annotate refer to parameters in the design whose top-level module is Test.

Upwards Name References



Upwards Name References

```
module Test;
  wire W;
  Top T ();
  initial
  begin : D
    reg R;
  end
endmodule

module Top;
  wire W;
  Block B1 ();
  Block B2 ();
endmodule

module Block;
  parameter P = 0;
endmodule
```

7-20 • Comprehensive Verilog: Operators, Parameters, Hierarchy

Copyright © 2001 Doulos

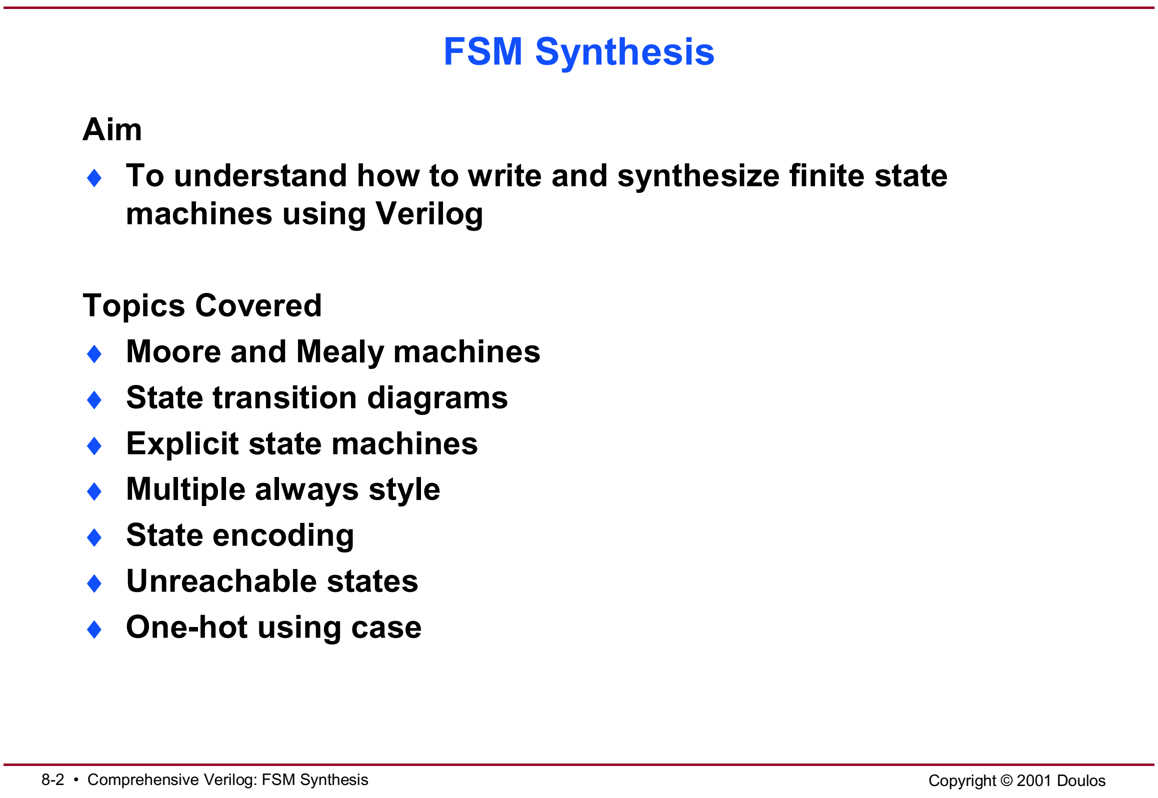
Notes:

The hierarchical names we have been looking are all full or downwards references: these hierarchical names start at the top-level module or with a module instance name or block name in the module in which the hierarchical name is used.

Verilog also supports upwards name referencing. This means that a module can reference a module or named block higher up in the hierarchy, using either an instance name of the module or its module name, or a block name.

Module 8
FSM Synthesis

FSM Synthesis



FSM Synthesis

Aim

- To understand how to write and synthesize finite state machines using Verilog

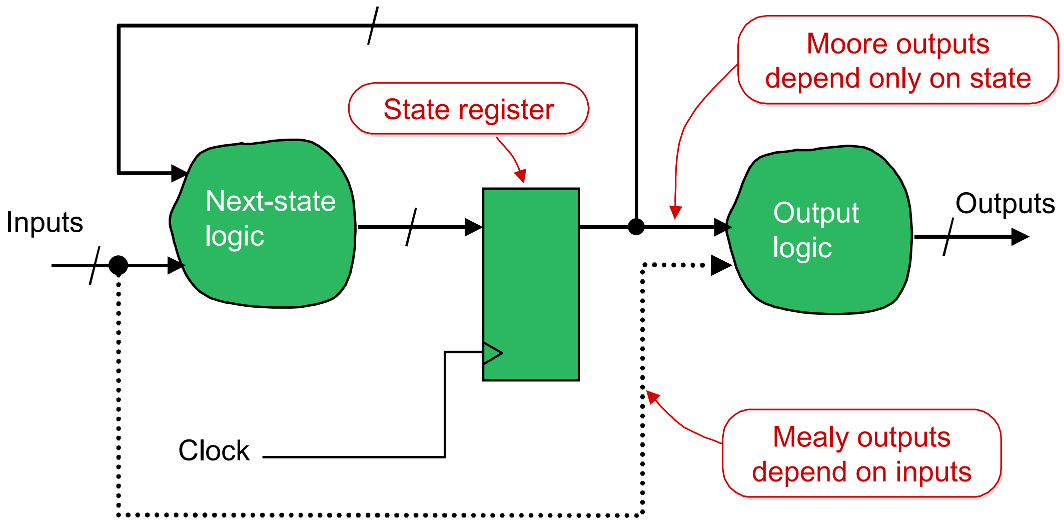
Topics Covered

- Moore and Mealy machines
- State transition diagrams
- Explicit state machines
- Multiple always style
- State encoding
- Unreachable states
- One-hot using case

Notes:

Finite State Machines

Finite State Machines


Inputs
Next-state logic
State register
Moore outputs depend only on state
Output logic
Outputs
Clock
Mealy outputs depend on inputs

Notes:

A finite state machine is a very useful abstraction for digital circuit design. As the name suggests, a finite state machine is a digital logic circuit with a finite number of internal states. The FSM uses the values of its inputs and its current state to determine the values of its outputs and its next state. We will consider only synchronous state machines, where the state machine changes state on the active edge of the clock.

The hardware architecture of a finite state machine consists of a state register, combinational logic to calculate the next state, and combinational logic to calculate the outputs.

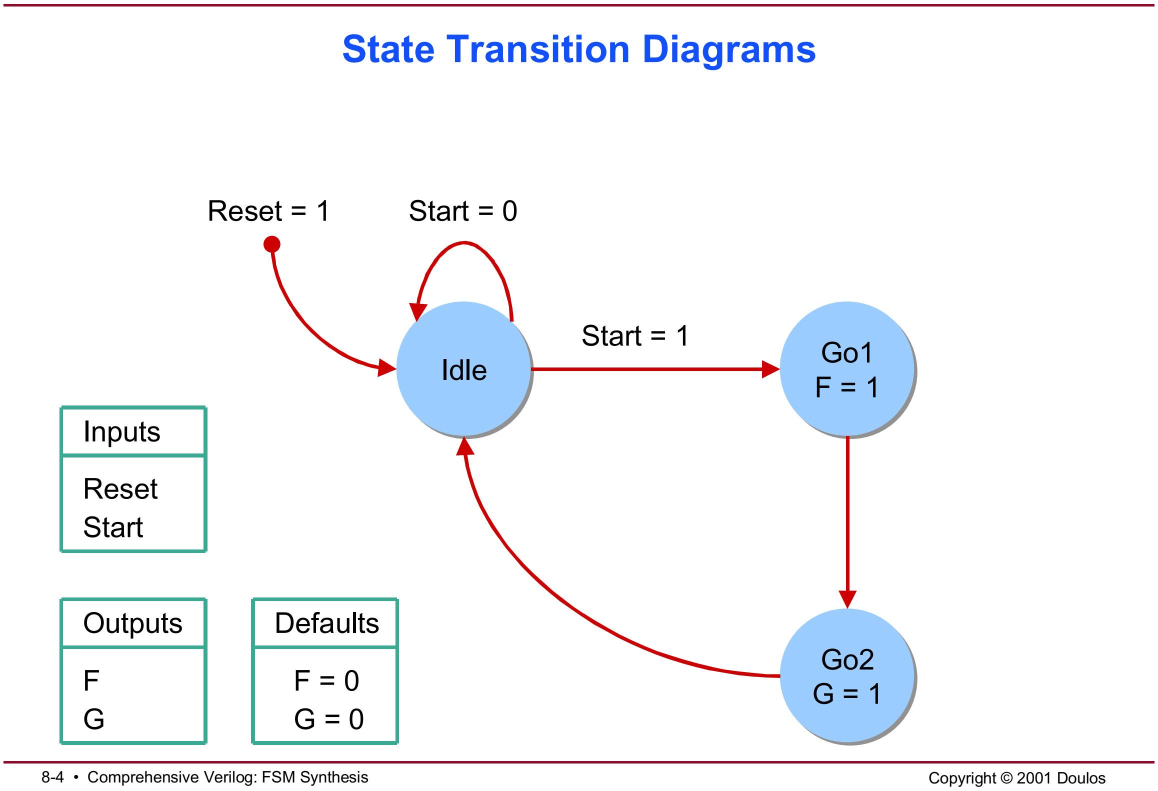
Moore outputs

Moore outputs depend only on the current state of the machine, such that the outputs only change value when the state machine changes state.

Mealy outputs

Mealy outputs depend on the current values of the inputs as well as the current state, such that a change in the inputs can cause the outputs to change after just a combinational delay, without having to wait for the next state transition.

State Transition Diagrams



State Transition Diagrams

Inputs
Reset Start

Outputs
F G

Defaults

F = 0
G = 0

8-4 • Comprehensive Verilog: FSM Synthesis

Copyright © 2001 Doulos

Notes:

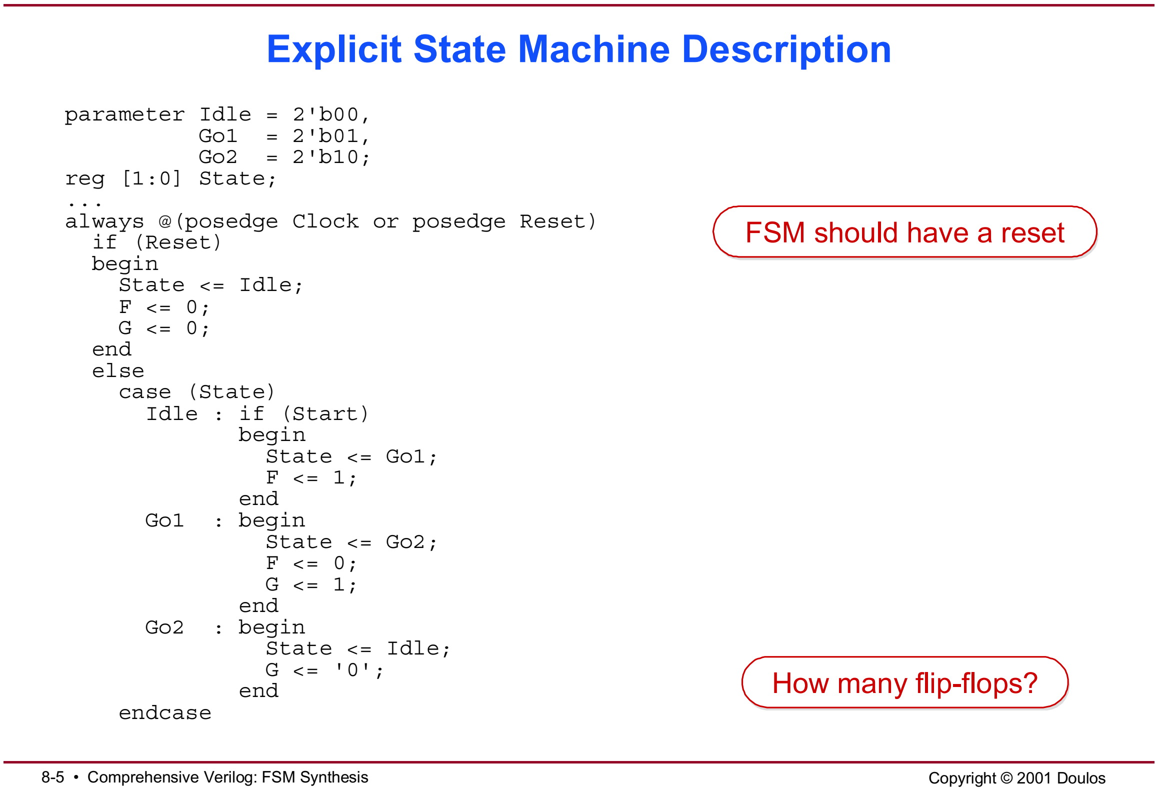
A finite state machine can be conveniently represented using a state transition diagram, as shown here. Each bubble on the diagram represents a different internal state of the machine. The arrows connecting the bubbles represent state transitions, and the annotations on some of the arrows (e.g. Start = 1) are the input conditions under which that particular state transition will occur. Each bubble contains a symbolic name for that state (e.g. Idle), and the output values in that state. These are Moore outputs, because the output value is a function only of which state the machine is in. Mealy outputs would be shown as annotations on the transition arrows.

State transition diagrams often take advantage of notational shortcuts. For example, transitions that start from a dot rather than from another state (e.g. Reset

= 1) are global transitions. These take priority over other input conditions, and cause the state machine to jump directly to the given state from any other state. This avoids cluttering the diagram with reset transitions from every state.

As another example of a notational shortcut, the outputs on this diagram are assumed to have a default value of zero. The diagram only shows the values of the outputs explicitly when they are non-zero.

Explicit State Machine Description



Explicit State Machine Description

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
...
always @(posedge Clock or posedge Reset)
  if (Reset)
  begin
    State <= Idle;
    F <= 0;
    G <= 0;
  end
  else
    case (State)
      Idle : if (Start)
             begin
               State <= Go1;
               F <= 1;
             end
      Go1  : begin
               State <= Go2;
               F <= 0;
               G <= 1;
             end
      Go2  : begin
               State <= Idle;
               G <= '0';
             end
    endcase
```

FSM should have a reset

How many flip-flops?

8-5 • Comprehensive Verilog: FSM Synthesis

Notes:

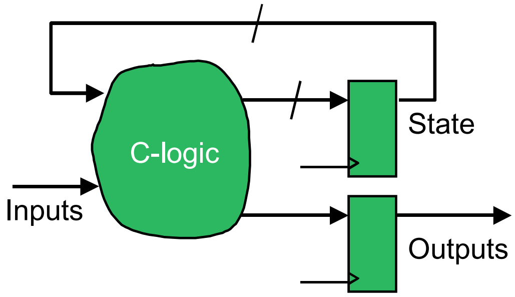
There are many ways to describe a finite state machine in Verilog. The most common approach is to use an always statement containing a case statement. The state of the machine is stored in a state register, and the possible states are represented with parameter values.

Initialization

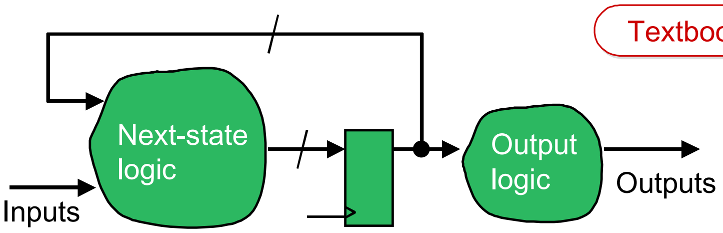
Finite state machines must be initialized by means of an explicit reset signal. Otherwise, there is no reliable way to get the Verilog and gate level representation of the FSM into the same known state, and thus no way to verify their equivalence.

State Machine Architecture

State Machine Architecture



Preceding *always*



Textbook Moore machine

Notes:

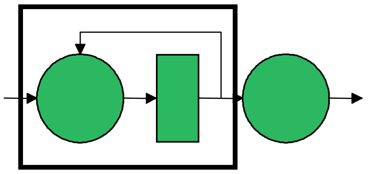
The preceding description of a finite state machine consists of an always statement, synchronized on a clock edge, and assigning the Verilog regs representing the state vector (State) and the outputs (F and G). Synthesis will generate flip-flops for each of these regs, in order to ensure that both the state transitions and the outputs are synchronized to the clock. However, the flip-flops on the outputs may well be unwanted! A textbook Moore machine consists just of registers to store the state vector, and combinational logic to decode the next state and outputs from the current state. In order to eliminate the unwanted flip-flops, we must re-write the Verilog.

Separate Output Decoding

Separate Output Decoding

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;

...

always @(posedge Clock or posedge Reset)
  if (Reset)
    State <= Idle;
  else
    case (State)
      Idle : if (Start)
               State <= Go1;
      Go1  : State <= Go2;
      Go2  : State <= Idle;
    endcase
```

```
always @(State)
begin
  F = 0;
  G = 0;
  if (State == Go1)
    F = 1;
  else if (State == Go2)
    G = 1;
end
```

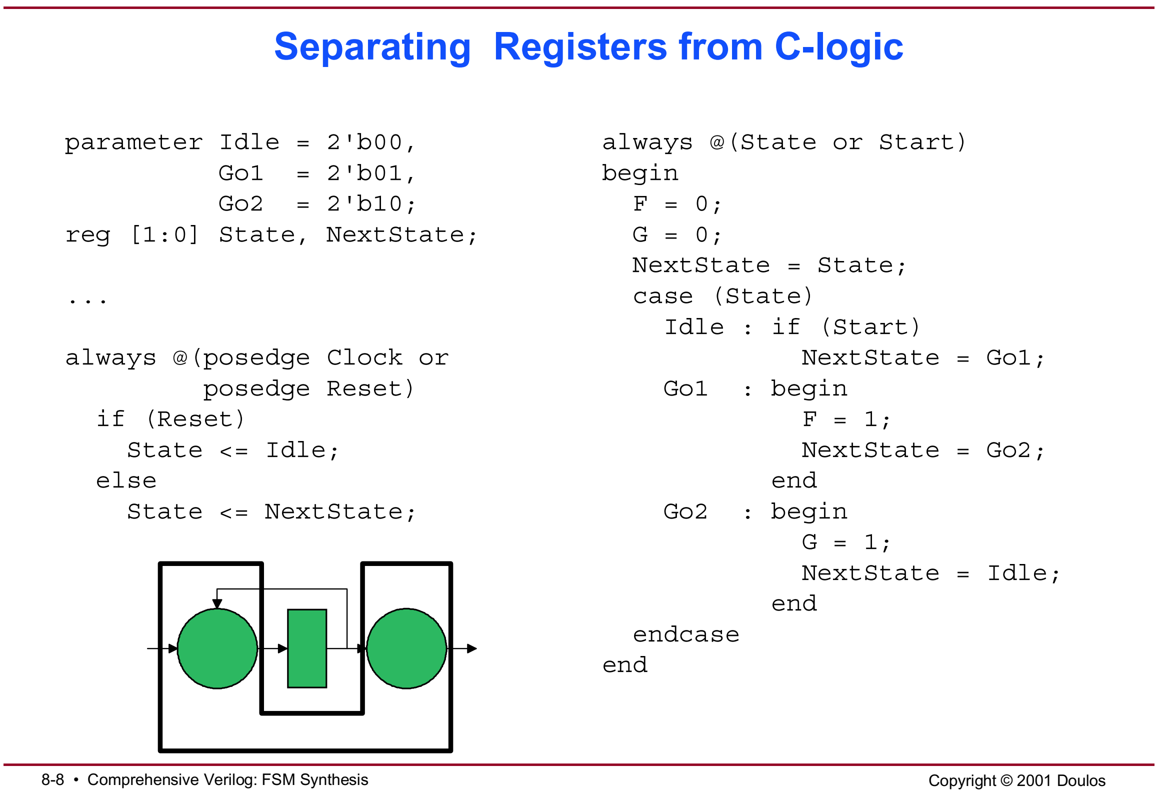


Notes:

In order to eliminate unwanted flip-flops, we must split the description into at least two always statements. There are many different ways of splitting up the description into multiple always statements, and two of the most common styles are shown above and on the next page.

The code above shows the previous state machine coded using two always statements, one describing the state vector and state transitions, the other describing the output decoding logic. Since the output always statement is now purely combinational, the extra flip-flops are eliminated.

Separating Registers from C-logic



Separating Registers from C-logic

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State, NextState;

...

always @(posedge Clock or
         posedge Reset)
  if (Reset)
    State <= Idle;
  else
    State <= NextState;
```

```
always @(State or Start)
begin
  F = 0;
  G = 0;
  NextState = State;
  case (State)
    Idle : if (Start)
             NextState = Go1;
    Go1  : begin
             F = 1;
             NextState = Go2;
           end
    Go2  : begin
             G = 1;
             NextState = Idle;
           end
  endcase
end
```

8-8 • Comprehensive Verilog: FSM Synthesis

Notes:

Another common style for describing state machines is to separate the registers from the combinational logic. The example shows the same state machine with one always statement describing the state registers, and the second one describing the next state logic and the output decoding logic.

An additional reg, NextState, is needed so that the combinational always statement can communicate with the clocked always statement. As NextState is a combinational function of State and Start, it must be completely assigned in the always block (otherwise latches will be inferred). The assignment

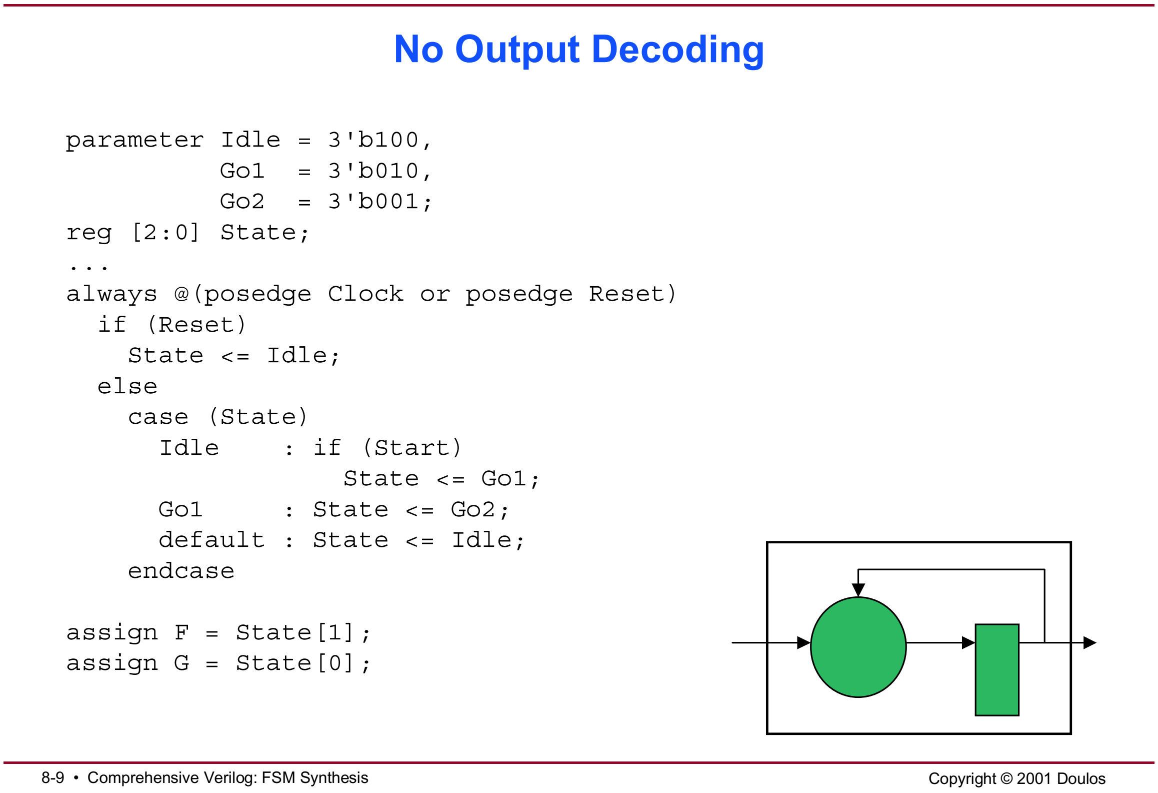
NextState = State ensures that no latches will be inferred. Note that this statement does not affect the simulation: it is only required because of the way synthesis tools interpret the always block.

A possible advantage over the "separate output decoding" style is that any commonality between the next state and output decoding logic need not be duplicated in two always statements, making the code shorter and the optimizer's job easier.

A possible disadvantage over the "separate output decoding" style is that simulation will be less efficient if the next state logic depends on the inputs (which is not the case opposite), because additional executions of the combinational always statement would occur whenever an input changed.

Both styles of description result in correct behavior and correct synthesis, and both styles have their enthusiastic followers! Which to chose is often a matter of personal taste.

No Output Decoding



No Output Decoding

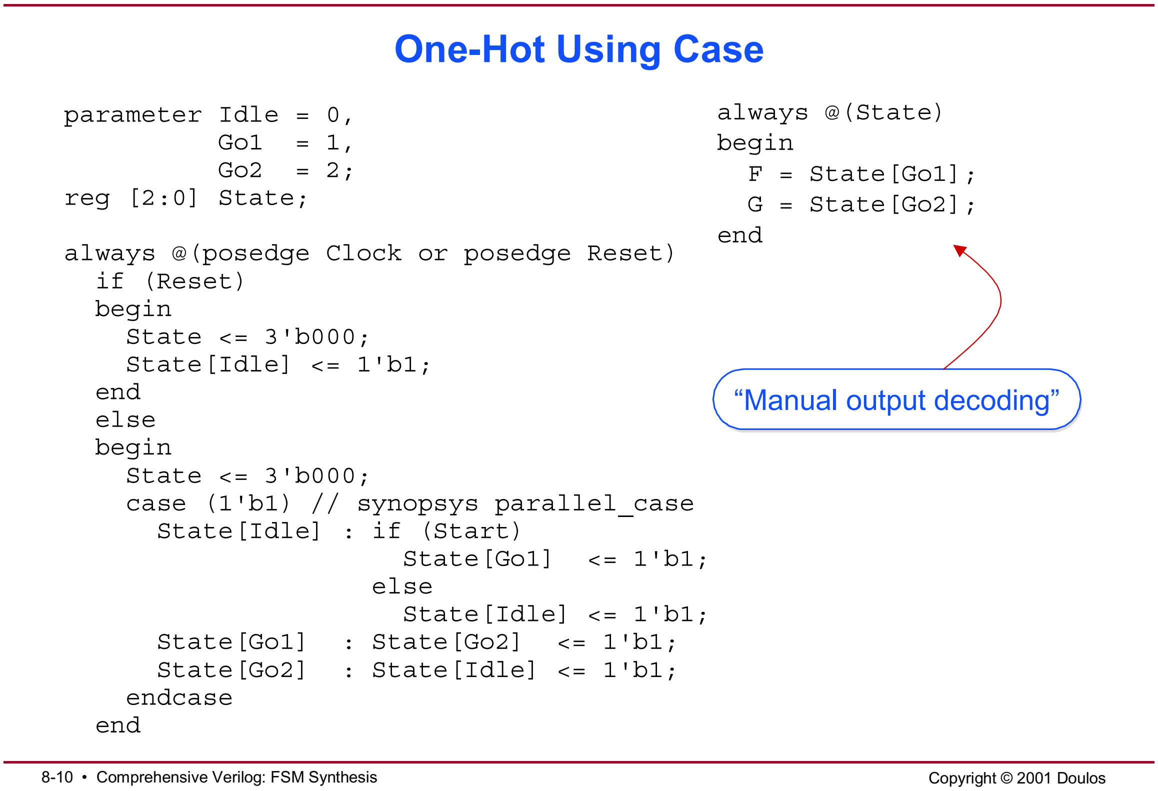
```
parameter Idle = 3'b100,
          Go1  = 3'b010,
          Go2  = 3'b001;
reg [2:0] State;
...
always @(posedge Clock or posedge Reset)
  if (Reset)
    State <= Idle;
  else
    case (State)
      Idle    : if (Start)
                  State <= Go1;
      Go1     : State <= Go2;
      default : State <= Idle;
    endcase

assign F = State[1];
assign G = State[0];
```

8-9 • Comprehensive Verilog: FSM Synthesis

Notes:

Sometimes we want to eliminate the output decoding logic altogether from the design of the finite state machine, by making the state vector encoding for each state equivalent to the output values for that state. To code this up in Verilog, you need to work out an appropriate encoding and declare the state parameters accordingly. Finally, the bits of the state vector are copied to the outputs.

One-Hot Using Case



One-Hot Using Case

```
parameter Idle = 0,
          Go1  = 1,
          Go2  = 2;
reg [2:0] State;

always @(posedge Clock or posedge Reset)
  if (Reset)
  begin
    State <= 3'b000;
    State[Idle] <= 1'b1;
  end
  else
  begin
    State <= 3'b000;
    case (1'b1) // synopsys parallel_case
      State[Idle] : if (Start)
                      State[Go1]  <= 1'b1;
                    else
                      State[Idle] <= 1'b1;
      State[Go1]  : State[Go2]  <= 1'b1;
      State[Go2]  : State[Idle] <= 1'b1;
    endcase
  end
```

```
always @(State)
begin
  F = State[Go1];
  G = State[Go2];
end
```

“Manual output decoding”

8-10 • Comprehensive Verilog: FSM Synthesis

Copyright © 2001 Doulos

Notes:

A one-hot state encoding is often a good choice for FSMs that are being implemented in FPGAs. This is because one-hot encoding suits the standard FPGA architecture.

The previous slide does in fact use a one-hot encoding, but the synthesis tool will not realize this fact and will fully decode the state flip-flop values to determine what the state is.

In the example here, the parameters Idle, Go1 and Go2 are no longer the flip-flop values for the states, but the position of the flip-flop for the corresponding state in the state flip-flops. For example, in the Idle state, State[Idle] will be 1'b1 and State[Go1] and State[Go2] will both be 1'b0.

The example uses a case statement with a constant case expression (1'b1) and variable case values (the bits of State). The parallel_case directive tells the synthesis tool that only one case value (State[Idle], State[Go1] or State[Go2]) matches the case expression (1'b1) i.e. the encoding is one-hot.

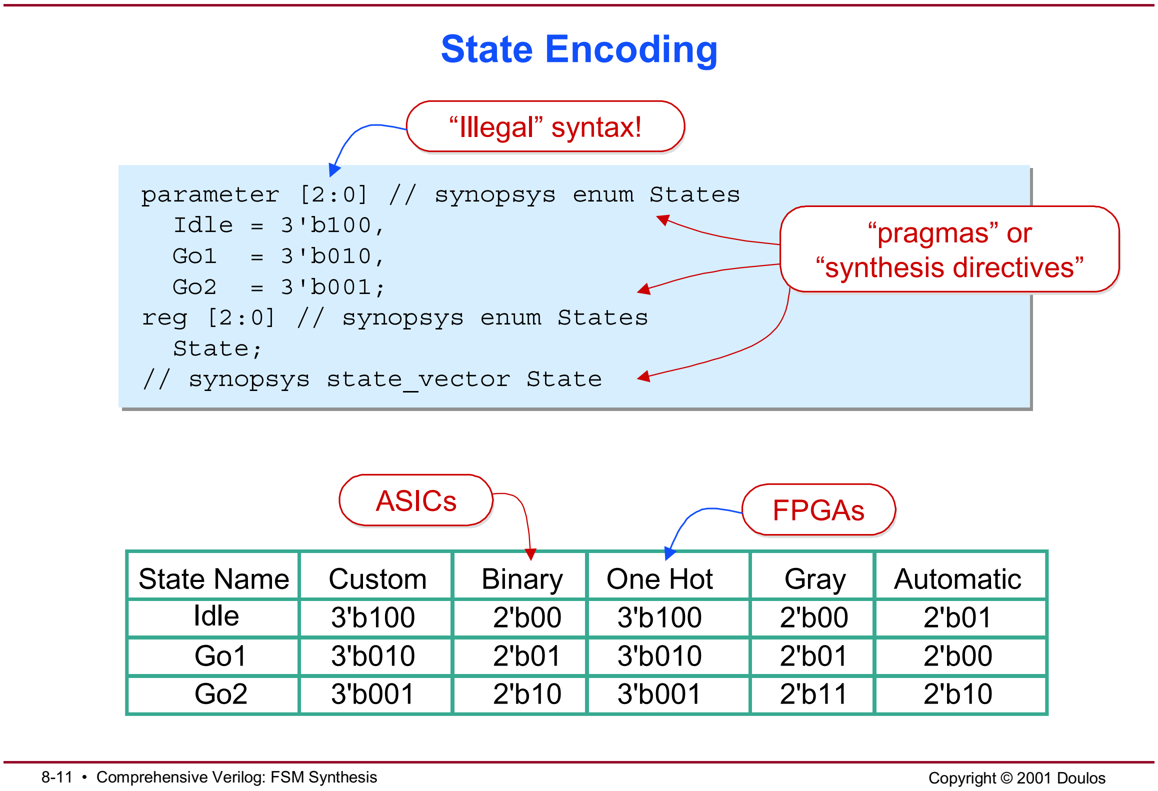
Note that the outputs are decoded manually. The following code would produce inefficient logic, as the synthesis tool would not assume a one-hot encoding.

```
always @(State)
begin
  F = 0;
  G = 0;
  if (State == Go1)
    F = 1;
  else if (State == Go2)
    G = 1;
end
```

However, another case statement with a parallel_case directive could be used, similar to the one in the state logic always block:

```
always @(State)
begin
  F = 0;
  G = 0;
  case (1'b1) // synopsys parallel_case
    State[Go1] : F = 1;
    State[Go2] : G = 1;
  endcase
end
```

State Encoding



State Encoding

"Illegal" syntax!
"pragmas" or "synthesis directives"
ASICs
FPGAs

```
parameter [2:0] // synopsys enum States
  Idle = 3'b100,
  Go1  = 3'b010,
  Go2  = 3'b001;
reg [2:0] // synopsys enum States
  State;
// synopsys state_vector State
```

State Name	Custom	Binary	One Hot	Gray	Automatic
Idle	3'b100	2'b00	3'b100	2'b00	2'b01
Go1	3'b010	2'b01	3'b010	2'b01	2'b00
Go2	3'b001	2'b10	3'b001	2'b11	2'b10

8-11 • Comprehensive Verilog: FSM Synthesis

Notes:

Usually, a synthesis tool will encode each state using the values of the parameters corresponding to the state names.

Some synthesis tools recognize that a state machine is being described, and automatically modify the encodings. Other tools modify the encodings if certain synthesis directives or pragmas are present in the Verilog source code.

The example above shows the syntax required by Synopsys synthesis tools to enable the tools to change the state encodings. Other tools support a similar syntax. Please refer to the documentation for your synthesis tool for details.

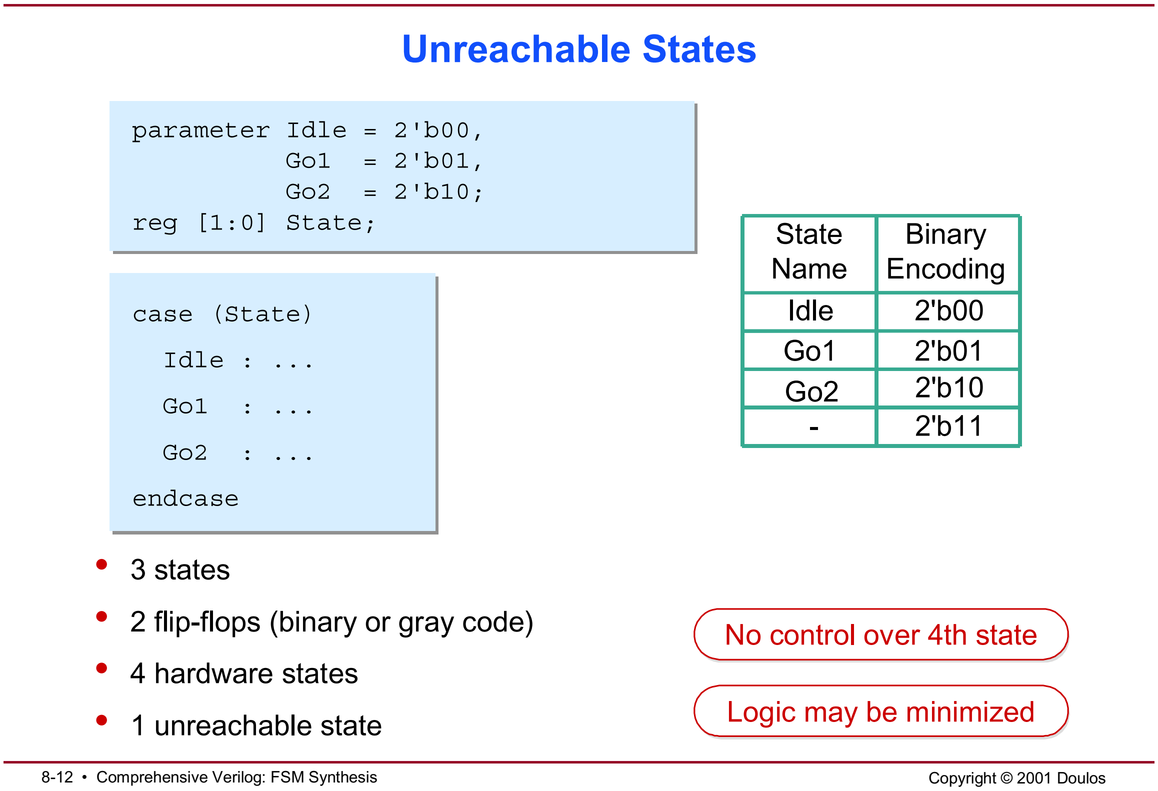
The new encoding will usually be either binary or one hot, depending on the target technology. With binary, the first state in the list of parameters becomes binary 0, the second state binary 1 and so on. With one hot, there is one flip-flop per state, and only one flip-flop is 1 in each state.

Many synthesis tools also allow the user to choose from a number of standard encoding methods, such as Gray, One Hot, Adjacency, Random, LFSR and so on. Many tools also provide an automatic encoding method that chooses the "best" encoding using various "clever" algorithms. In practice, however, the best results are often obtained by using a little engineering knowledge, and specifying an appropriate encoding manually.

(In this example, the directive // synopsys enum States tells the synthesis tool that the parameters Idle, Go1 and Go2 are to be considered to be members of an enum type called States. The reg State is then declared to have the type States. This means that it may only take one of the values Idle, Go1 and Go2 - you are not allowed to assign numerical values, only these parameter names. The other directive - // synopsys state_vector State - tells the tool that State is a state vector in a state machine. The tool will usually work this out automatically.

Note also that including a vector range in a parameter declaration - which is required by some synthesis tools in this context - is not allowed according to the IEEE 1364 Verilog standard. However, most tools do support this extension to the language.

Unreachable States



Unreachable States

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
```

```
case (State)
  Idle : ...
  Go1  : ...
  Go2  : ...
endcase
```

State Name	Binary Encoding
Idle	2'b00
Go1	2'b01
Go2	2'b10
-	2'b11

- 3 states
- 2 flip-flops (binary or gray code)
- 4 hardware states
- 1 unreachable state

No control over 4th state

Logic may be minimized

8-12 • Comprehensive Verilog: FSM Synthesis

Copyright © 2001 Doulos

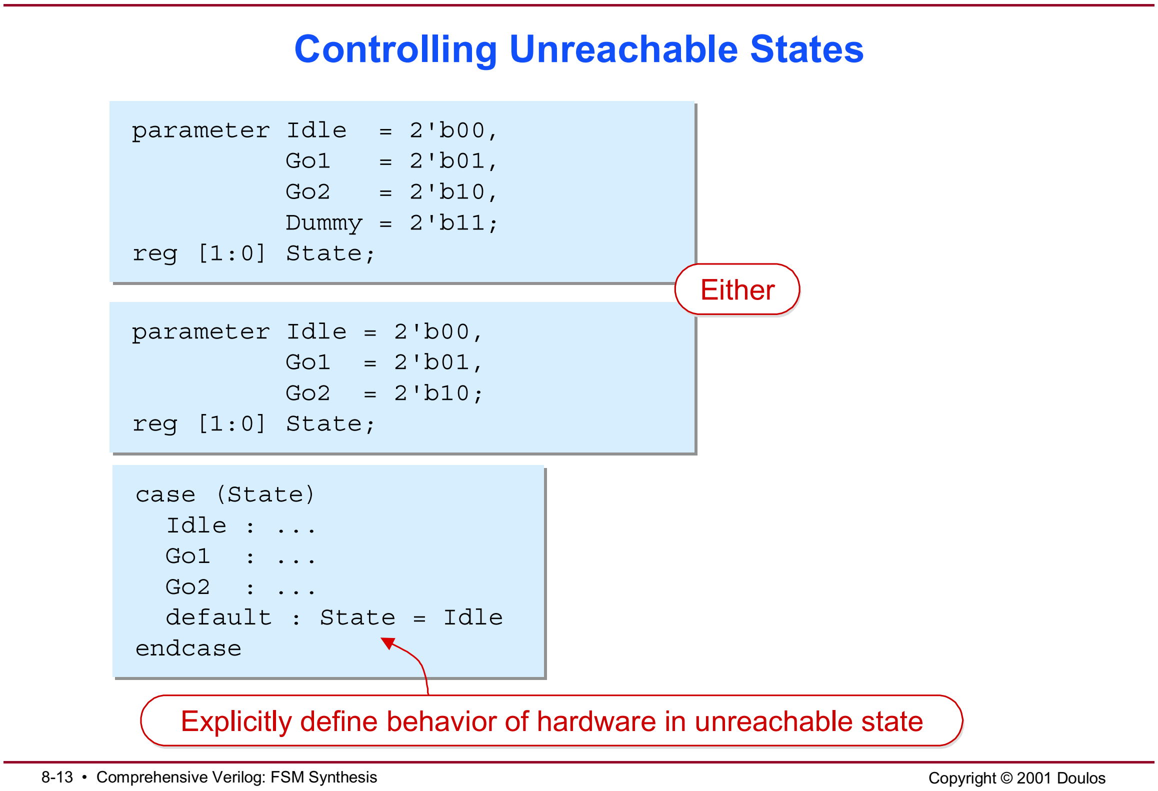
Notes:

When a state machine is implemented in hardware, the number of states will be 2N, where N is the number of bits in the state vector. If the state machine has fewer than 2N states, then the remaining states are unreachable during the normal logical operation of the state machine. However, the state machine may power up into one of these unreachable states, or move into such a state due to extraordinary operating conditions.

Don't care

If the behavior of the state machine in any unreachable state is left unspecified, then some synthesis tools can optimize the logic to exploit the fact that the designer does not care what happens in such states.

Controlling Unreachable States



Controlling Unreachable States

```
parameter Idle  = 2'b00,
          Go1   = 2'b01,
          Go2   = 2'b10,
          Dummy = 2'b11;
reg [1:0] State;
```

Either

```
parameter Idle = 2'b00,
          Go1  = 2'b01,
          Go2  = 2'b10;
reg [1:0] State;
```

```
case (State)
  Idle : ...
  Go1  : ...
  Go2  : ...
  default : State = Idle
endcase
```

Explicitly define behavior of hardware in unreachable state

8-13 • Comprehensive Verilog: FSM Synthesis

Notes:

On the other hand, if you wish to control the behavior of the state machine in unreachable states (e.g. by moving directly to the reset state), then you must specify the behavior explicitly in the Verilog source code by defining all 2N states. This may mean adding extra parameter names to bring the number of values up to a power of 2, then adding appropriate code to define the state transitions.

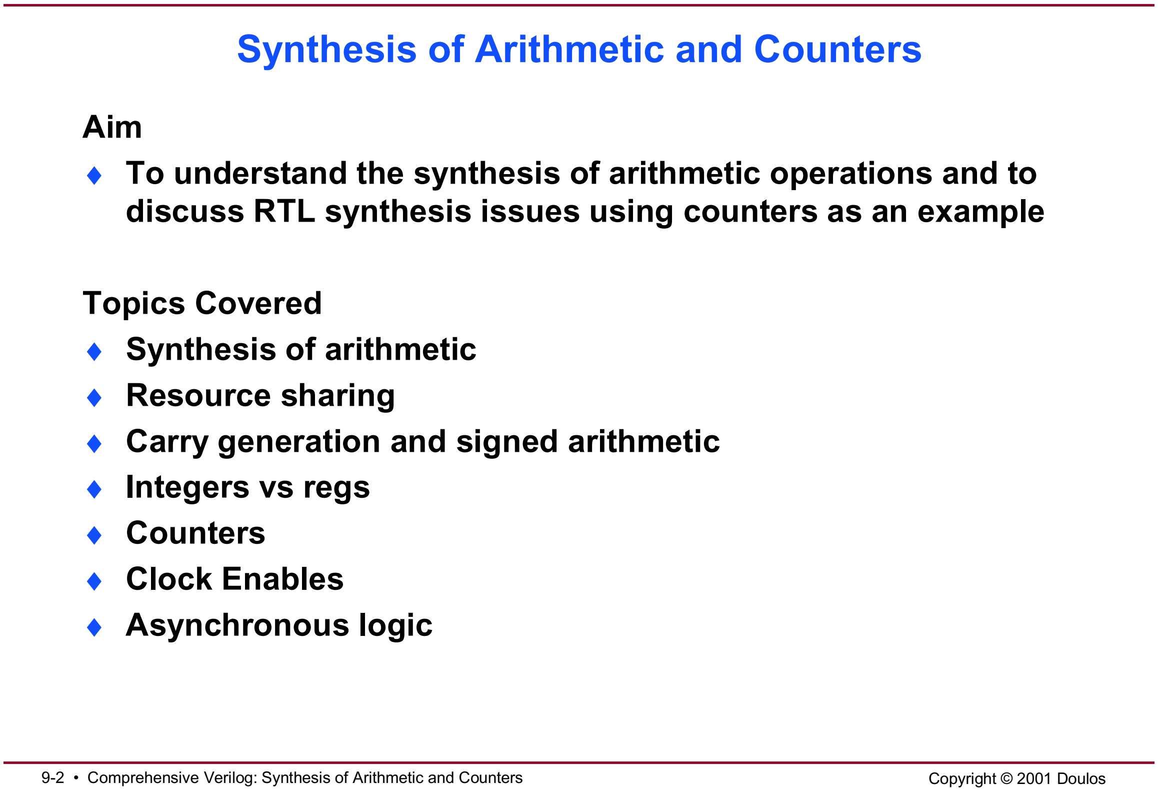
If you are concerned about defining the behavior in unreachable states, then it is important not to allow FSM optimization to merge equivalent states. In examples with several dummy states, if you switch on FSM optimization, then the dummy states may get merged during optimization, resulting in some well defined states (one of which is unreachable) and other truly undefined unreachable states!

Default

Many synthesis tools are influenced by the Verilog code written within the default part of a case statement, regardless of whether there actually are any other cases!

Module 9
Synthesis of Arithmetic and Counters

Synthesis of Arithmetic and Counters



Synthesis of Arithmetic and Counters

Aim

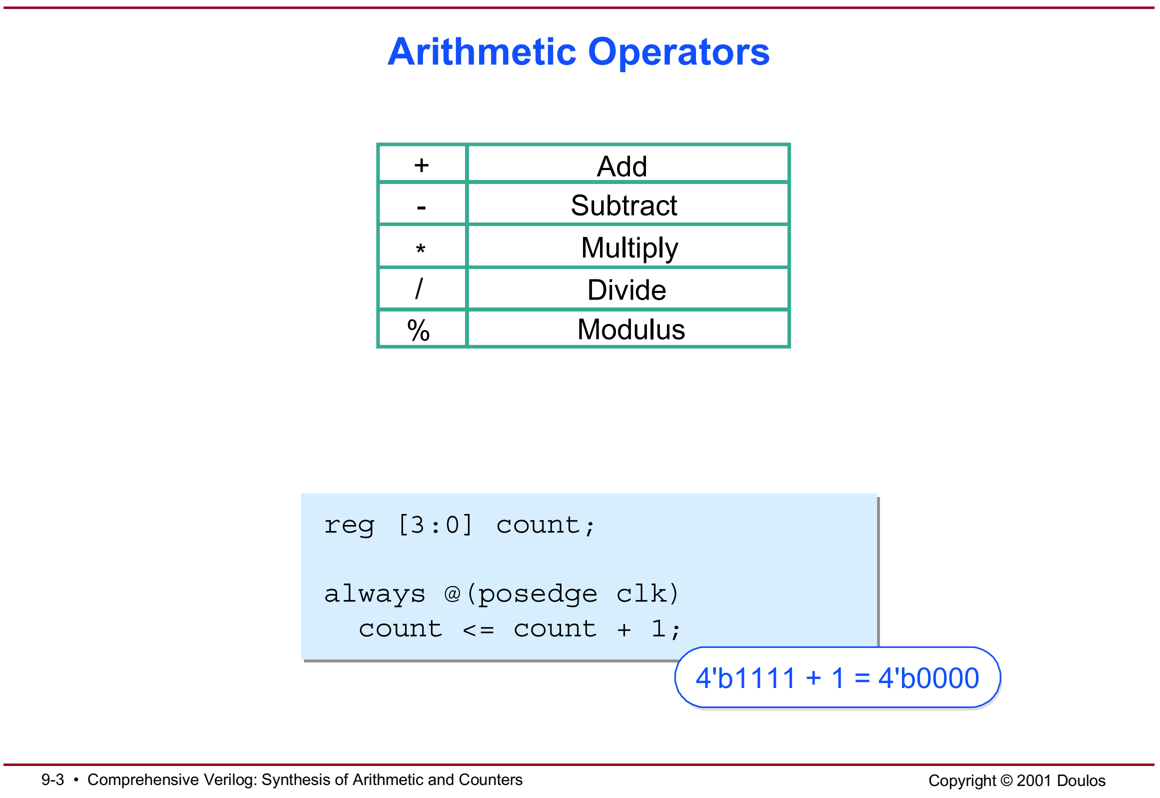
- To understand the synthesis of arithmetic operations and to discuss RTL synthesis issues using counters as an example

Topics Covered

- Synthesis of arithmetic
- Resource sharing
- Carry generation and signed arithmetic
- Integers vs regs
- Counters
- Clock Enables
- Asynchronous logic

Notes:

Arithmetic Operators



Arithmetic Operators

+	Add
-	Subtract
*	Multiply
/	Divide
%	Modulus

```
reg [3:0] count;

always @(posedge clk)
  count <= count + 1;
```

4'b1111 + 1 = 4'b0000

9-3 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Notes:

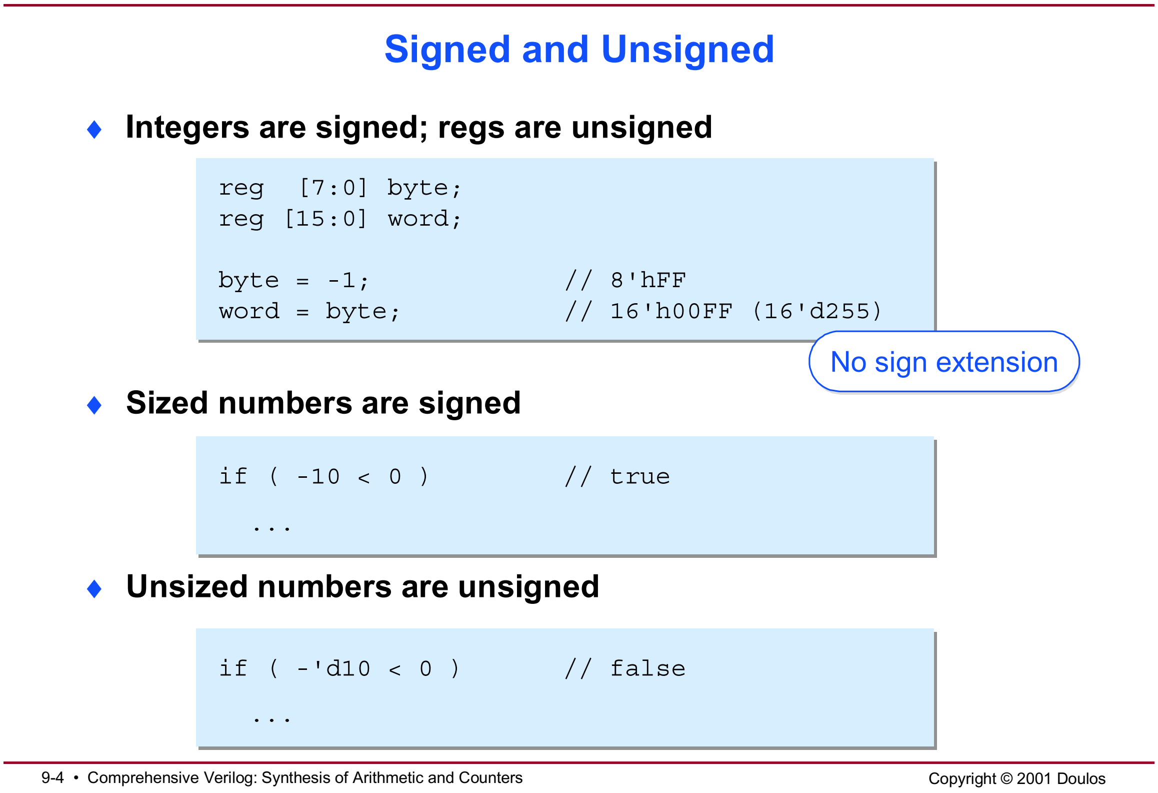
Verilog includes several operators to perform common arithmetic functions.

Generally these operators work as you might expect. Numbers can be added, subtracted, multiplied, divided and the modulus can be taken. There are a couple of things to be aware of though.

When a number overflows, the value "rolls round", as in a hardware register. Therefore adding 1 to 4'b1111 and assigning the result to a 4 bit register gives 4'b0000.

Secondly, sign extension does not automatically take place when using 'negative' (i.e. twos complement) numbers. This is explained on the next slide and again later in this section.

Signed and Unsigned



Signed and Unsigned

- Integers are signed; regs are unsigned

```
reg  [7:0] byte;
reg [15:0] word;

byte = -1;                // 8'hFF
word = byte;              // 16'h00FF (16'd255)
```

No sign extension

- Sized numbers are signed

```
if ( -10 < 0 )            // true
  ...
```

- Unsized numbers are unsigned

```
if ( -'d10 < 0 )          // false
  ...
```

9-4 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Copyright © 2001 Doulos

Notes:

In Verilog, numbers are treated as unsigned quantities, unless the register type integer is used. (An integer is a signed 32 bit number.)

For example, the following assignment to the reg byte

byte = -1;

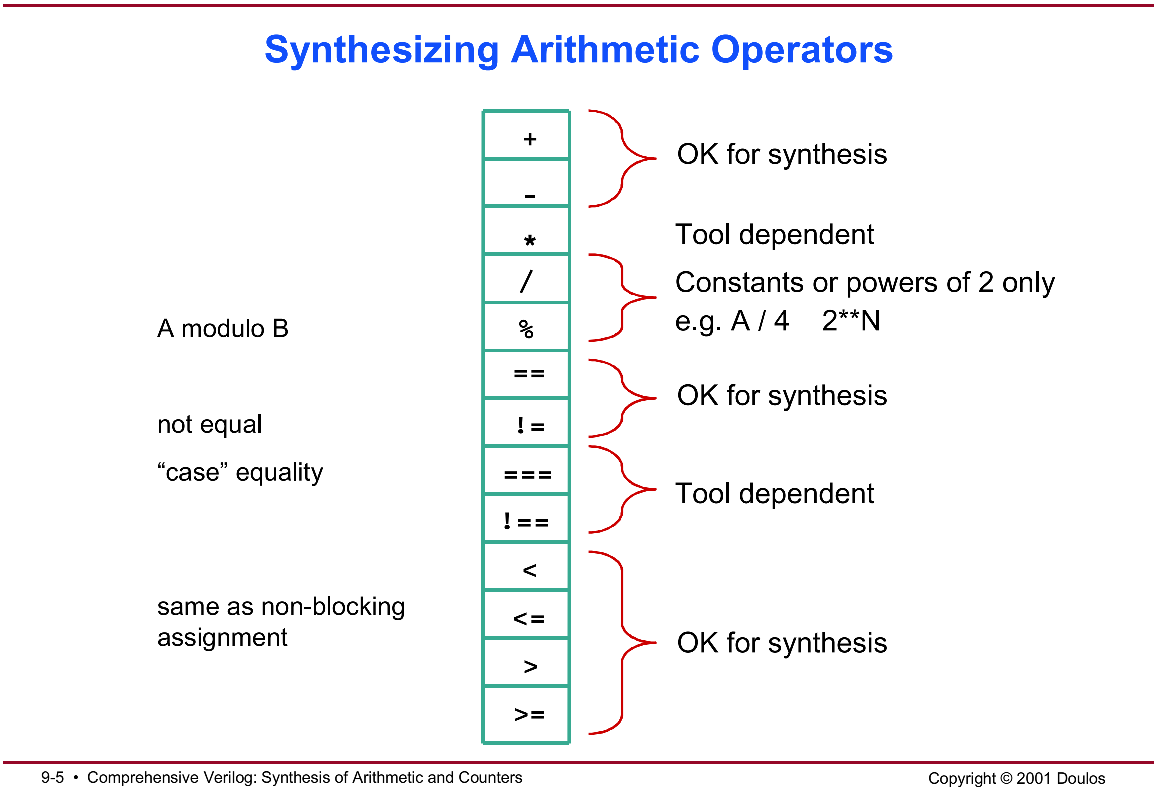
gives byte the value 8'b11111111. Although this is the twos complement representation of -1, the fact that this is a signed number is lost. So when the assignment to the larger reg word is made:

word = byte;

the value of byte is zero extended, and word becomes 16'b0000000011111111, which is clearly not the twos complement representation of -1.

When using numbers in expressions, unsized integers are treated as being signed, but sized numbers as unsigned, even if the size is not specified. So -10 is a signed number, whereas -'d10 is a 32-bit unsigned number, which has a large positive value.

Synthesizing Arithmetic Operators



Synthesizing Arithmetic Operators

	Operator	
	+	OK for synthesis
	-	
	*	Tool dependent
	/	Constants or powers of 2 only e.g. A / 4 2**N
A modulo B	%	
	==	OK for synthesis
not equal	!=	
"case" equality	===	Tool dependent
	!==	
	<	OK for synthesis
same as non-blocking assignment	<=	
	>	
	>=	

9-5 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Notes:

The table above shows how the Verilog arithmetic and comparison operators are synthesized.

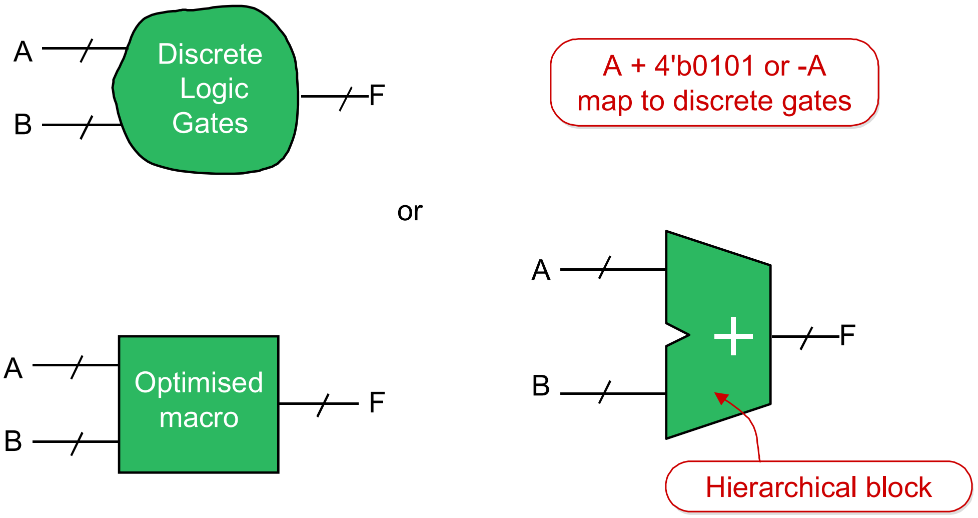
The only generally synthesizable operators are add (+), subtract (-) and comparison (=   != < <= > >=). Some synthesis tools will fully handle multiplication, generating a combinational multiplier. The operators / and % are not synthesizable in general. However, they can be synthesized when the operands are constant, or when used to represent mask and shift operations. This happens when the appropriate operand is a static power of 2 (e.g. A * 2, A / 4, N % 2). Although some synthesis tools might be able to synthesis the case equality operators (=== and !==), these are not recommended - use ordinary equality instead.

You can, and should, use parentheses ( ) to determine the order that the operators within an expression are evaluated.

Arithmetic Operators

Arithmetic Operators

- **What happens when you write F <= A + B?**



A + 4'b0101 or -A map to discrete gates

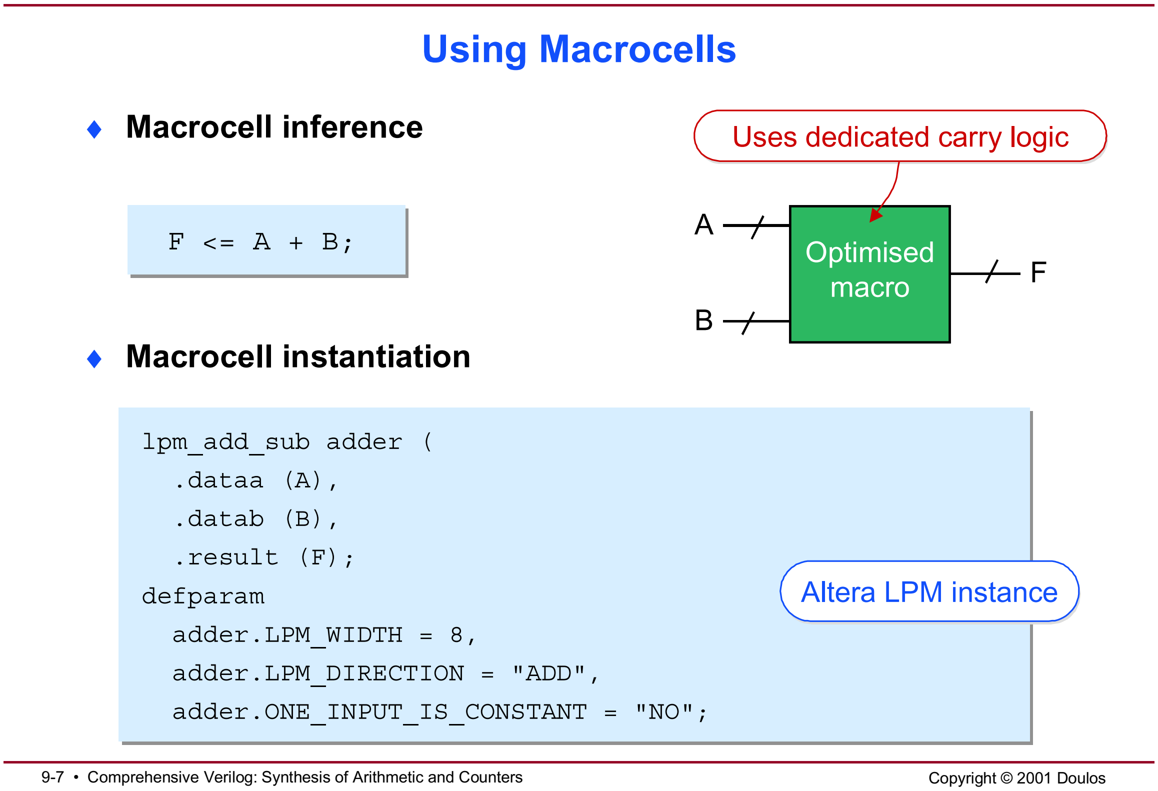
Hierarchical block

Notes:

In practice, what happens to arithmetic operators is dependent on the synthesis tool.

Some tools will map the operators to discrete gates, others will make use of macro-cells optimized for the target technology. Some tools create hierarchical blocks by default, others create flat structures. Most RTL synthesis tools work by having a library of arithmetic parts built in, represented in a technology independent format (e.g. boolean equations), which they then map to the target architecture.

Using Macrocells



Using Macrocells

- **Macrocell inference**

```
F <= A + B;
```

Uses dedicated carry logic
A
B
Optimised macro
F

- **Macrocell instantiation**

```
lpm_add_sub adder (
  .dataa (A),
  .datab (B),
  .result (F);
defparam
  adder.LPM_WIDTH = 8,
  adder.LPM_DIRECTION = "ADD",
  adder.ONE_INPUT_IS_CONSTANT = "NO";
```

Altera LPM instance

9-7 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Copyright © 2001 Doulos

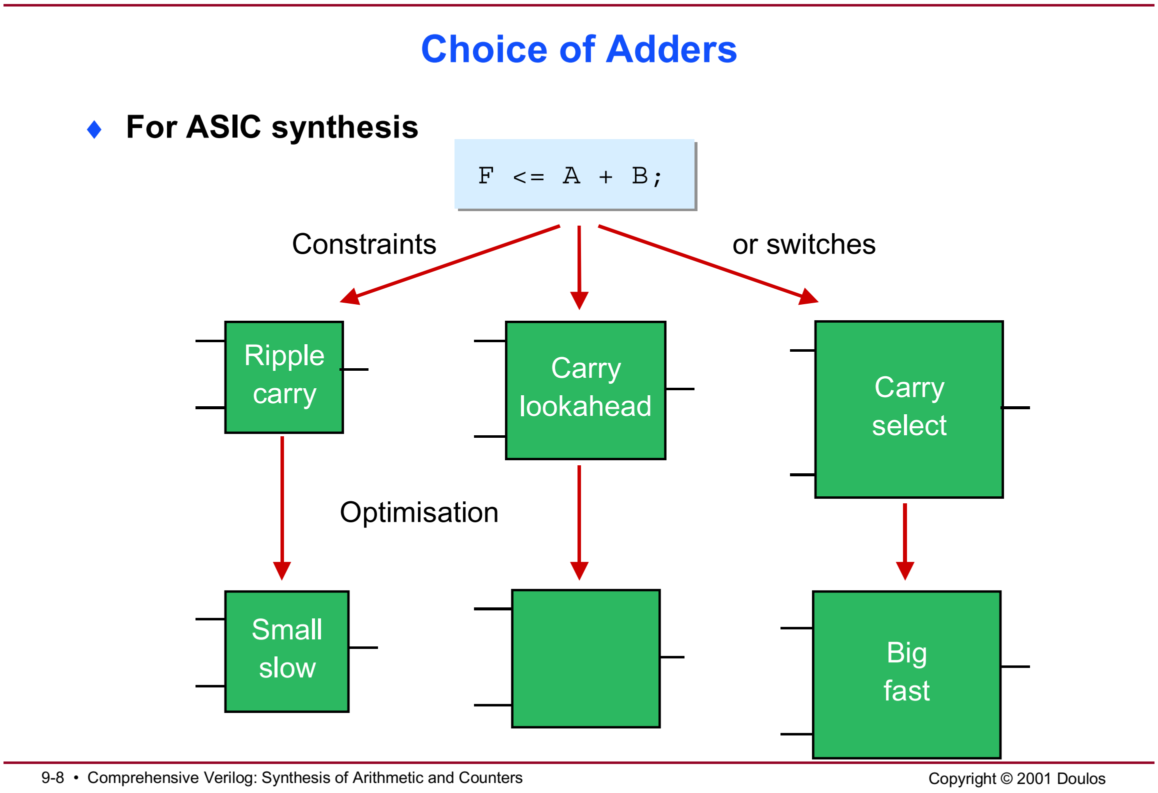
Notes:

RTL synthesis tools can either convert the arithmetic operator to Boolean equations and thence to gates, or else map the operation directly to a macrocell. The use of optimized macro-cells is crucial for FPGA devices in order to achieve good performance and utilization.

If your synthesis tool can't map an arithmetic operator to an optimized macro-cell, you will need to hand instantiate the macro-cell directly in your Verilog code. (Vendors may supply tools to create Verilog code to copy and paste into your design.) This may make your code vendor specific.

Some macrocells, for example decoders, cannot be inferred, so technology specific instantiation is the only option.

Choice of Adders



Choice of Adders

- For ASIC synthesis

```
F <= A + B;
```

Constraints

or switches

Optimisation

9-8 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Notes:

An addition operation can be implemented in hardware using one of several hardware architectures, trading off silicon area for speed. For ASIC design, at one extreme, a ripple carry adder is the smallest and slowest option, whereas the carry select adder is the largest and often the fastest. (For FPGAs that include dedicated carry chains, ripple carry adders are usually the fastest and smallest means to implement adders up to 32 bits.)

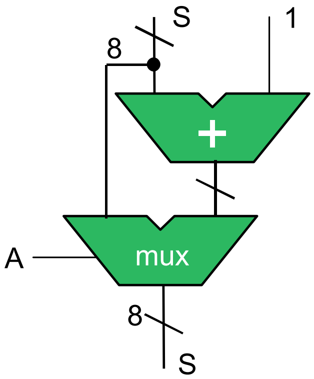
Some ASIC synthesis tools automatically infer the most appropriate architecture using your design constraints, other ASIC synthesis tools allow you to make the selection manually. FPGA synthesis tools typically do not give you a choice, since the adder type is fixed by the macros optimized for the technology.

Arithmetic is not Optimized

Arithmetic Isn't Optimised

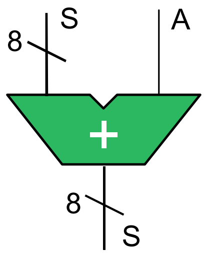
```
reg [7:0] S;
reg A;
```

```
if (A)
  S <= S + 1;
```



```
S <= S + A;
```

Rewritten to reduce area



Notes:

Synthesis tools typically do not optimize arithmetic structures in most situations. Thus, you have to be particularly careful when writing any Verilog code that includes any arithmetic operators.

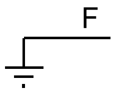
The reason that synthesis tools have problems with optimizing arithmetic structures has to do with the way functions are represented internally within the tool. Most synthesis algorithms represent functions as single level sum-of-products boolean equations, but with this representation, arithmetic functions have a huge number of product terms. This representation severely limits the amount of manipulation and optimization that can be done.

The if statement will be synthesized with an 8 bit incrementer and an 8 bit multiplexer. With most synthesis tools, no amount of fiddling with constraints and synthesis settings will significantly alter the architecture. You must re-write the source code entirely to direct the synthesis tool to the architecture you want, as shown at the bottom.

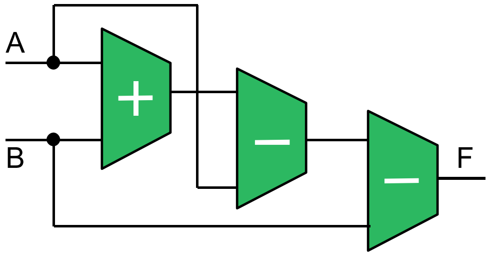
Arithmetic Really isn’t Optimized!

Arithmetic Really Isn’t Optimised!

```
assign F = A & B & !A & !B;
```



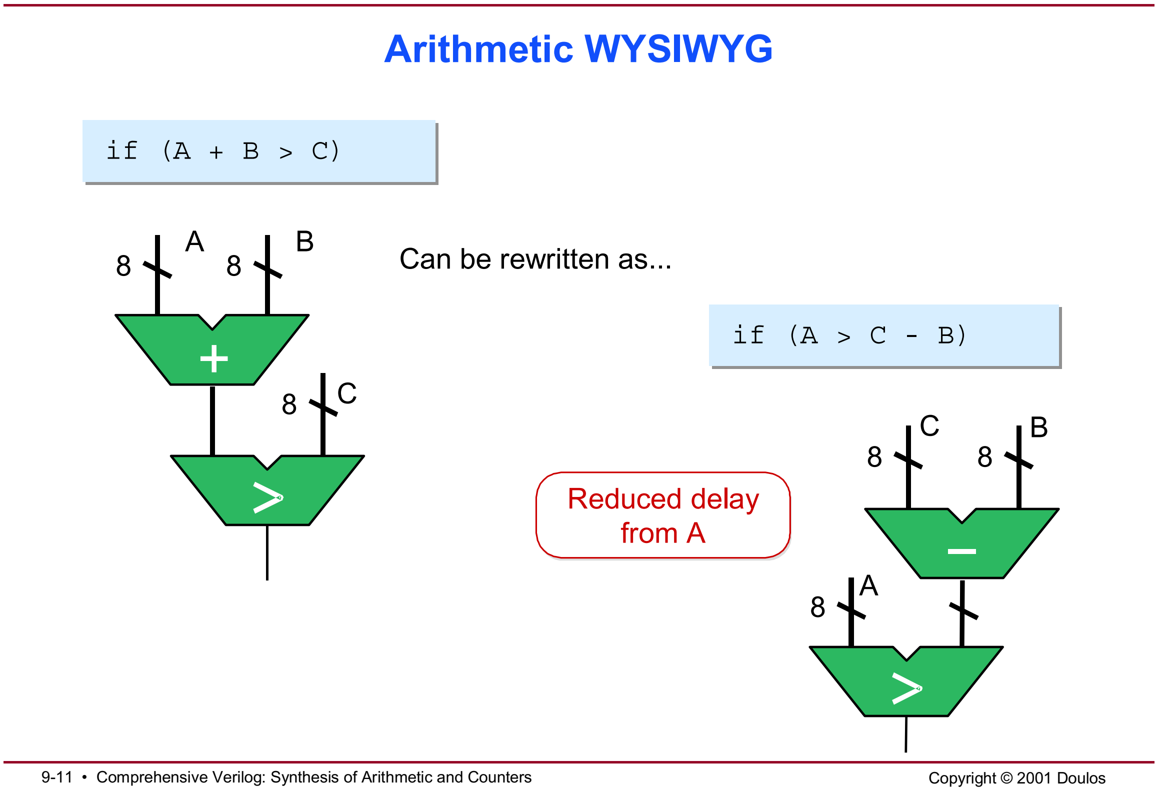
```
assign F = A + B - A - B;
```



Notes:

This example is a classic case of the lack of arithmetic optimization. No matter what you do by way of setting constraints or manipulating hierarchy, most RTL synthesis tools will stubbornly refuse to reduce the arithmetic circuit, whereas the boolean equation is effortlessly minimized to a bit of wire.

Arithmetic WYSIWYG



Arithmetic WYSIWYG

```
if (A + B > C)
```

Can be rewritten as...

```
if (A > C - B)
```

Reduced delay from A

9-11 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

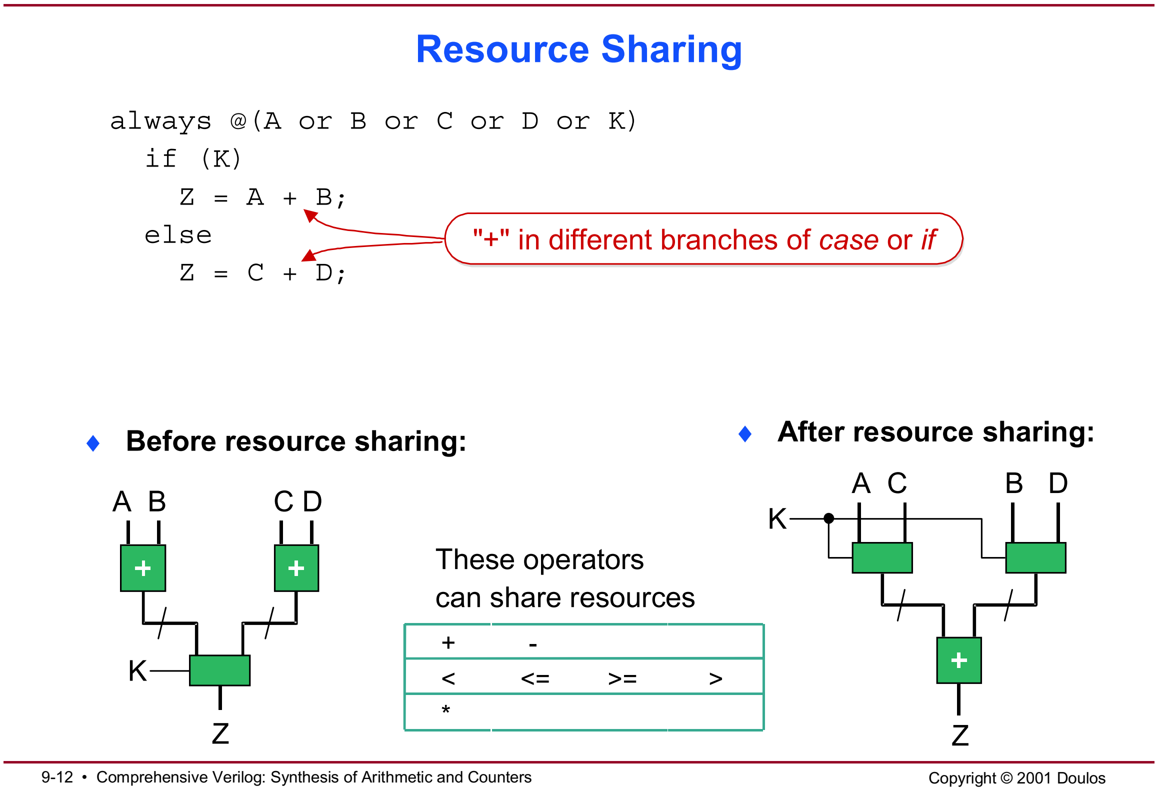
Notes:

Unlike the synthesis of boolean equations, with arithmetic what you see is what you get! Thus you have to think carefully about the hardware architecture you want when using arithmetic operators.

For example, if we write A + B > C, then the addition is done before the comparison, and the delay from A or B to the output is longer than the delay from C, which is fine if C is on the critical path. If we re-write the equation as

A > C - B, then the delay from A is the shorter, which is better if A is on the critical path.

Resource Sharing



Resource Sharing

```
always @(A or B or C or D or K)
  if (K)
    Z = A + B;
  else
    Z = C + D;
```

"+" in different branches of *case* or *if*

- **Before resource sharing:**

- **After resource sharing:**

These operators can share resources

+	-		
<	<=	>=	>
*			

9-12 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

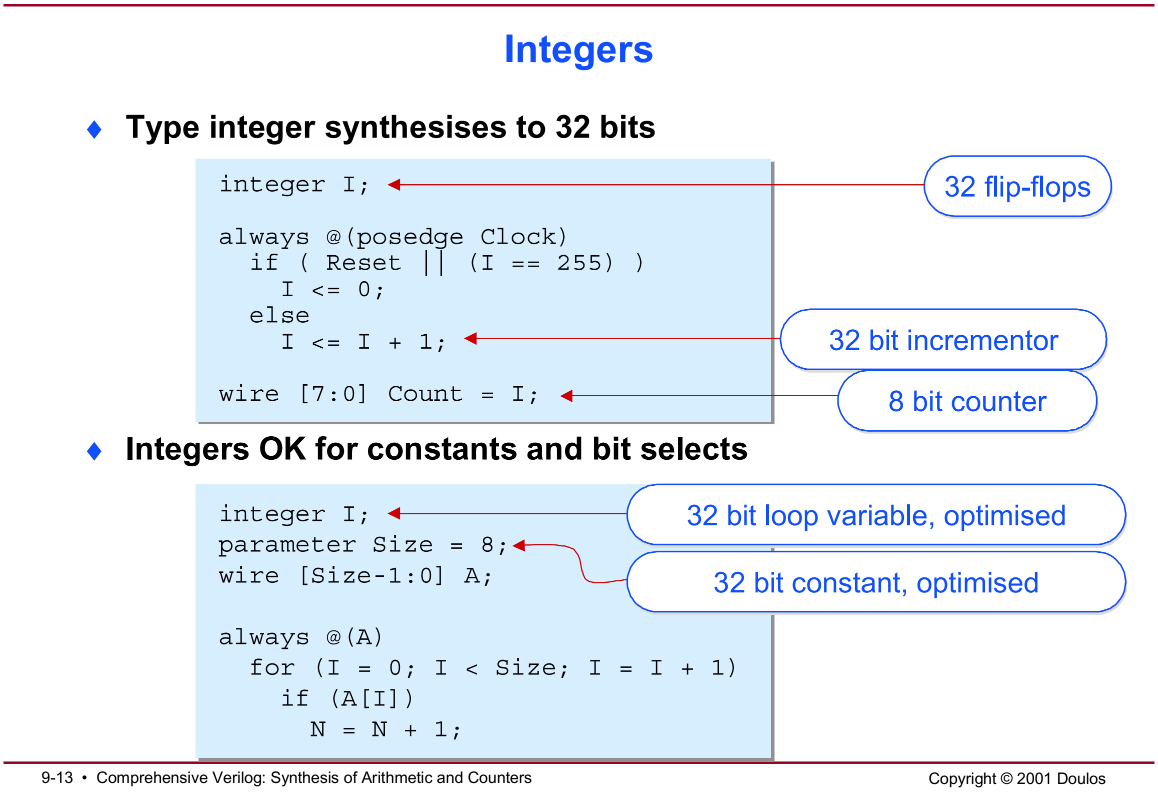
Copyright © 2001 Doulos

Notes:

Resource sharing is a high level optimization, necessary because of the limited power of boolean optimization techniques to deal effectively with arithmetic functions. Resource sharing endeavours to share a single hardware resource (e.g. an adder) for several different operations, and can do so only if the operations are executed in exclusive control paths in a case or if statement in the Verilog HDL source code. In the example opposite, the two "+" operations in the branches of the if statement can share a single adder, with additional multiplexing necessary on the inputs.

Most synthesis tools provide a means of manually controlling resource sharing. This may be done by means of directives in the Verilog HDL source code, or through a graphical user interface to a schematic representation of the synthesized design.

Integers



Integers

- Type integer synthesises to 32 bits

```
integer I;

always @(posedge Clock)
  if ( Reset || (I == 255) )
    I <= 0;
  else
    I <= I + 1;

wire [7:0] Count = I;
```

32 flip-flops

32 bit incrementor

8 bit counter

- Integers OK for constants and bit selects

```
integer I;
parameter Size = 8;
wire [Size-1:0] A;

always @(A)
  for (I = 0; I < Size; I = I + 1)
    if (A[I])
      N = N + 1;
```

32 bit loop variable, optimised

32 bit constant, optimised

9-13 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

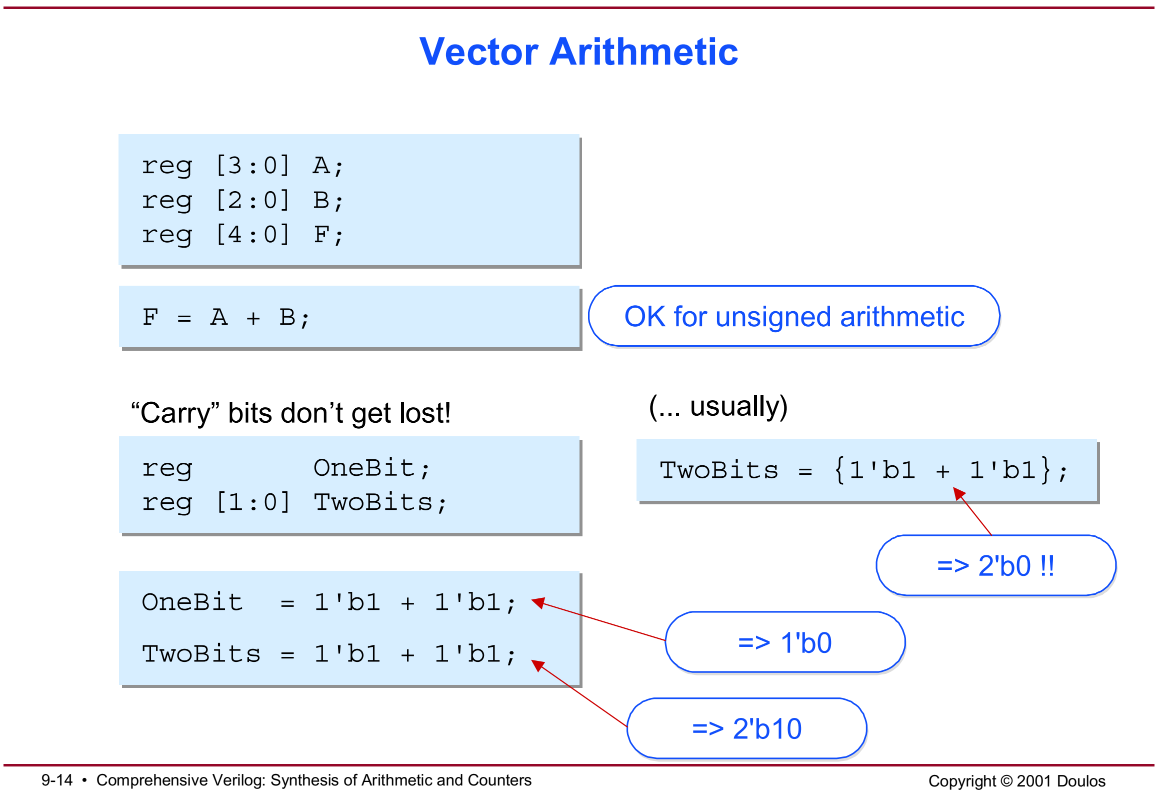
Notes:

integer

An integer in Verilog is a register type that represents 32 bit signed numbers. So using an integer to infer flip-flops will result in 32 flip-flops being inferred!

Some, but not all, synthesis tools may be able to remove any redundant flip-flops from the design. Therefore, integers should generally be avoided for synthesis. It is better to use regs and explicitly define the width (e.g. reg [15:0] R;). The exception is loop variables, parameters, and vector indexes, where integers may be used safely without the risk of 32 bit busses.

Vector Arithmetic



Vector Arithmetic

```
reg [3:0] A;
reg [2:0] B;
reg [4:0] F;
```

```
F = A + B;
```

OK for unsigned arithmetic

"Carry" bits don't get lost!

```
reg       OneBit;
reg [1:0] TwoBits;
```

```
OneBit  = 1'b1 + 1'b1;

TwoBits = 1'b1 + 1'b1;
```

=> 1'b0

=> 2'b10

(... usually)

```
TwoBits = {1'b1 + 1'b1};
```

=> 2'b0 !!

9-14 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Copyright © 2001 Doulos

Notes:

Vectors in Verilog are treated as unsigned quantities. If the regs in a design are indeed storing unsigned values, then regs of different widths can be added and subtracted with no special treatment.

Note in particular, that if the reg on the left hand side of an assignment is big enough, any carry or overflow bits generated by the expression on the right hand side are not lost. The same also applies to wires in continuous assignments.

Sometimes, strange things happen, though. In the assignment

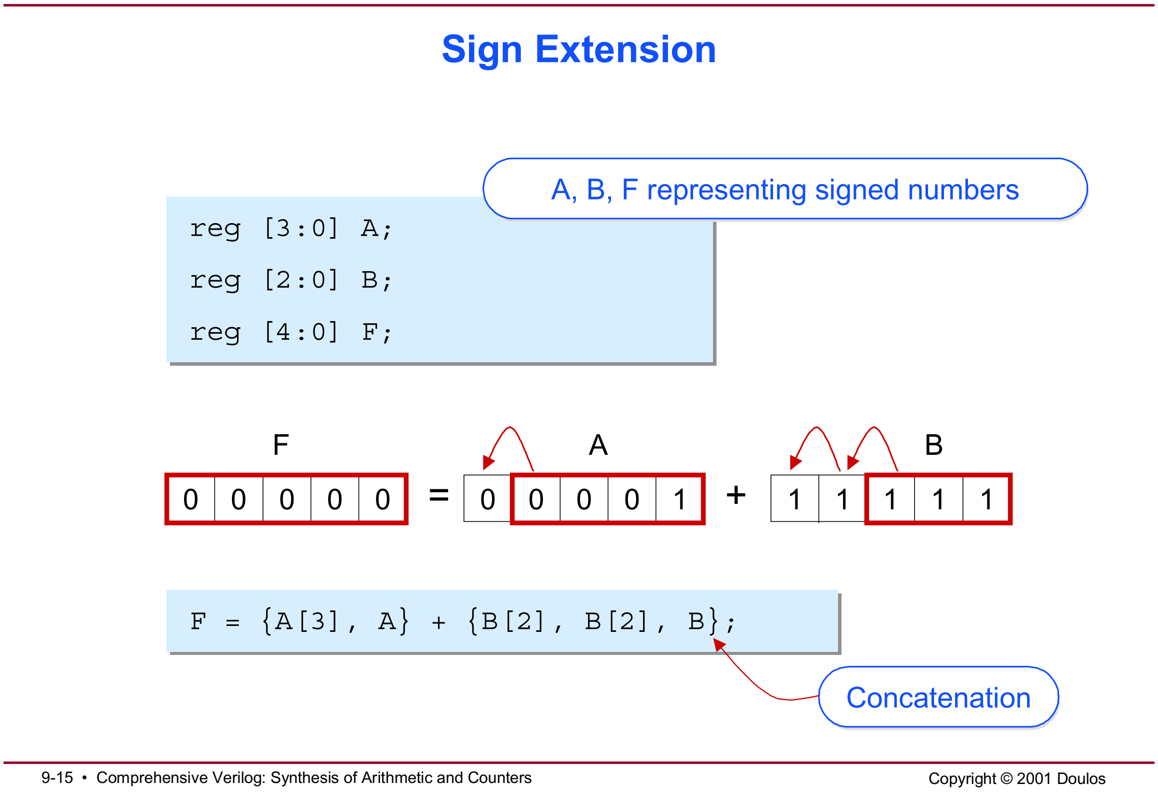
```
TwoBits = { 1'b1 + 1'b1 };
```

the concatenation "hides" the size of the target of the assignment (TwoBits), resulting in the carry bit being lost!

The following assignment:

```
TwoBits = { 2'b01 + 2'b01 };
does give TwoBits = 2'b10.
```

Sign Extension



Sign Extension

A, B, F representing signed numbers

```
reg [3:0] A;
reg [2:0] B;
reg [4:0] F;
```

```
F = {A[3], A} + {B[2], B[2], B};
```

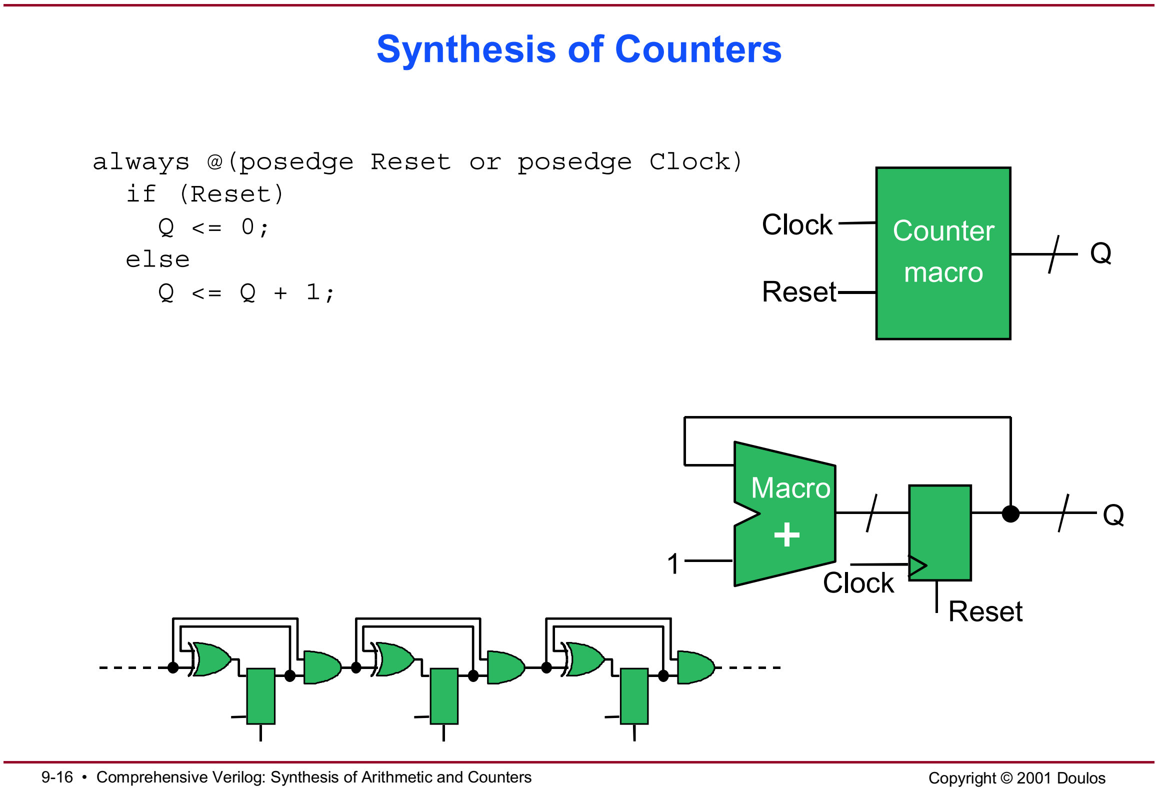
Concatenation

9-15 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Notes:

Suppose that we are trying to store signed (i.e. twos complement) numbers in regs. In order to generate the correct result, we must sign extend the operands so that they are both the same size as the result. This can be done using the concatenation operator, as shown.

Synthesis of Counters



Synthesis of Counters

```
always @(posedge Reset or posedge Clock)
  if (Reset)
    Q <= 0;
  else
    Q <= Q + 1;
```

9-16 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Copyright © 2001 Doulos

Notes:

A counter can be described in RTL Verilog code by performing the operation +1 on a clock edge, as shown in the example opposite. The + operator will synthesis to an adder with one of the inputs tied to arithmetic 1, and thus the always statement will synthesis to a synchronous binary counter with an asynchronous reset. It is easy to add variations such as a synchronous reset, parallel load, enable, or up-down counter.

FPGA Synthesis

Many FPGA technology vendors supply macros to allow the efficient implementation of arithmetic and counter structures. Certain Verilog synthesis tools will infer a counter macro when the function of the Verilog source code matches the function of an available macro. Some synthesis tools will infer an adder macro from the "+1" function, whereas other synthesis tools will not infer an adder when one of the operands is a constant.

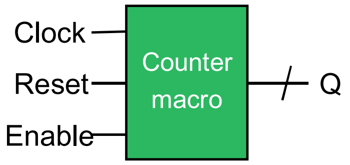
ASIC Synthesis

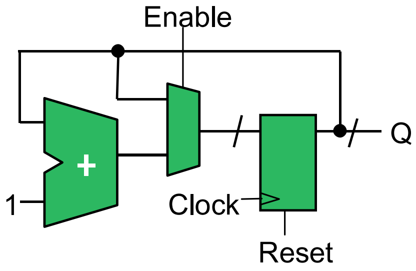
Most ASIC synthesis tools allow you to direct the synthesis of the "+" operator to use either a predefined adder component or to expand the operation into boolean logic gates. When this is done to the synchronous binary counter, we get a conventional implementation as shown at the bottom. Note the regular carry chain of AND gates. We can force the synthesis tool to re-implement the logic by setting a clock constraint. Optimization will re-structure the carry generation to speed up the logic, at the expense of increased area.

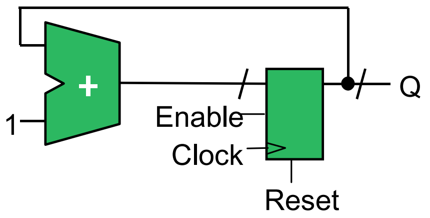
Clock Enables

Clock Enables

```
always @(posedge Reset or posedge Clock)
  if (Reset)
    Q <= 0;
  else if (Enable)
    Q <= Q + 1;
```







Notes:

Synthesis tools generally choose synchronous implementation. For example, an enable on a synchronous counter will be implemented as a synchronous enable, with the count recirculated when the counter is disabled.

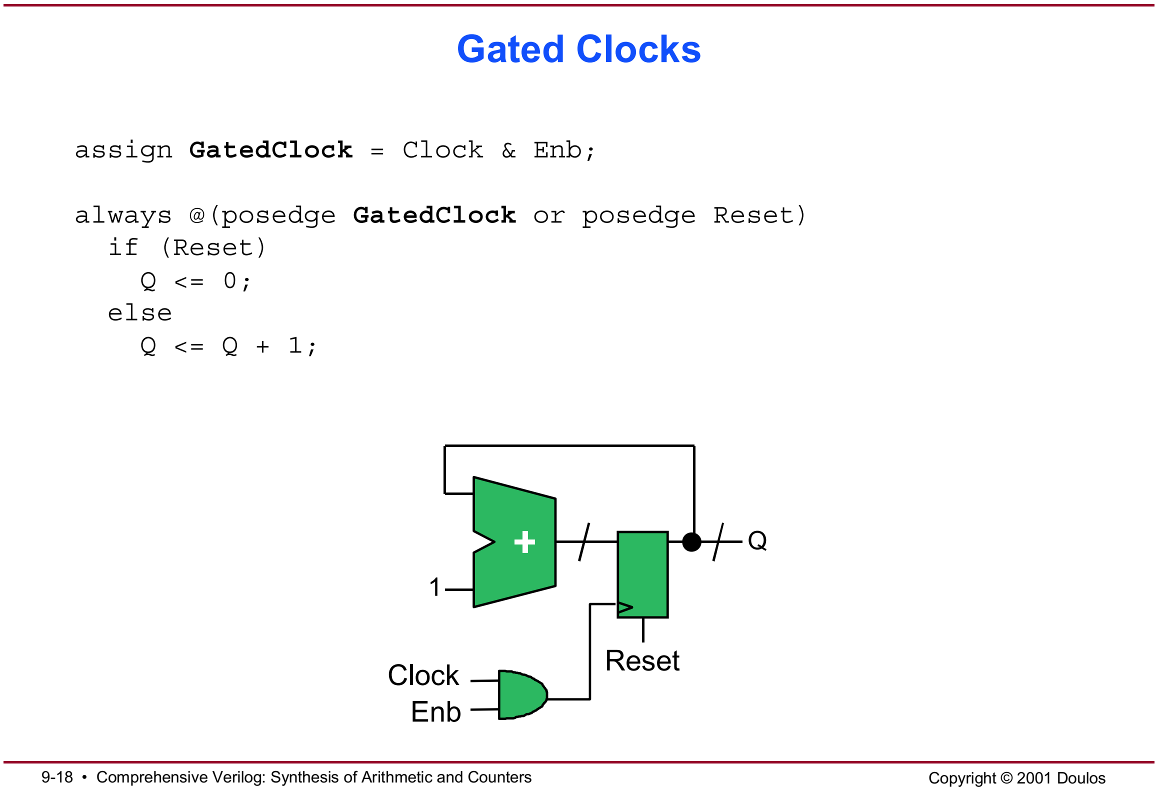
Some synthesis tools, particularly those tuned to synthesis particular FPGA architectures, will make use of the dedicated clock enables built into the technology. To exploit such features, the Verilog code must reflect exactly the behavior of the dedicated hardware. This means making sure that the levels and the priorities of the control inputs match the data sheet. If the levels and priorities are wrong, then the dedicated clock enables will not be used, and extra logic blocks will be consumed to implement the function.

Synthesis tools are more likely to use dedicated clock enables if clocked always statements are written to conform to the following template:

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    // Reset actions
  else if (Enable)
    // Synchronous actions
```

Only the principles are being explained here; the details will depend on the technology and tools that you are using. Check out the details in your synthesis tool manual and technology data sheets.

Gated Clocks



Gated Clocks

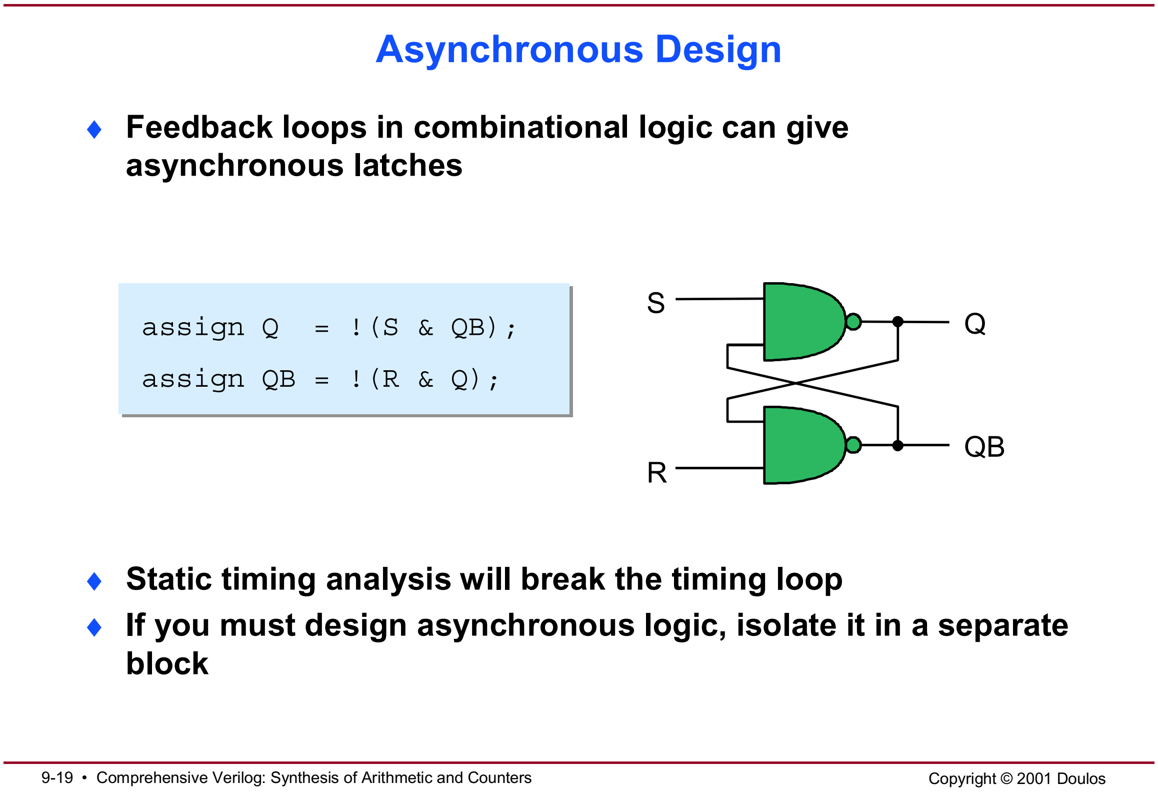
```
assign GatedClock = Clock & Enb;

always @(posedge GatedClock or posedge Reset)
  if (Reset)
    Q <= 0;
  else
    Q <= Q + 1;
```

Notes:

If you want to be sure of synthesizing gated clocks, then they must be coded explicitly, as shown in the top example opposite. Of course, you are responsible for avoiding timing problems caused by gating the clock, such as glitches on the gated clock and clock skew.

This sort of trick is sometimes regarded as being bad design practice. On the other hand, it can be a very effective way of reducing power consumption. In any case, it is best avoided when using FPGA and CPLD technologies; use the dedicated clock enable instead.

Asynchronous Design



Asynchronous Design

- Feedback loops in combinational logic can give asynchronous latches

```
assign Q  = !(S & QB);
assign QB = !(R & Q);
```

- Static timing analysis will break the timing loop
- If you must design asynchronous logic, isolate it in a separate block

9-19 • Comprehensive Verilog: Synthesis of Arithmetic and Counters

Copyright © 2001 Doulos

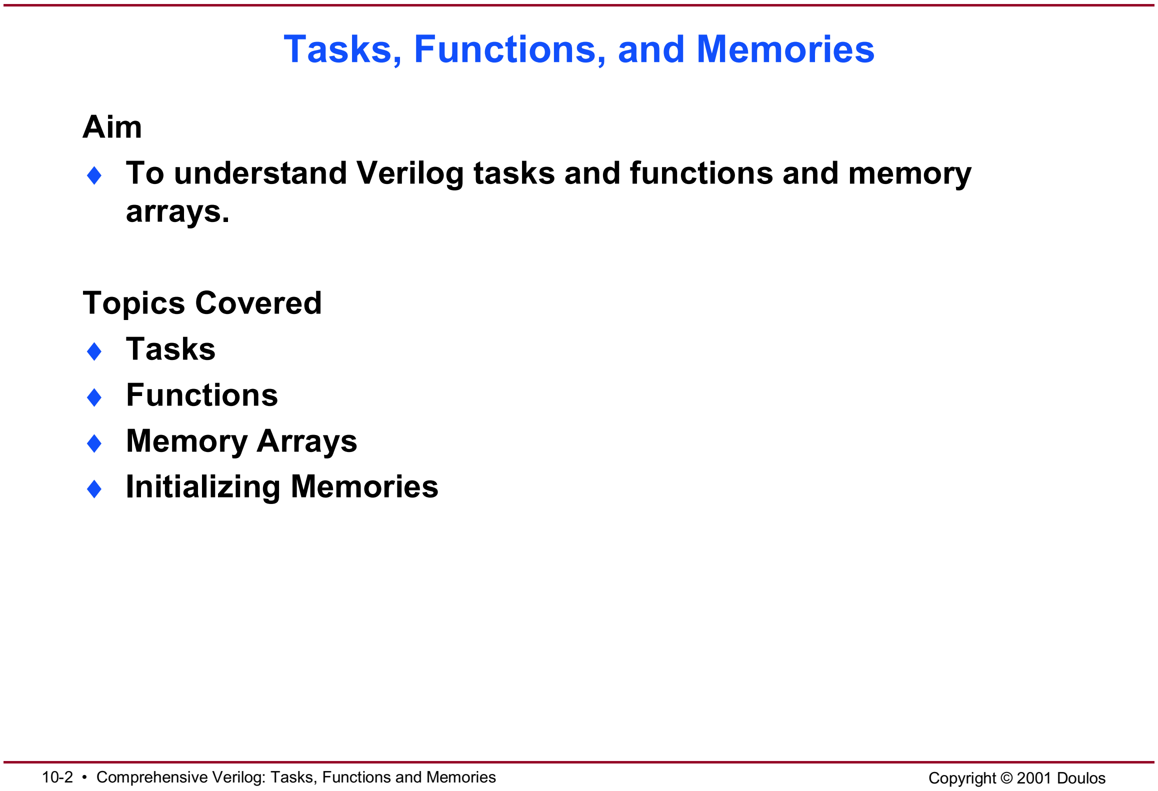
Notes:

Synthesis permits asynchronous design. Inherently asynchronous Verilog will synthesis to asynchronous hardware. However, asynchronous designs require careful manual verification to ensure their correctness. The synthesis tool gives no guarantee that the synthesized hardware will function correctly, because it does not understand the asynchronous timing relationships in the design. The responsibility for identifying, fixing and verifying problems associated with asynchronous design lies entirely with you!

It is best practice to partition your design into separate synchronous blocks, where each block has a single clock. You should then include explicit synchronizers to avoid metastability problems at the block interfaces. Asynchronous logic should be isolated within separate blocks, and tested thoroughly.

Module 10
Tasks, Functions, and Memories

Tasks, Functions, and Memories



Tasks, Functions, and Memories

Aim

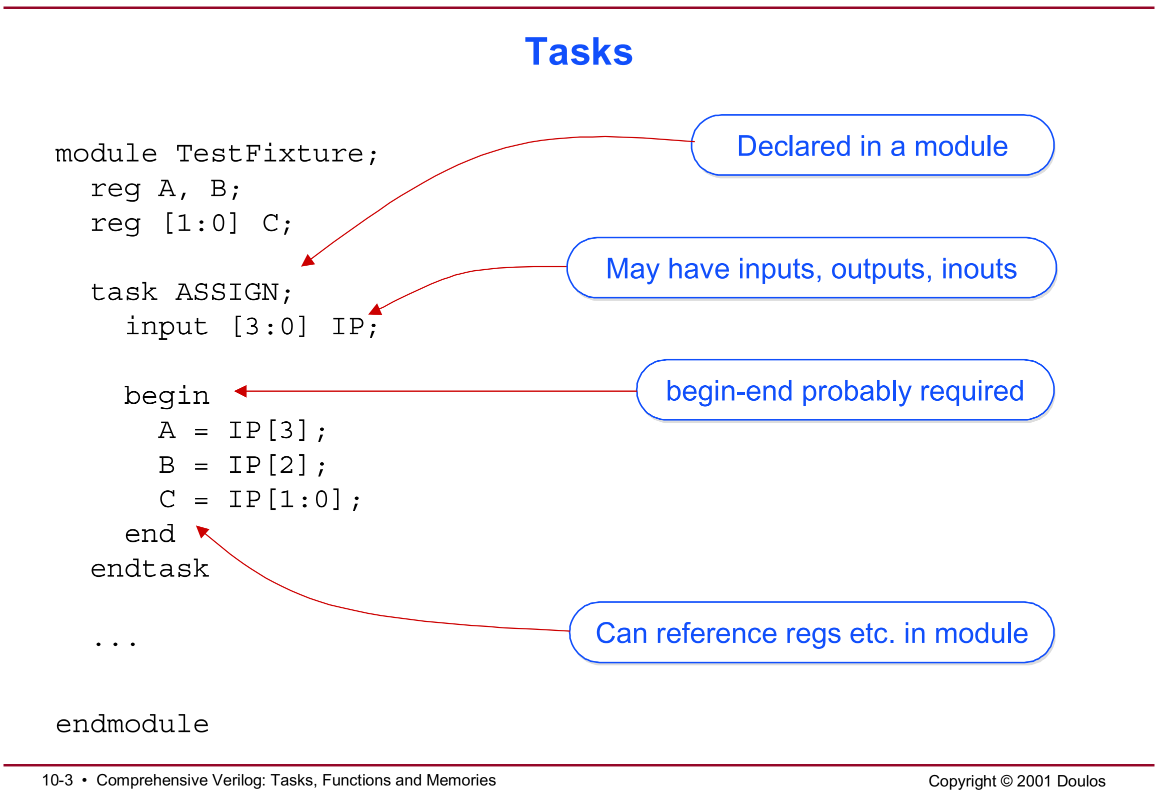
- To understand Verilog tasks and functions and memory arrays.

Topics Covered

- Tasks
- Functions
- Memory Arrays
- Initializing Memories

Notes:

Tasks



Tasks

```
module TestFixture;
  reg A, B;
  reg [1:0] C;

  task ASSIGN;
    input [3:0] IP;

    begin
      A = IP[3];
      B = IP[2];
      C = IP[1:0];
    end
  endtask

  ...


endmodule
```

Declared in a module

May have inputs, outputs, inouts

begin-end probably required

Can reference regs etc. in module

10-3 • Comprehensive Verilog: Tasks, Functions and Memories

Copyright © 2001 Doulos

Notes:

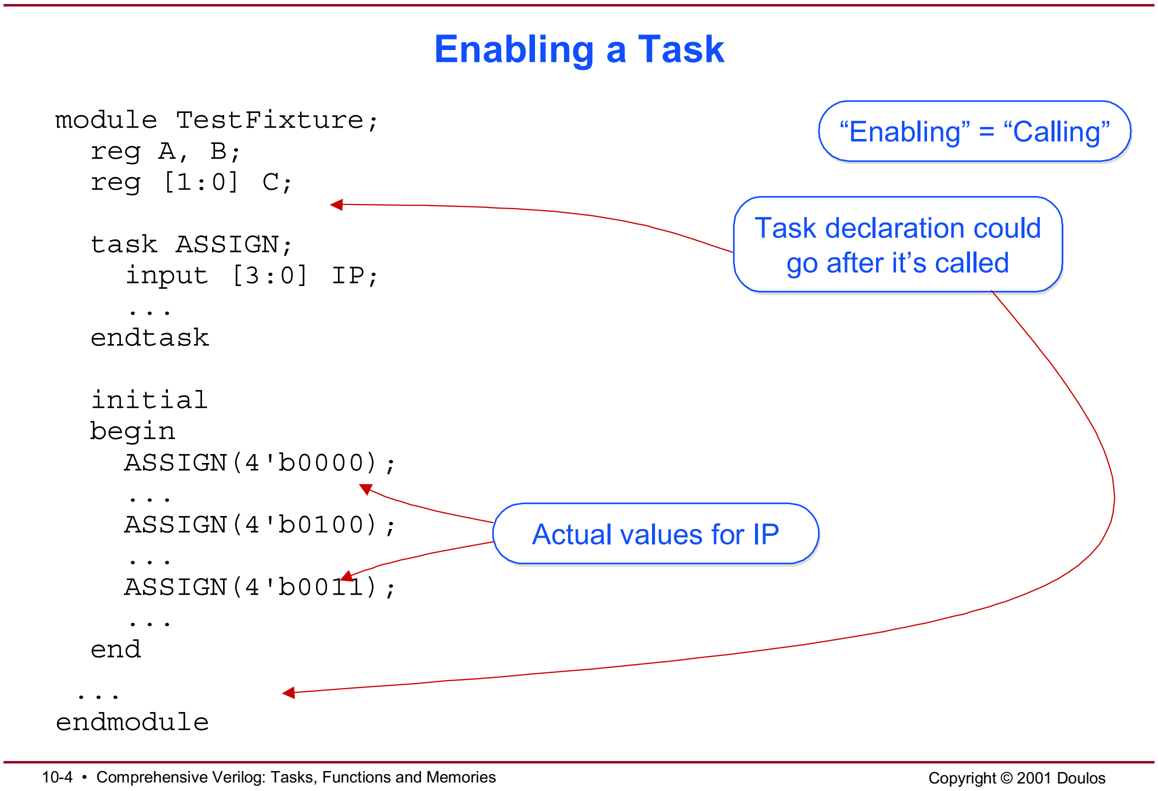
Tasks are used to partition always statements into smaller, hierarchical sequences of executable code. They are declared within modules.

A task consists of a name, an optional set of arguments (inputs, outputs and inouts) and declarations, and a single statement, or sequence of statements surrounded by begin ... end.

Note that the syntax for a task does not include a bracketed list of the task's arguments after the task's name. This is different from the syntax for module port declarations.

A Task may reference the regs and wires in the module in which the task is declared using their simple names: values don't have to be passed through task arguments. New registers may also be declared within the task. In the example opposite, the task uses the regs A, B, and C that are declared in the module TestFixture.

Enabling a Task



Enabling a Task

```
module TestFixture;
  reg A, B;
  reg [1:0] C;

  task ASSIGN;
    input [3:0] IP;
    ...
  endtask

  initial
  begin
    ASSIGN(4'b0000);
    ...
    ASSIGN(4'b0100);
    ...
    ASSIGN(4'b0011);
    ...
  end

 ...
endmodule
```

"Enabling" = "Calling"

Task declaration could go after it's called

Actual values for IP

10-4 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

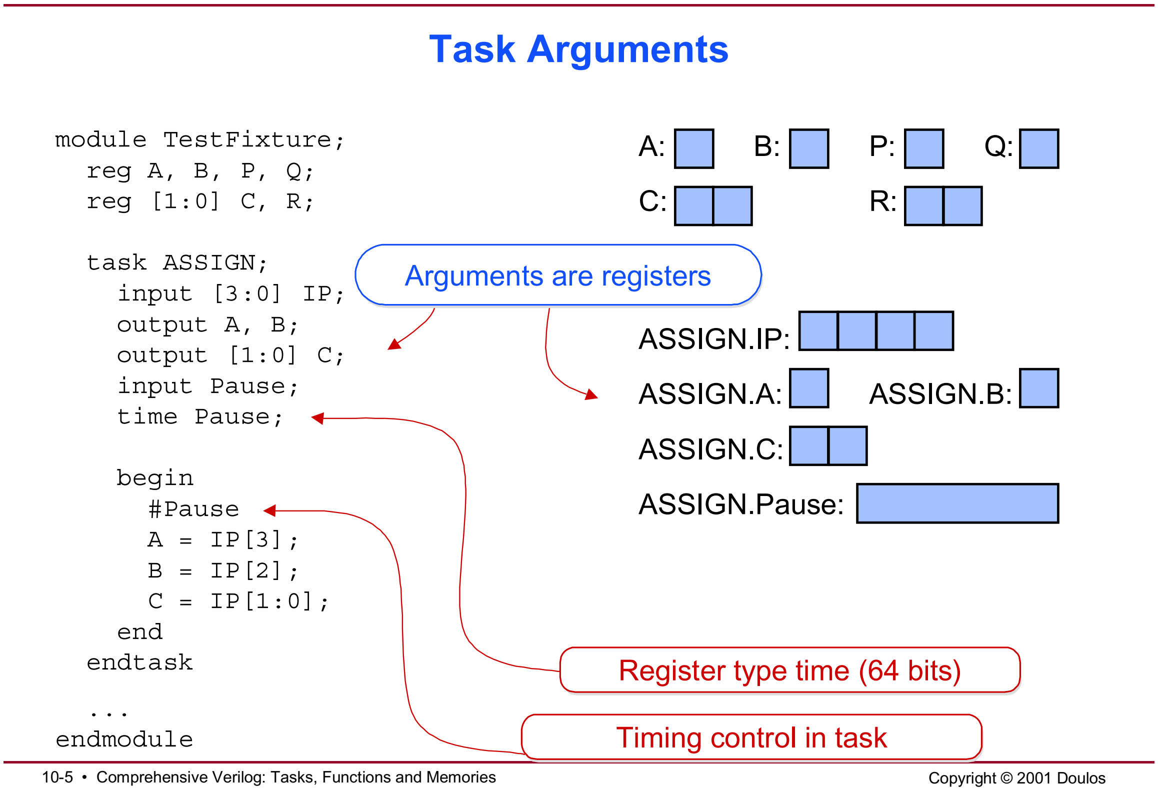
A task is enabled (i.e. called) by giving its name, and supplying a list of values and/or regs for the arguments. Tasks may be enabled from initial or always statements, or even from within other tasks.

When a task is enabled, the statement(s) in the task are executed; the flow of control then returns to the place from where the task was enabled.

In this example, the initial statement causes the task to be enabled three times. The first time, IP is given the value 4'b0000 and so A, B and C are all set to 0. The second time the task is enabled, B will be set to 1, A and C to 0. The final task enable causes A and B to be set to 0, and C to 2'b11.

Note that the declaration of a task need not appear before the task is enabled. So in this example, the task declaration could be the last thing in the module. The task could also be declared in another module. If that module is on an upwards hierarchical path from the module where the task is enabled, the task's simple name (ASSIGN) may be used, and the definition of ASSIGN will be found by the upwards name referencing rules.

Task Arguments



Task Arguments

```
module TestFixture;
  reg A, B, P, Q;
  reg [1:0] C, R;

  task ASSIGN;
    input [3:0] IP;
    output A, B;
    output [1:0] C;
    input Pause;
    time Pause;

    begin
      #Pause
      A = IP[3];
      B = IP[2];
      C = IP[1:0];
    end
  endtask

  ...
endmodule
```

Arguments are registers

Register type time (64 bits)

Timing control in task

10-5 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

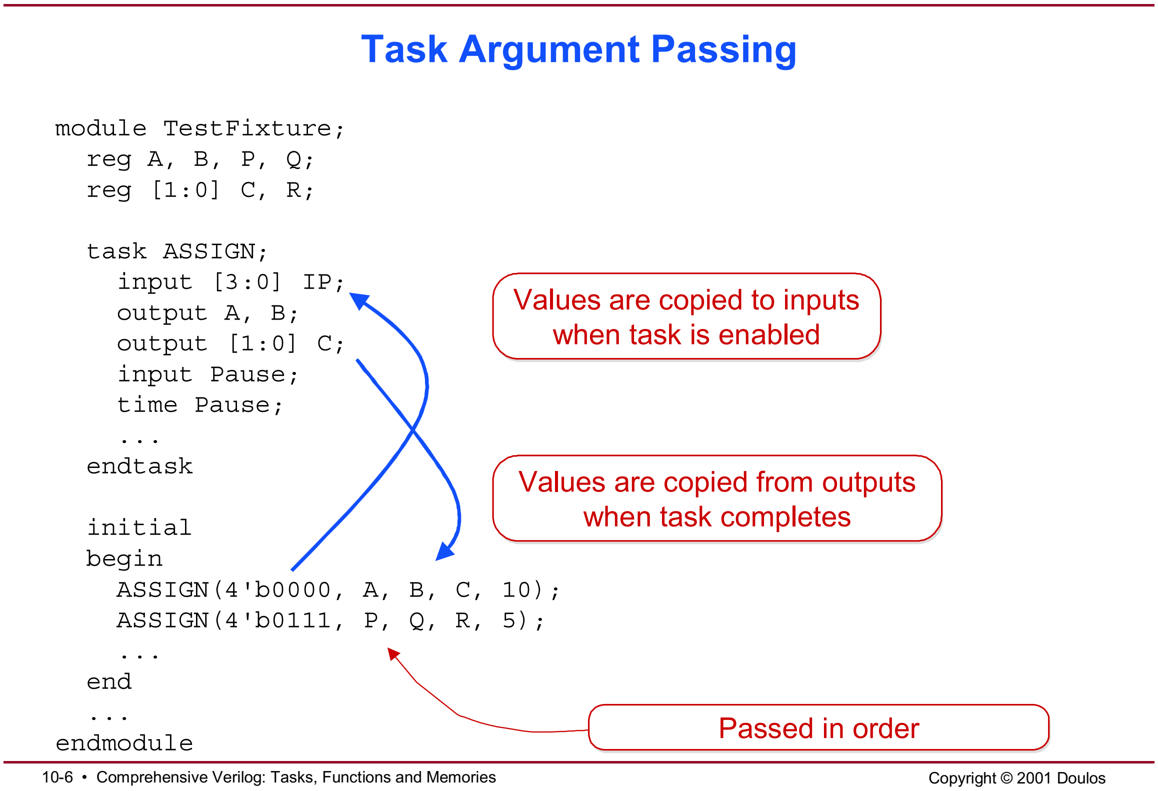
Tasks can have input, output and inout arguments. These are declared using the appropriate keyword; however, they do not appear in a list after the task name like module ports.

All the arguments are in fact registers, local to the task. The values for the input and inout arguments are copied to the respective registers when the task is enabled. Output and inout values are copied from the respective registers when the task completes.

A task may contain any procedural statements, including timing controls. (i.e. any statements that are allowed in initial or always statements.)

The type time, which is mentioned here, is another of Verilog register types (we have already met the reg and integer register types). time registers store 64 bit unsigned values; this is how delays and simulation times are represented in Verilog.

Task Argument Passing



Task Argument Passing

```
module TestFixture;
  reg A, B, P, Q;
  reg [1:0] C, R;

  task ASSIGN;
    input  [3:0] IP;
    output A, B;
    output  [1:0] C;
    input Pause;
    time Pause;
    ...
  endtask

  initial
  begin
    ASSIGN(4'b0000, A, B, C, 10);
    ASSIGN(4'b0111, P, Q, R, 5);
    ...
  end
  ...
endmodule
```

Values are copied to inputs when task is enabled

Values are copied from outputs when task completes

Passed in order

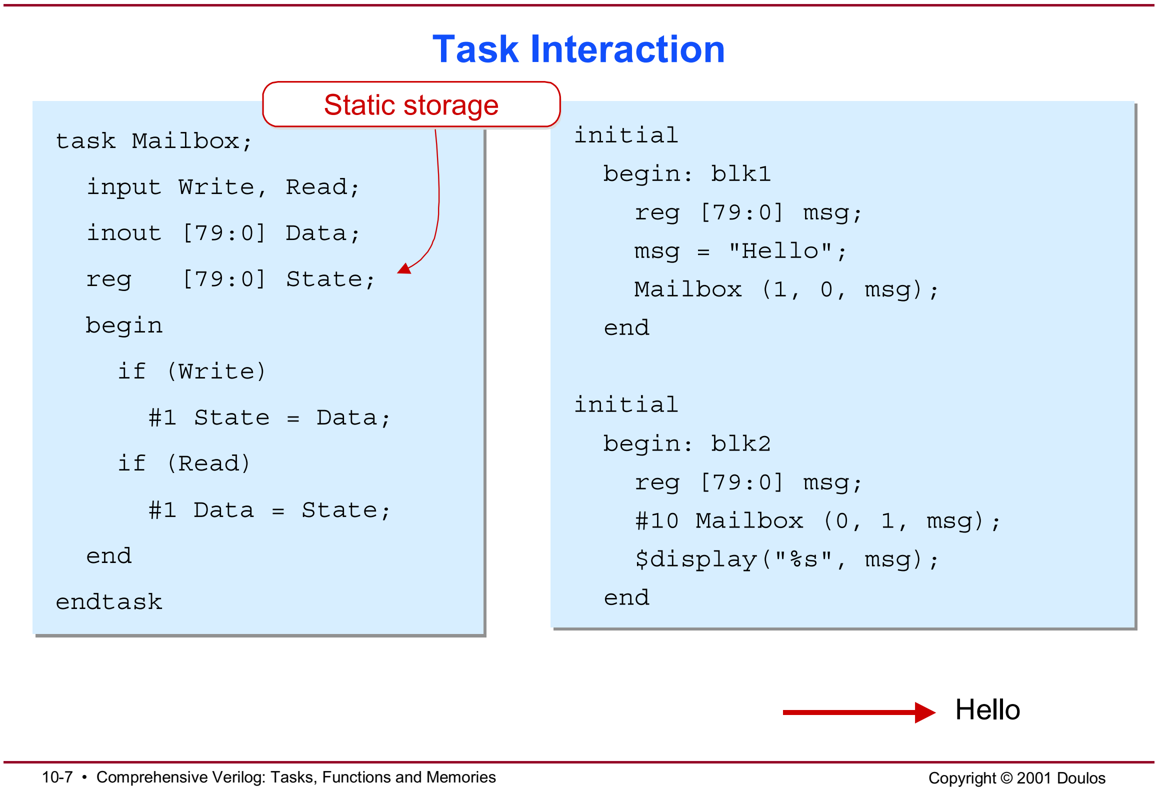
10-6 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

When a task is enabled, values or expressions for the task's inputs and outputs are listed in the order in which they were declared in the task. (Named mapping cannot be used when enabling tasks.)

It follows from the fact that a task's output and inout values are copied out when the task completes, that appropriate register names are required for the outputs and inouts when the task is enabled. In this example, A, B, C, P, Q, and R must be regs.

Task Interaction



Task Interaction

Static storage

```
task Mailbox;
  input Write, Read;
  inout [79:0] Data;
  reg   [79:0] State;
  begin
    if (Write)
      #1 State = Data;
    if (Read)
      #1 Data = State;
  end
endtask
```

```
initial
  begin: blk1
    reg [79:0] msg;
    msg = "Hello";
    Mailbox (1, 0, msg);
  end

initial
  begin: blk2
    reg [79:0] msg;
    #10 Mailbox (0, 1, msg);
    $display("%s", msg);
  end
```

→ Hello

10-7 • Comprehensive Verilog: Tasks, Functions and Memories

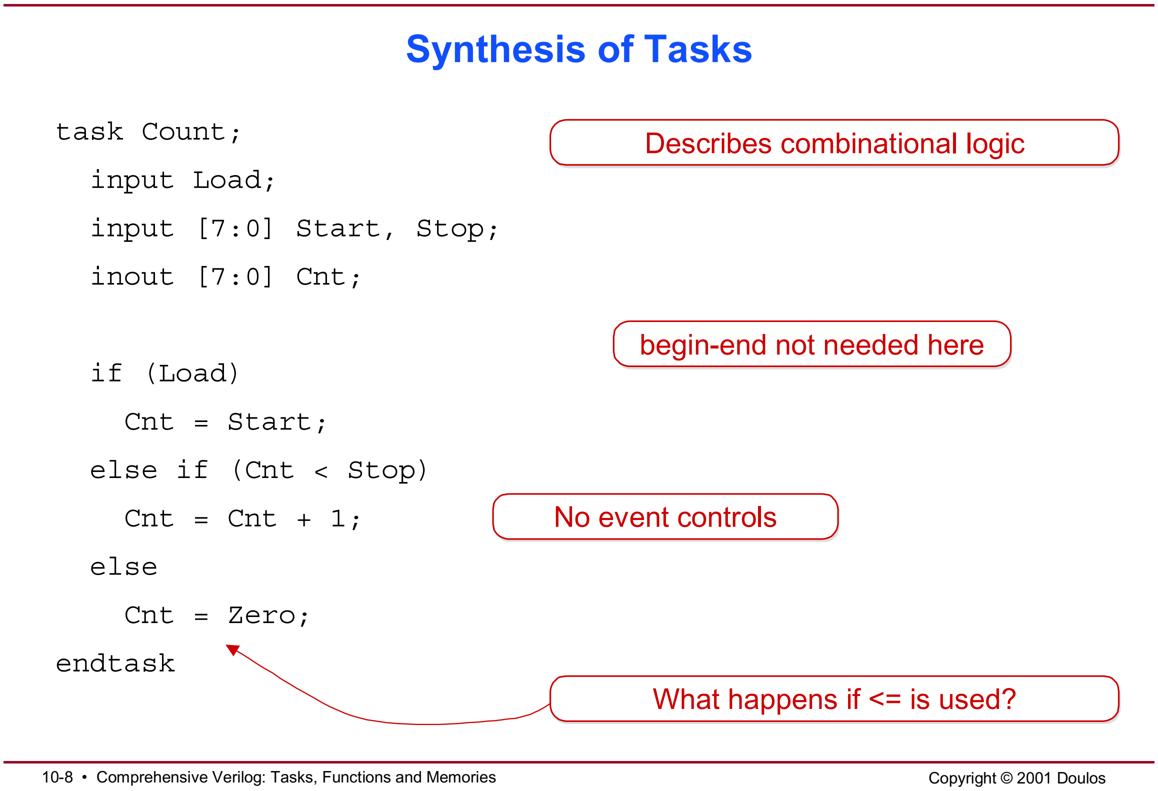
Copyright © 2001 Doulos

Notes:

Any registers defined inside a task (including the task arguments) are stored statically i.e. the same storage is shared by all calls to the task. This can be used as a positive feature as shown in the example opposite, but most of the time it is an unwanted feature and can cause a lot of problems.

Generally, if you have a task containing timing controls and internal storage (including arguments), you should be careful not to make concurrent calls to that task.

Synthesis of Tasks



Synthesis of Tasks

```
task Count;
  input Load;
  input [7:0] Start, Stop;
  inout [7:0] Cnt;

  if (Load)
    Cnt = Start;
  else if (Cnt < Stop)
    Cnt = Cnt + 1;
  else
    Cnt = Zero;
endtask
```

Describes combinational logic

begin-end not needed here

No event controls

What happens if <= is used?

10-8 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

This example shows a task being used to "factor out" code that would otherwise have to be repeated in each of two procedural blocks. This makes the code shorter and easier to modify. The next page shows the task being enabled.

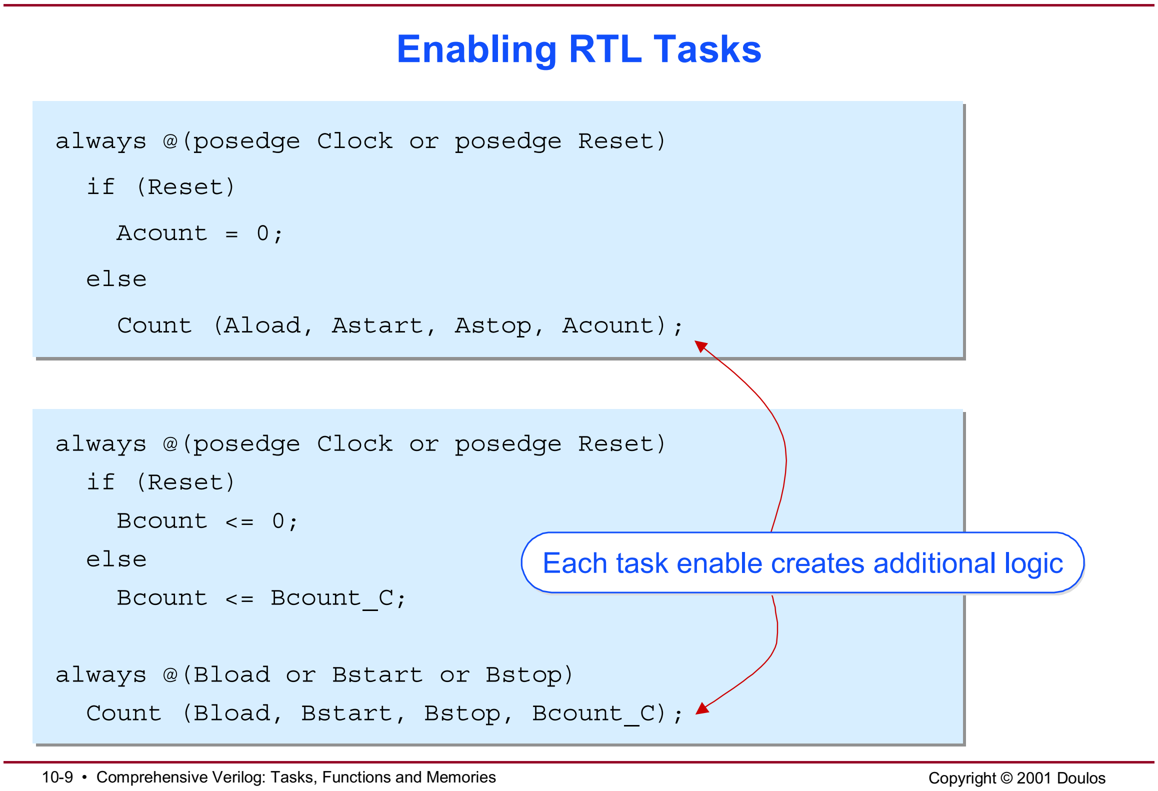
Synthesis

Synthesis tools require that tasks represent blocks of combinational logic, with the qualification that out and inout parameters may be synthesized as flip-flops if the task is called from a clocked always block (as shown on the next page). The practical consequence is that for synthesis, tasks must not detect edges or contain event controls.

It is very important that a task that may be called more than once at the same (simulation) time must not include any delays, otherwise two concurrent calls might interfere with each other.

Note that "blocking" assignments are used for Cnt in this example. This is necessary because "non-blocking" (RTL) assignments would not update Cnt before the end of the task, and so enabling the task would have no effect.

Enabling RTL Tasks



Enabling RTL Tasks

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Acount = 0;
  else
    Count (Aload, Astart, Astop, Acount);
```

```
always @(posedge Clock or posedge Reset)
  if (Reset)
    Bcount <= 0;
  else
    Bcount <= Bcount_C;

always @(Bload or Bstart or Bstop)
  Count (Bload, Bstart, Bstop, Bcount_C);
```

Each task enable creates additional logic

10-9 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

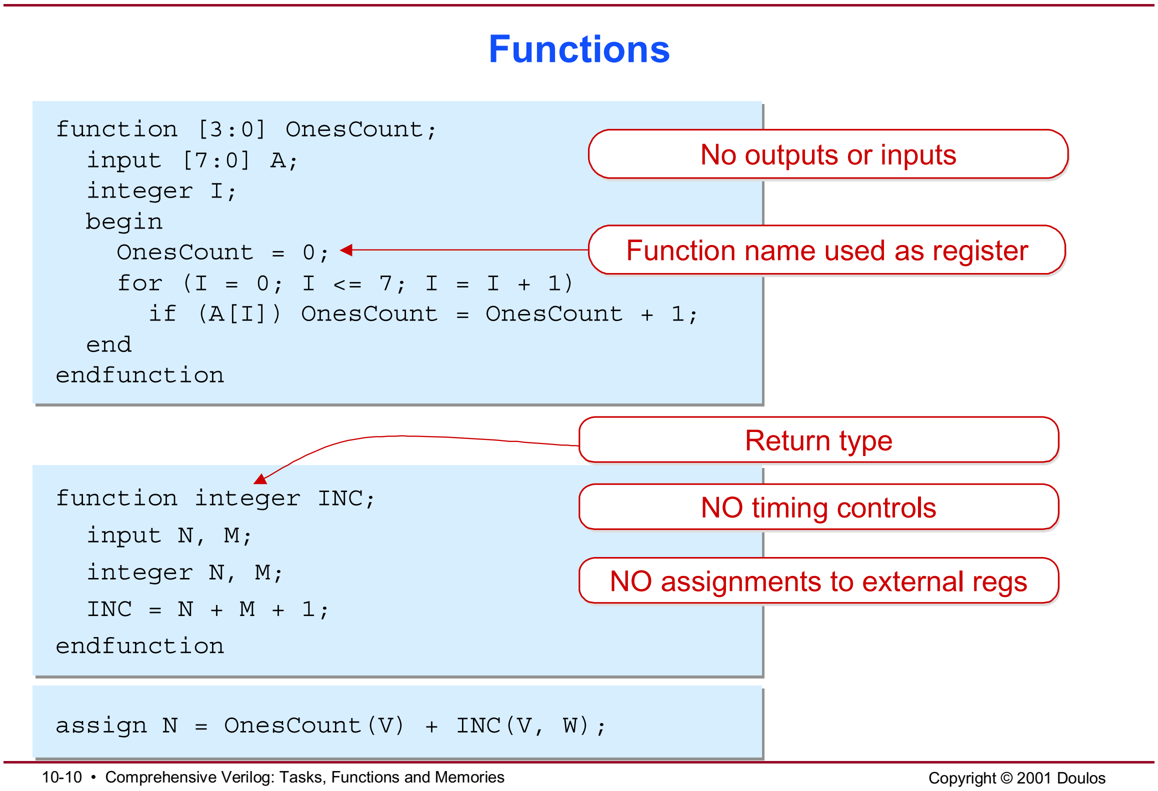
Each call to a task is synthesized by replacing the task enable with a copy of the logic represented by the task. This example describes two separate counters. In other words, a task is NOT treated as a single copy of a resource that can be shared by calling it at different points in an always statement - you would need to code that explicitly.

In the example at the top, it is not possible to move the asynchronous reset and clock edge detection into the task. Also, a blocking assignment has been used for Acount; synthesis tools may require this because the task Count uses a blocking assignment for the inout Cnt.

The second example shows how non-blocking assignments can be used together with a task; however, two always blocks are required.

For these reasons, it is often better to structure Verilog using functions (see the next slide) or modules rather than tasks.

Functions



Functions

```
function [3:0] OnesCount;
  input [7:0] A;
  integer I;
  begin
    OnesCount = 0;
    for (I = 0; I <= 7; I = I + 1)
      if (A[I]) OnesCount = OnesCount + 1;
  end
endfunction
```

No outputs or inputs

Function name used as register

Return type

```
function integer INC;
  input N, M;
  integer N, M;
  INC = N + M + 1;
endfunction
```

NO timing controls

NO assignments to external regs

```
assign N = OnesCount(V) + INC(V, W);
```

10-10 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

Functions are used to define any logical or mathematical function which calculates a single result based on the values of a set of parameters. Functions may be used in expressions in procedural statements and continuous assignments.

A function consists of an optional vector width or register type (integer, time); a name; a set of input arguments (outputs and inouts are not allowed); optional register declarations; and a single statement or a sequence of statements surrounded by begin ... end.

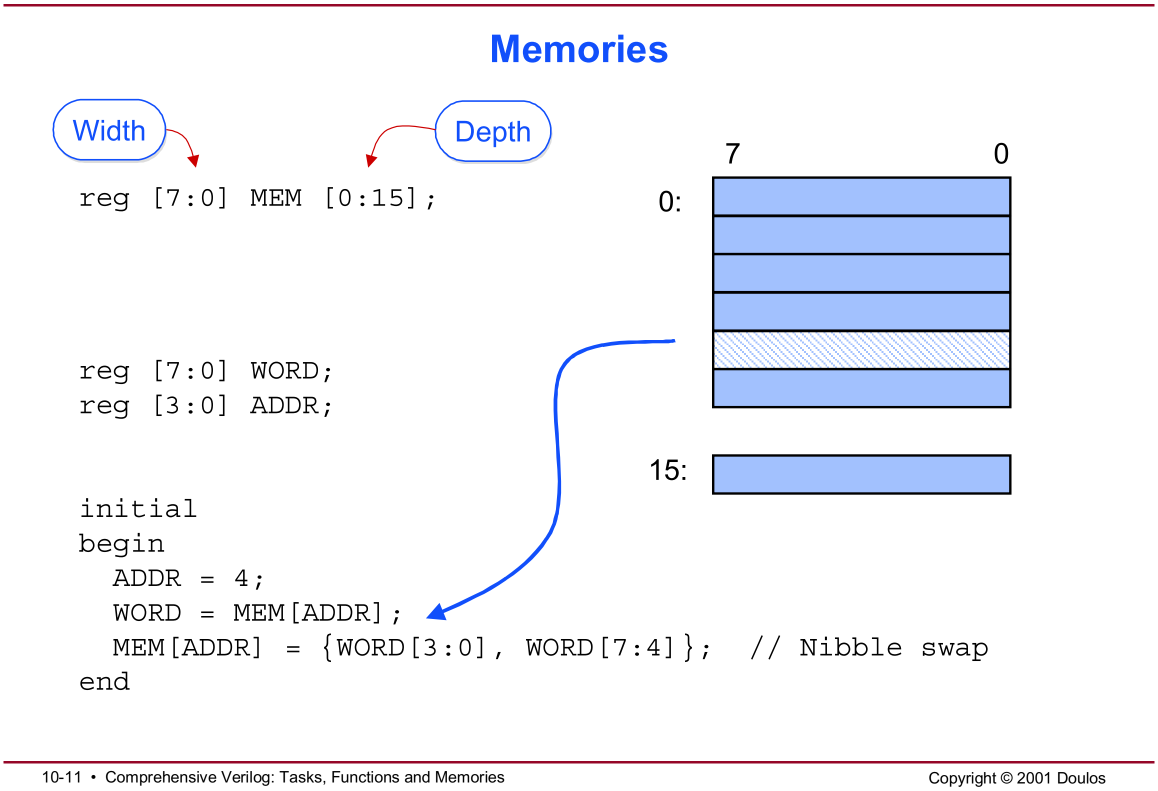
A function returns a value. The name of the function is used as a register, and when the function completes, that value is the value the function returns.

A function may not include references to regs or wires declared outside the function. Nor may a function contain timing controls such as delays (#) and events (@).

Synthesis

Functions may be synthesized. They describe combinational logic.

Memories



Memories

Width

Depth

```
reg [7:0] MEM [0:15];



reg [7:0] WORD;
reg [3:0] ADDR;


initial
begin
  ADDR = 4;
  WORD = MEM[ADDR];
  MEM[ADDR] = {WORD[3:0], WORD[7:4]};  // Nibble swap
end
```

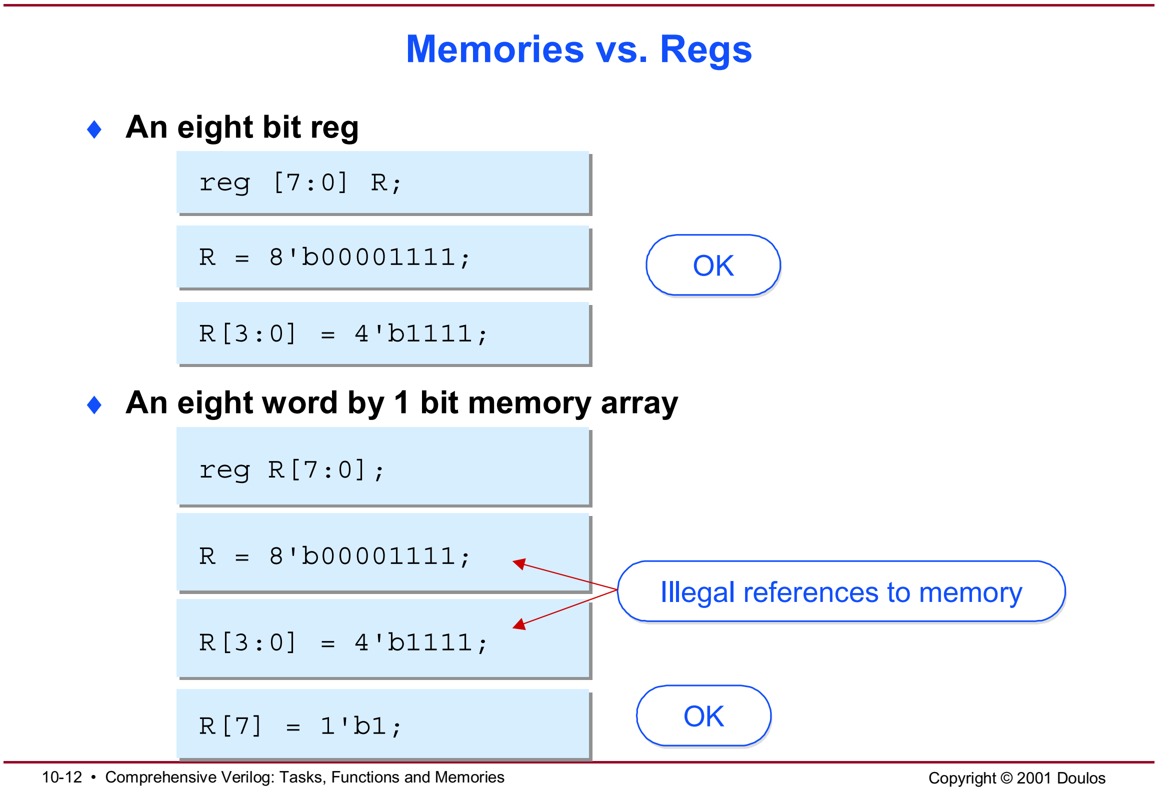
10-11 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

Verilog supports the declaration and use of memory arrays. A memory array is a one dimensional array of registers. Arrays of integers and regs of various widths are allowed.

Memory arrays may only be addressed one word at a time. There is no simple way of assigning values to or reading values from individual bits of the vectors that make up the array, or of manipulating the array as a single entity.

Memories vs. Regs



Memories vs. Regs

- An eight bit reg

```
reg [7:0] R;
```

```
R = 8'b00001111;
```

OK

```
R[3:0] = 4'b1111;
```

- An eight word by 1 bit memory array

```
reg R[7:0];
```

```
R = 8'b00001111;
```

Illegal references to memory

```
R[3:0] = 4'b1111;
```

```
R[7] = 1'b1;
```

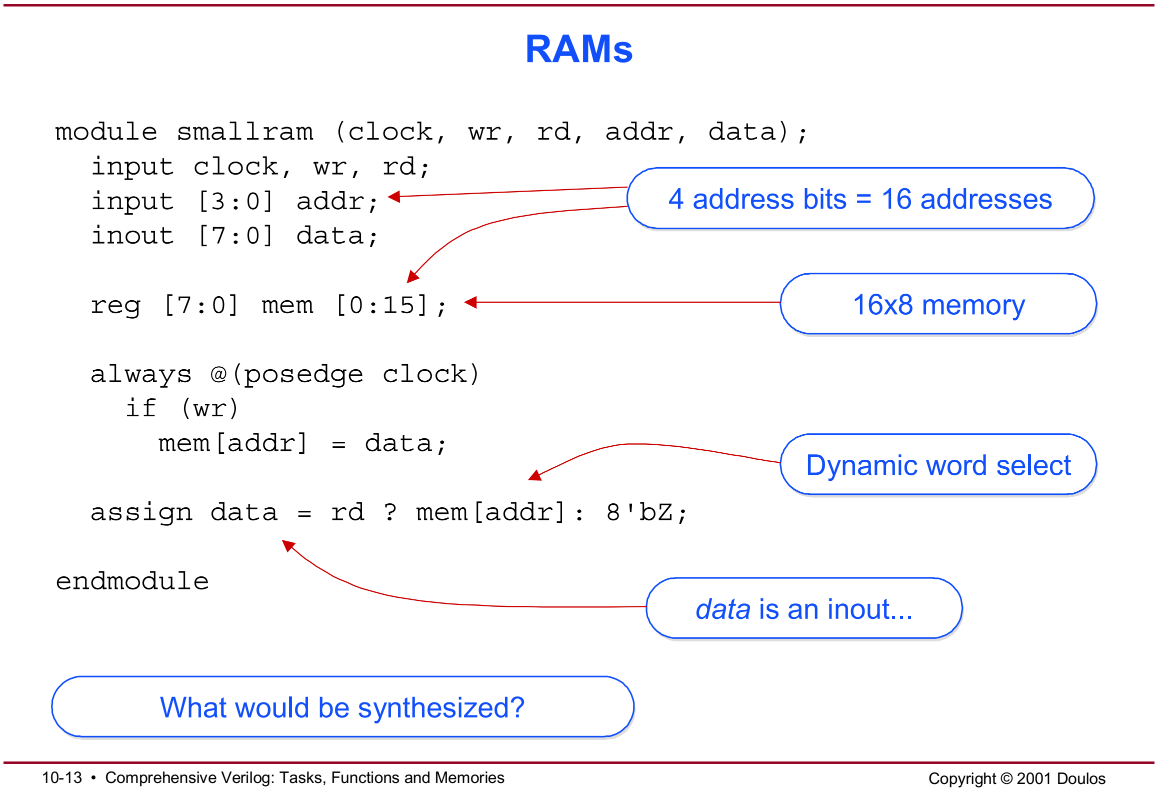
OK

10-12 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

The declaration of an eight bit reg and of an eight word by one bit memory are similar. If you inadvertently declare a memory instead of a reg, errors will be reported if you try to reference the name of the memory illegally.

RAMs



RAMs

```
module smallram (clock, wr, rd, addr, data);
  input clock, wr, rd;
  input [3:0] addr;
  inout [7:0] data;

  reg [7:0] mem [0:15];

  always @(posedge clock)
    if (wr)
      mem[addr] = data;

  assign data = rd ? mem[addr]: 8'bZ;

endmodule
```

4 address bits = 16 addresses

16x8 memory

Dynamic word select

data is an inout...

What would be synthesized?

10-13 • Comprehensive Verilog: Tasks, Functions and Memories

Copyright © 2001 Doulos

Notes:

Memory arrays should be used with care for synthesis, because each indexed name will be synthesized as a multiplexor structure. If a RAM were described and synthesized in this way, the result would not be a regular RAM structure but a huge unoptimizable mass of multiplexors. However, some synthesis tools include an explicit memory synthesis capability, which allows the synthesis tool to generate a practical RAM architecture from a behavioral description of a RAM conforming to certain guidelines.

ASICs

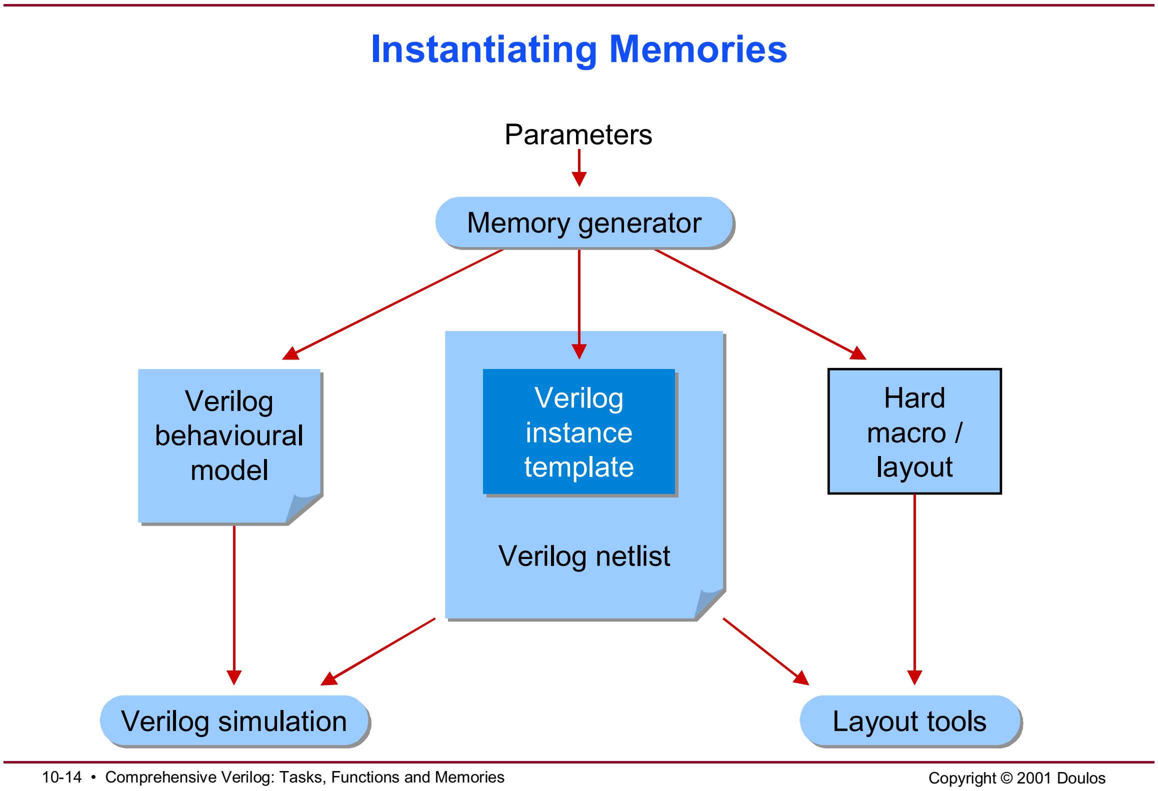
ASIC Memory synthesis is typically restricted to synthesizing purely synchronous RAMs constructed out of gates and flipflops. Multi-port RAMs may be allowed.

The fundamental rules are that a value written into the RAM in one clock cycle must not be read out again in the same cycle, and that two ports must not write to the same address in the same cycle.

FPGAs

Certain FPGA synthesis tools are able to infer the use of memory macros from RTL descriptions such as this, but as for ASICs, this capability is both tool dependent and technology dependent.

Instantiating Memories



Instantiating Memories

10-14 • Comprehensive Verilog: Tasks, Functions and Memories

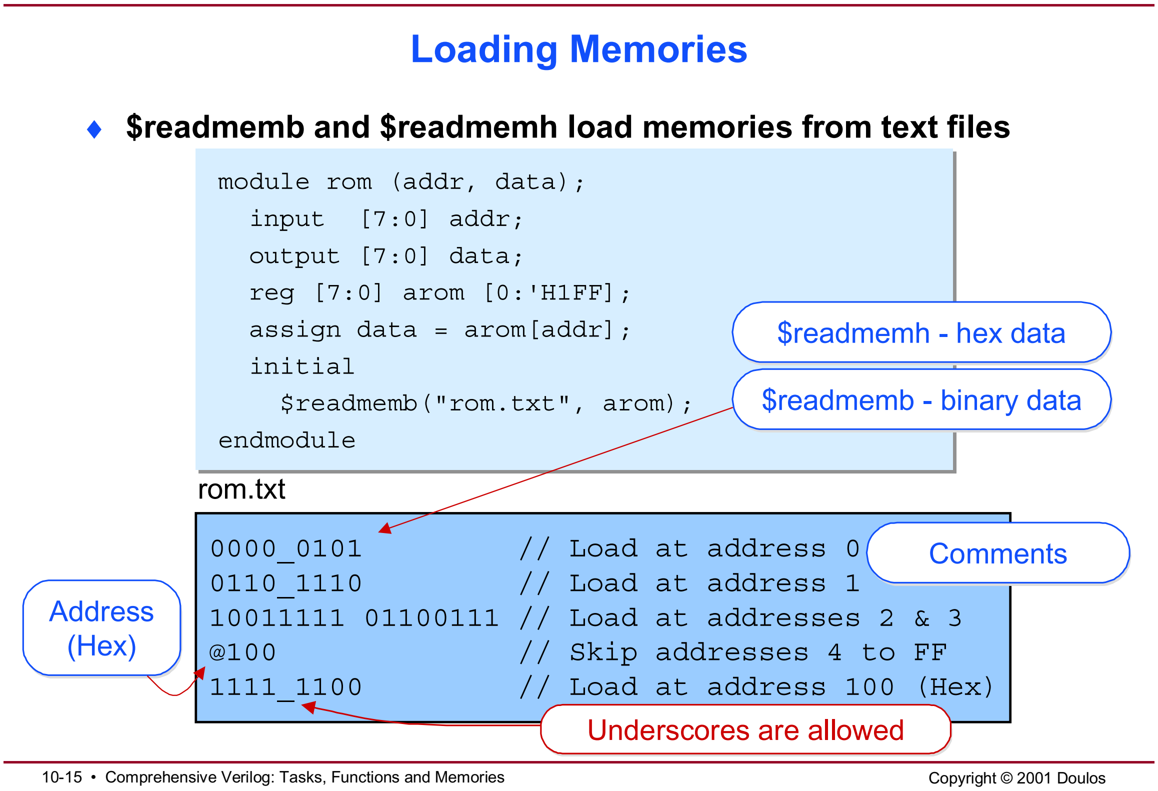
Copyright © 2001 Doulos

Notes:

The more general solution to the problem of including memories in a design is to instantiate the memory macros as components. The diagram shows the design flow. Memory parameters such as bus widths and memory depth are fed into a specialized piece of software which generates the actual memory, i.e. a hard macro cell or a physical layout as appropriate for the implementation technology. The memory generator may also create a behavioral Verilog model purely for simulation purposes, and a module instance template to facilitate instantiation in the top level netlist.

This same design flow can be used to generate other parameterized macros such as counters and shift registers.

Loading Memories



Loading Memories

- $readmemb and $readmemh load memories from text files

```
module rom (addr, data);
  input  [7:0] addr;
  output [7:0] data;
  reg [7:0] arom [0:'H1FF];
  assign data = arom[addr];
  initial
    $readmemb("rom.txt", arom);
endmodule
```

$readmemh - hex data

$readmemb - binary data

rom.txt

```
0000_0101          // Load at address 0
0110_1110          // Load at address 1
10011111 01100111 // Load at addresses 2 & 3
@100               // Skip addresses 4 to FF
1111_1100          // Load at address 100 (Hex)
```

Comments

Address (Hex)

Underscores are allowed

10-15 • Comprehensive Verilog: Tasks, Functions and Memories

Notes:

Verilog has a pair of system tasks to load data from a text file to a Verilog memory array.

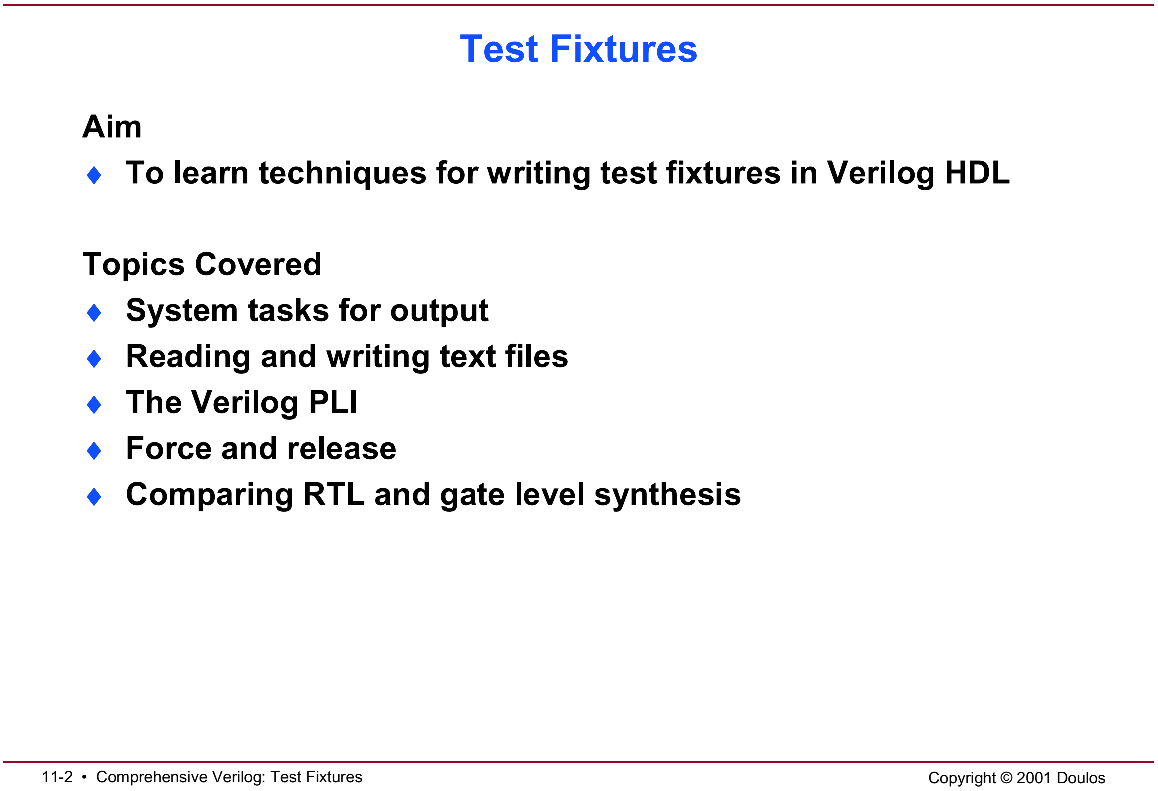
The format of the files that $readmemb and $readmemh read is shown opposite.

A file may contain binary or hexadecimal data, separated by white spaces. The width and base are not required, as these are inferred from the size of the memory being initialized.

A file may also contain comments and addresses. An address is a hexadecimal number preceded by an @ character. When an address is encountered, the following data words are read into that address, and subsequent addresses. Any addresses that are skipped over in this way remain at an unknown value (8'bX).

Module 11
Test Fixtures

Test Fixtures



Test Fixtures

Aim

- To learn techniques for writing test fixtures in Verilog HDL

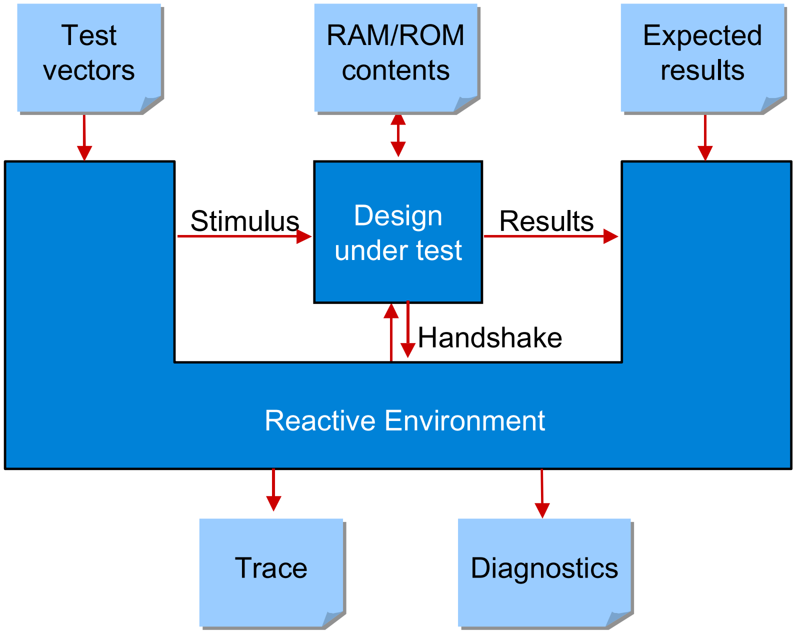
Topics Covered

- System tasks for output
- Reading and writing text files
- The Verilog PLI
- Force and release
- Comparing RTL and gate level synthesis

Notes:

Modelling the Test Environment

Modelling the Test Environment



11-3 • Comprehensive Verilog: Test Fixtures

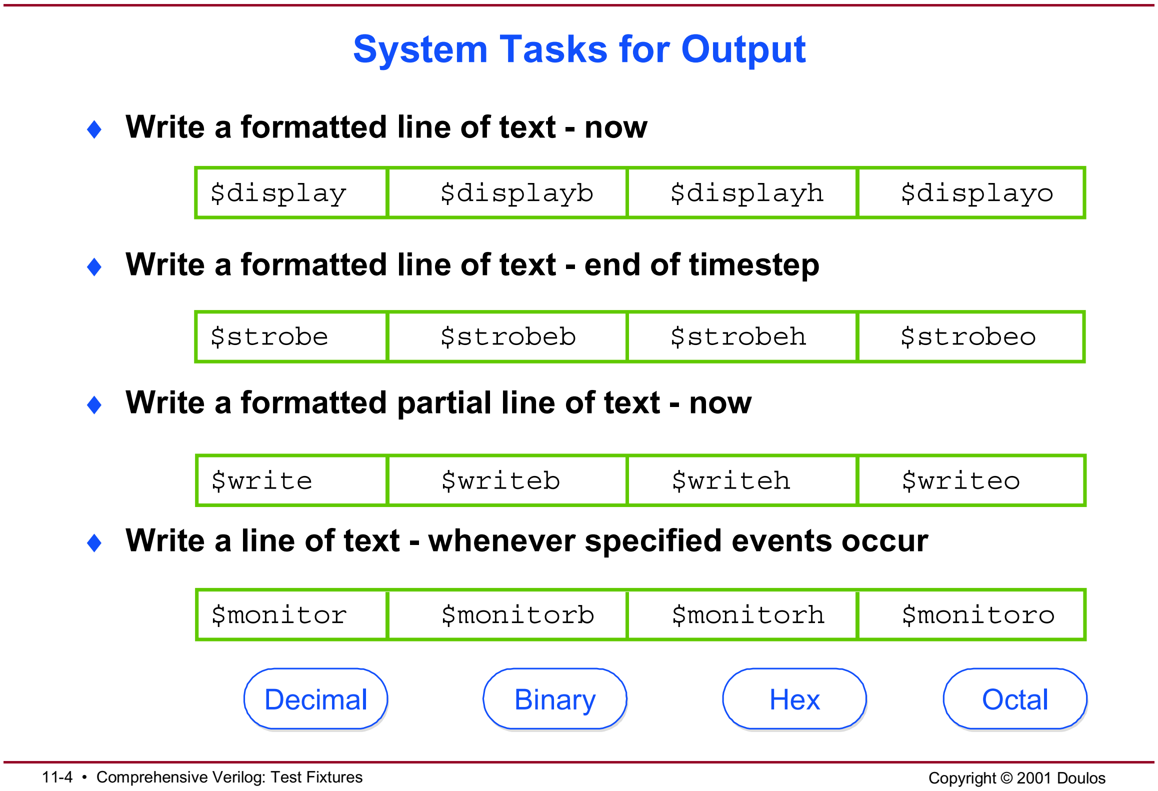
Notes:

It is usual to use Verilog HDL to describe both the hardware design and the environment used to test the design, analogous to a physical test bench. A Verilog HDL test fixture includes code to generate the test cases, and possibly code to analyze the output from the design or compare actual output with expected output.

Test Fixtures

A Verilog HDL test fixture often does more than just generate stimulus, as illustrated. It can read test vectors from a file, write outputs to a file, compare outputs with expected responses read from a file, write out statistics or diagnostics, or load and dump memory contents.

System Tasks for Output



System Tasks for Output

- Write a formatted line of text - now

$display	$displayb	$displayh	$displayo

- Write a formatted line of text - end of timestep

$strobe	$strobeb	$strobeh	$strobeo

- Write a formatted partial line of text - now

$write	$writeb	$writeh	$writeo

- Write a line of text - whenever specified events occur

$monitor	$monitorb	$monitorh	$monitoro
Decimal	Binary	Hex	Octal

Notes:

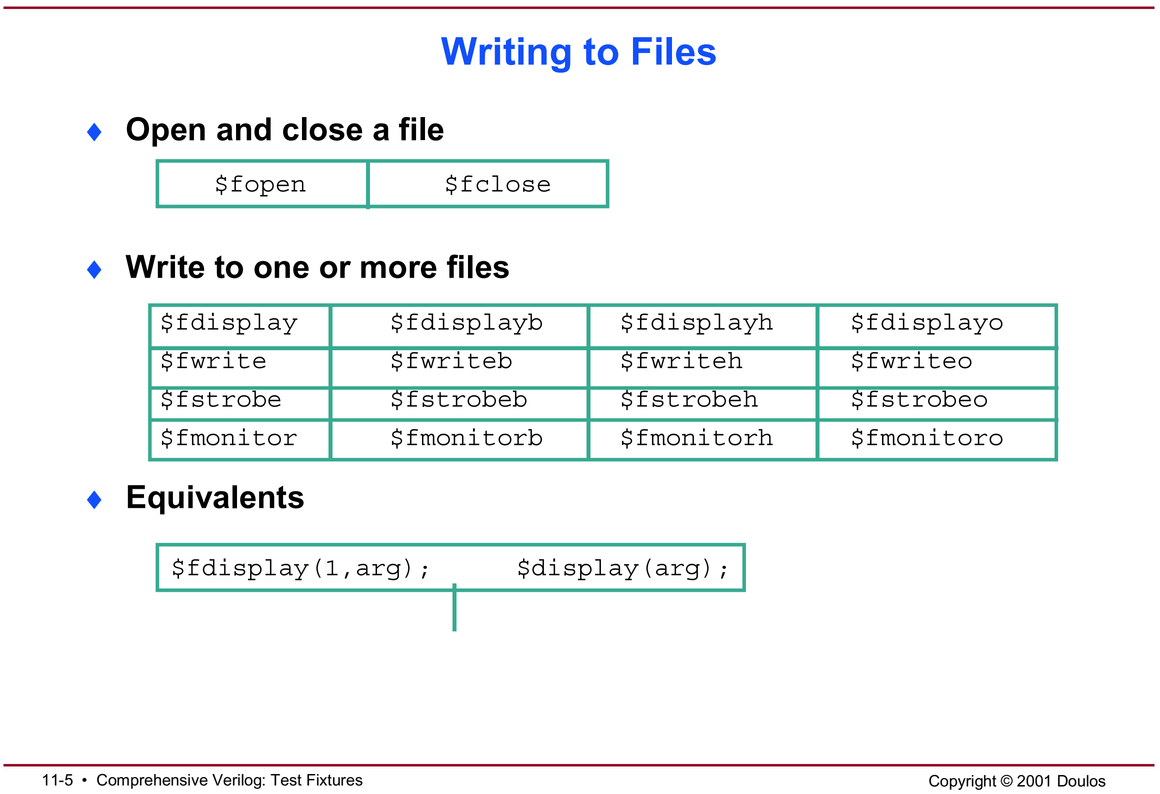
There is a family of 16 system tasks to write to the simulator log, and another family of 16 tasks to write to a file (See the next slide).

$write is equivalent to $display except that it doesn't write a newline character after the last argument.

$strobe is equivalent to $display except that it postpones writing out the values of its arguments until the very end of the simulation time step.

Each of $display, $write, $strobe and $monitor has four variants, each having a different default base. These are decimal, binary, octal and hex. Of course, you can also use format specifiers to use whatever base you want.

Writing to Files



Writing to Files

- **Open and close a file**

$fopen	$fclose

- **Write to one or more files**

$fdisplay	$fdisplayb	$fdisplayh	$fdisplayo
$fwrite	$fwriteb	$fwriteh	$fwriteo
$fstrobe	$fstrobeb	$fstrobeh	$fstrobeo
$fmonitor	$fmonitorb	$fmonitorh	$fmonitoro

- **Equivalents**

$fdisplay(1,arg);	$display(arg);

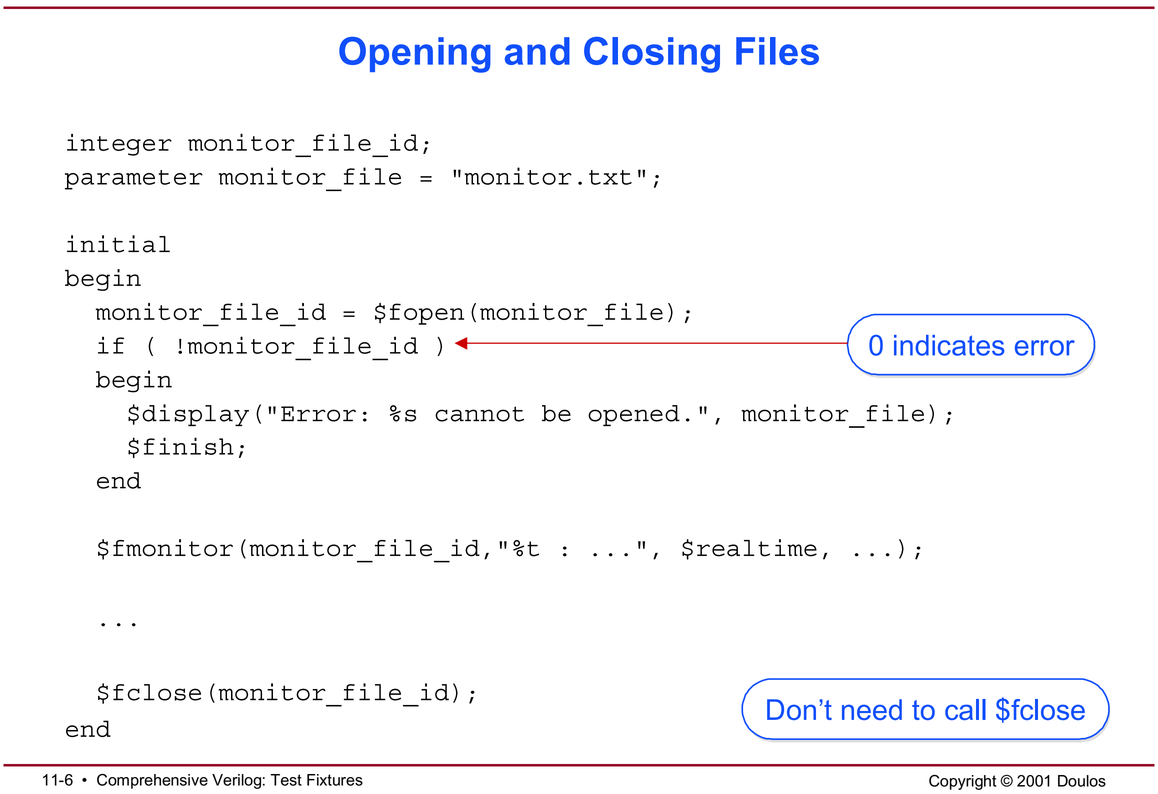
Copyright © 2001 Doulos

Notes:

Each of the output system tasks on the previous page has an equivalent system task for writing to a files. (The tasks we have already met produce results in the standard output,.typically the simulator's console window. This is equivalent to writing to file number 1.)

To write to a file, the file must be opened using $fopen, which is a system function. Files may be closed with $fclose.

Opening and Closing Files



Opening and Closing Files

```
integer monitor_file_id;
parameter monitor_file = "monitor.txt";

initial
begin
  monitor_file_id = $fopen(monitor_file);
  if ( !monitor_file_id )
  begin
    $display("Error: %s cannot be opened.", monitor_file);
    $finish;
  end

  $fmonitor(monitor_file_id,"%t : ...", $realtime, ...);

  ...

  $fclose(monitor_file_id);
end
```

0 indicates error

Don’t need to call $fclose

11-6 • Comprehensive Verilog: Test Fixtures

Notes:

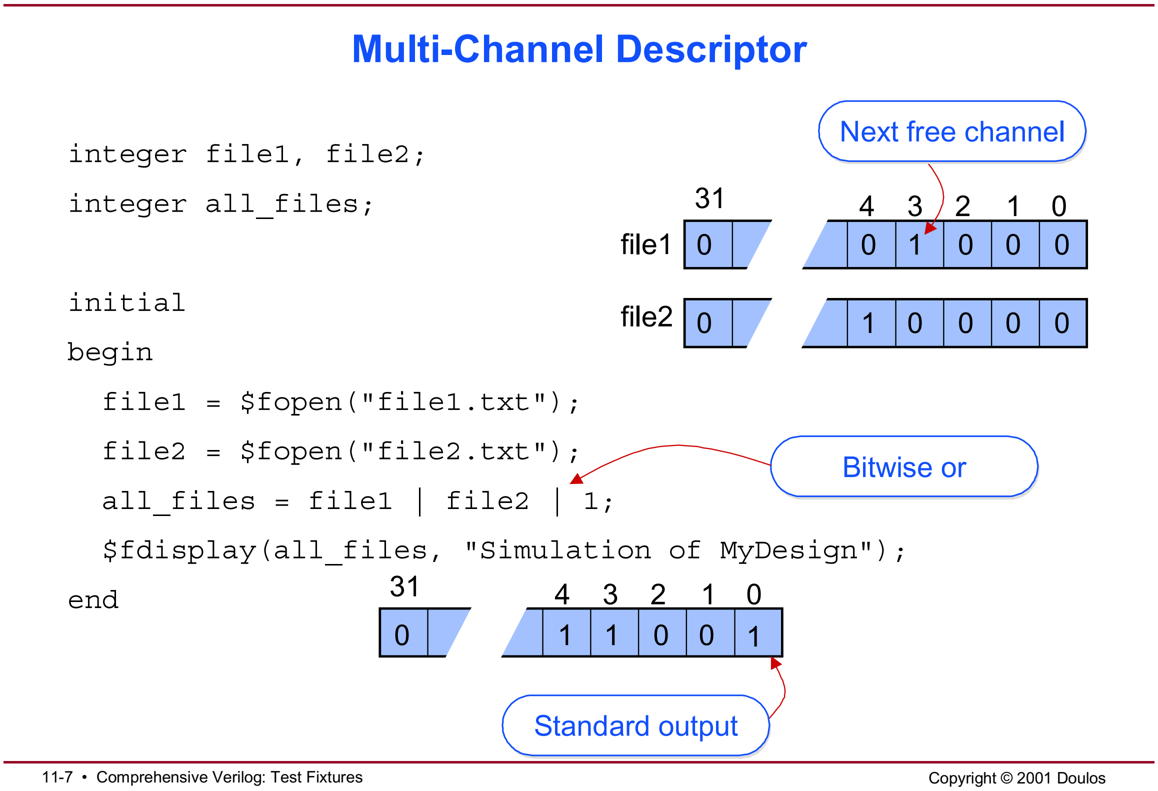
In order to use the tasks $fdisplay, $fmonitor etc., you must first open a file using the system function $fopen.

$fopen returns an integer value, or 0 if the file could not be opened. This may happen if, for example, the disk is full, or there is no privilege to write to the specified file.

The integer value that is returned is then used as the first argument when calling $fdisplay, $fmonitor etc. The other arguments are exactly the same as for the corresponding tasks ($display, $monitor etc.)

When you have finished writing to the file, you can close the file with the system task $fclose. Files that are not closed with this system task are automatically closed when simulation finishes.

Multi-Channel Descriptor



Multi-Channel Descriptor

```
integer file1, file2;
integer all_files;

initial
begin
  file1 = $fopen("file1.txt");
  file2 = $fopen("file2.txt");
  all_files = file1 | file2 | 1;
  $fdisplay(all_files, "Simulation of MyDesign");
end
```

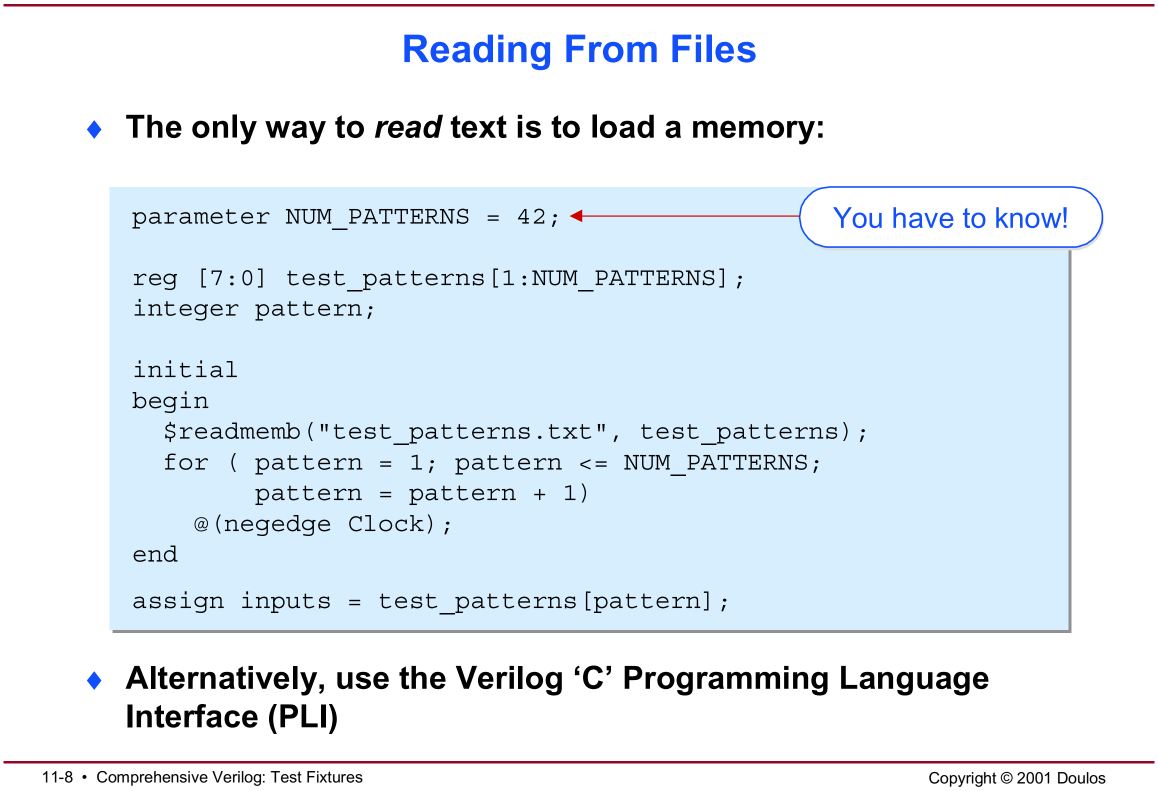
11-7 • Comprehensive Verilog: Test Fixtures

Copyright © 2001 Doulos

Notes:

The integer returned by $fopen is called a multi-channel descriptor (MCD), as is the first argument to $fdisplay etc. The MCD returned by $fopen has one bit set, corresponding to the file that has just been opened. MCDs with more than one bit set can be used to write the same text simultaneously to more than one file. The least significant bit of an MCD corresponds to the standard output or simulator log file. (Many Verilog simulators will write the standard output to a log file and to the simulator's main window.) Thus you can have up to 31 files (plus the log) open at any one time. $fclose releases the file channel, which can then be recycled for another file.

Reading From Files



Reading From Files

- The only way to *read* text is to load a memory:

```
parameter NUM_PATTERNS = 42;

reg [7:0] test_patterns[1:NUM_PATTERNS];
integer pattern;

initial
begin
  $readmemb("test_patterns.txt", test_patterns);
  for ( pattern = 1; pattern <= NUM_PATTERNS;
        pattern = pattern + 1)
    @(negedge Clock);
end

assign inputs = test_patterns[pattern];
```

You have to know!

- Alternatively, use the Verilog ‘C’ Programming Language Interface (PLI)

11-8 • Comprehensive Verilog: Test Fixtures

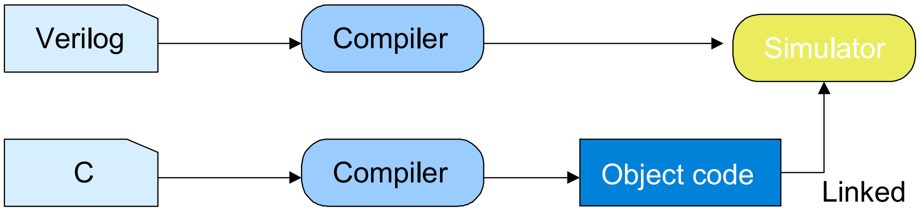
Notes:

There are no general purpose tasks or functions in Verilog to read arbitrary text from a file, except using the PLI. The only way to read from a file without using the PLI is to read binary or hexadecimal data using the system tasks $readmemb and $readmemh. These tasks respectively read binary and hexadecimal data into a memory array and were described in the section on tasks, functions and memories.

The Verilog PLI

The Verilog PLI

- Part of IEEE Std. 1364-1995
- Allows C functions to be called from Verilog



- Use the PLI for:
  - Calling C library functions
  - Direct access file I/O or fast file I/O
  - Graphical output direct to your screen
  - Including C models in your Verilog simulation
  - Interfacing to hardware modelers or emulators
  - Delay calculation

11-9 • Comprehensive Verilog: Test Fixtures

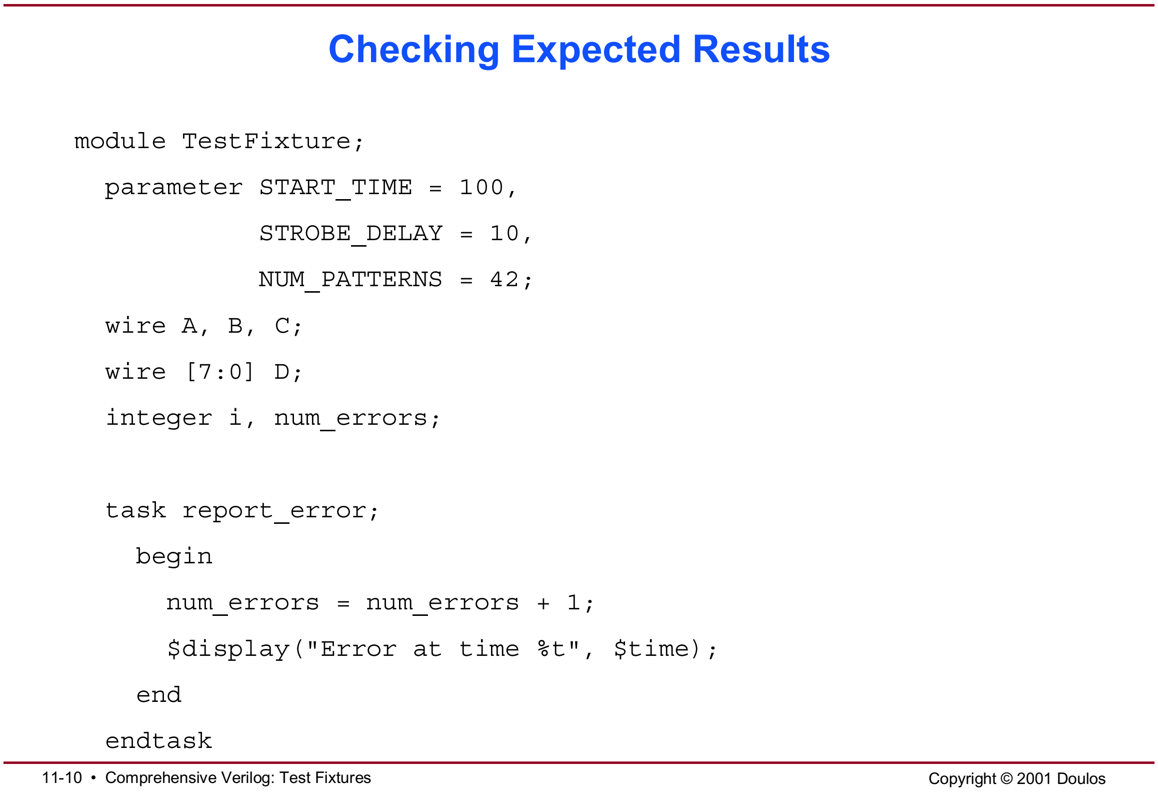
Notes:

The Verilog PLI permits functions written in C to be called from Verilog.

Although the PLI is as much part of the IEEE 1364 Verilog standard as the rest of the Verilog language, the exact manner in which C functions are linked to a particular simulator differ between different simulators.

The PLI is introduced in more detail in an appendix to this course.

Checking Expected Results



Checking Expected Results

```
module TestFixture;
  parameter START_TIME = 100,
            STROBE_DELAY = 10,
            NUM_PATTERNS = 42;
  wire A, B, C;
  wire [7:0] D;
  integer i, num_errors;

  task report_error;
    begin
      num_errors = num_errors + 1;
      $display("Error at time %t", $time);
    end
  endtask
```

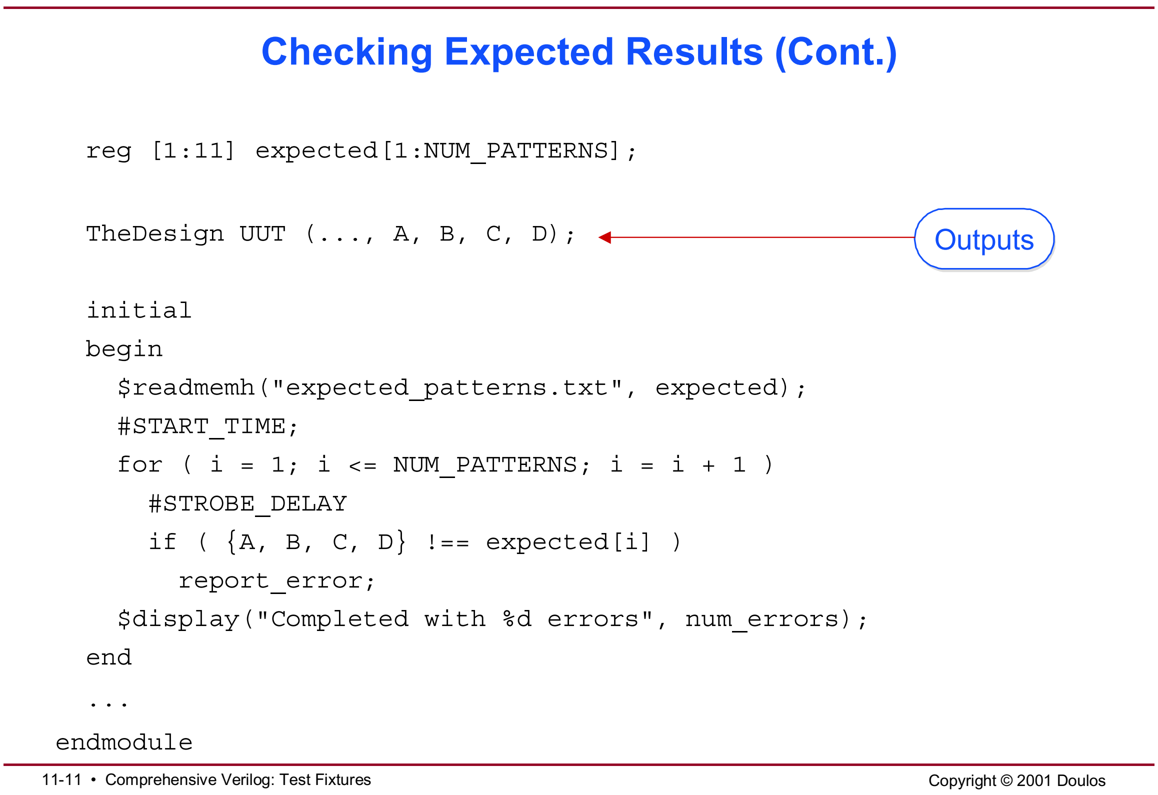
11-10 • Comprehensive Verilog: Test Fixtures

Notes:

Expected simulation results may also be read into a memory array. These may have been written by a test fixture in a previous simulation, or they may have been generated in some other way.

This slide shows the first part of an example showing how to read expected simulation results from a file. The example is continued on the next page.

Checking Expected Results (Cont.)



Checking Expected Results (Cont.)

```
    reg [1:11] expected[1:NUM_PATTERNS];

    TheDesign UUT (..., A, B, C, D);

    initial
    begin
      $readmemh("expected_patterns.txt", expected);
      #START_TIME;
      for ( i = 1; i <= NUM_PATTERNS; i = i + 1 )
        #STROBE_DELAY
        if ( {A, B, C, D} !== expected[i] )
          report_error;
      $display("Completed with %d errors", num_errors);
    end
    ...
  endmodule
```

Outputs

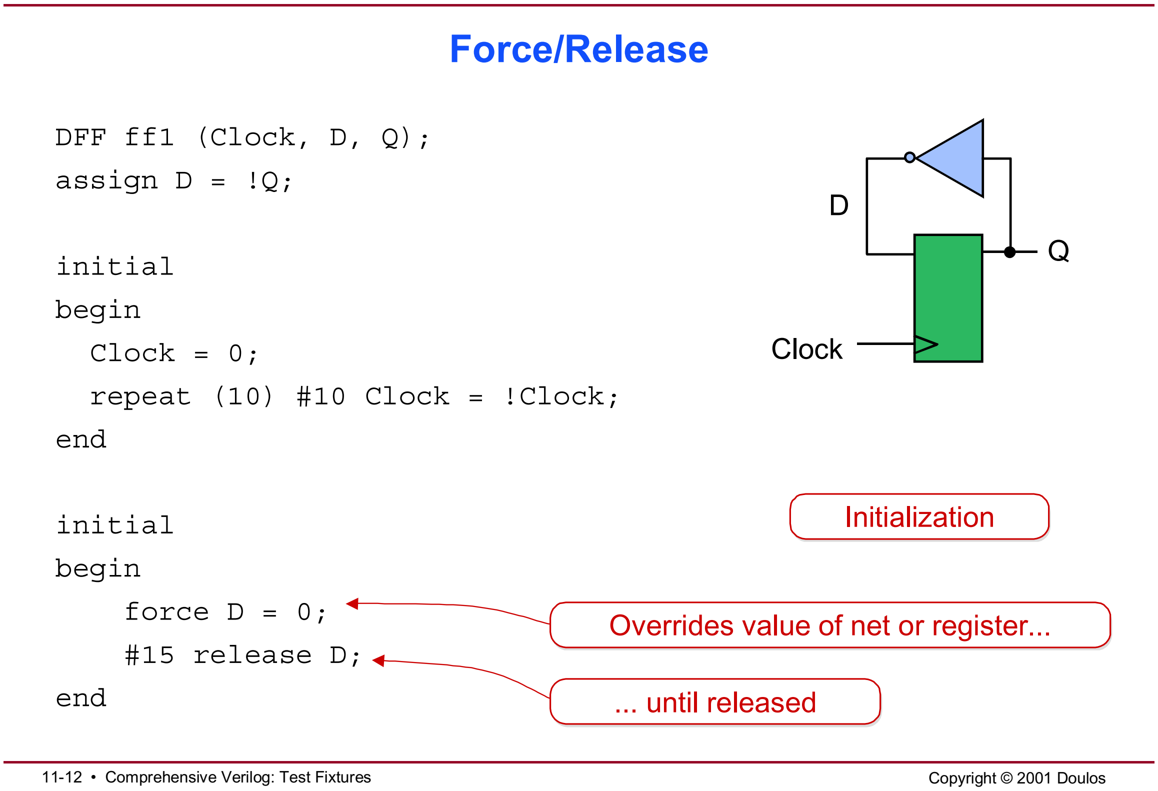
11-11 • Comprehensive Verilog: Test Fixtures

Copyright © 2001 Doulos

Notes:

Note the use of a task to encapsulate reporting an error during simulation.

Force/Release



Force/Release

```
DFF ff1 (Clock, D, Q);
assign D = !Q;

initial
begin
  Clock = 0;
  repeat (10) #10 Clock = !Clock;
end

initial
begin
    force D = 0;
    #15 release D;
end
```

Initialization

Overrides value of net or register...

... until released

11-12 • Comprehensive Verilog: Test Fixtures

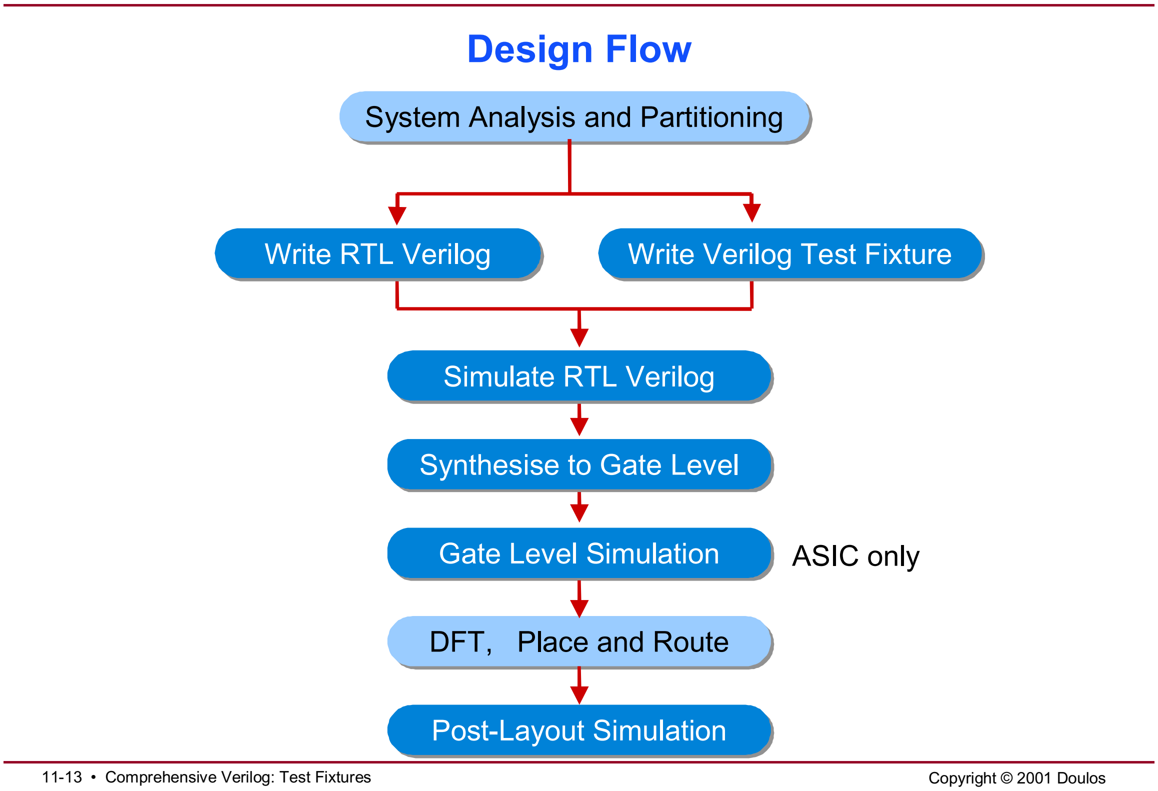
Notes:

force/release

Force and release are procedural statements intended for debugging. Force can be used to override the value of any net or register until it is released. These are the only statements in Verilog that cause nets (e.g. wires) to be assigned values in procedures (initial or always statements).

A common use of force and release is to initialize regs or wires, especially at the gate-level, where the simulator is unable to do so. This should be avoided as far as possible (How do you know that the hardware will initialize in the same way?), but is still sometimes required.

Design Flow



Design Flow

System Analysis and Partitioning

Write RTL Verilog

Write Verilog Test Fixture

Simulate RTL Verilog

Synthesise to Gate Level

Gate Level Simulation

ASIC only

DFT, Place and Route

Post-Layout Simulation

11-13 • Comprehensive Verilog: Test Fixtures

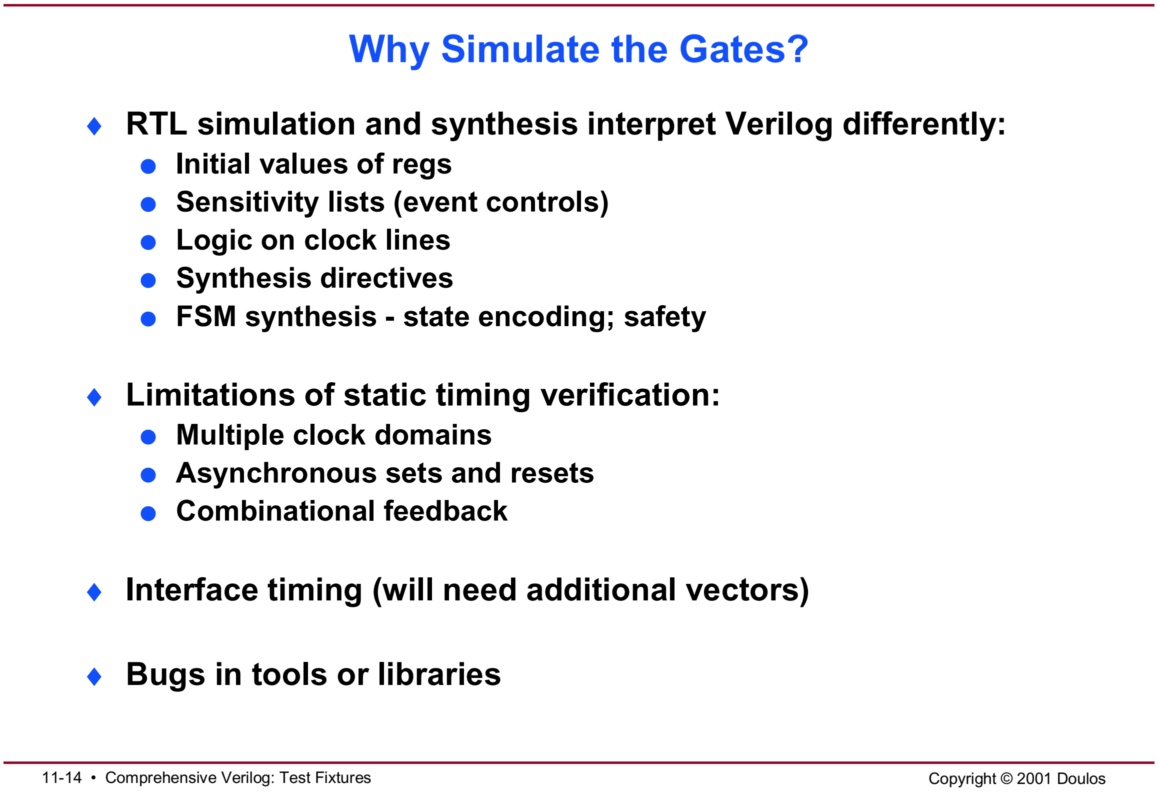
Copyright © 2001 Doulos

Notes:

Following synthesis, we need to simulate the design at gate level. For ASIC design, simulation is usually done both pre-layout (using estimated delays) and post-layout (using back-annotated delays). For FPGA and CPLD design, the pre-layout simulation is often omitted because of the inaccuracy of pre-layout timing estimates. Sometimes simulation may be omitted altogether and be replaced by programming an actual hardware part.

The high level design flow using Verilog involves simulating the design at two or more levels of abstraction, and comparing the simulation outputs across these level of abstraction. If synthesis is used, this means simulating the RTL Verilog, simulating the synthesized gate level design, and rigorously comparing the results of the two simulations.

Why Simulate the Gates?



Why Simulate the Gates?

- RTL simulation and synthesis interpret Verilog differently:
  - Initial values of regs
  - Sensitivity lists (event controls)
  - Logic on clock lines
  - Synthesis directives
  - FSM synthesis - state encoding; safety
- Limitations of static timing verification:
  - Multiple clock domains
  - Asynchronous sets and resets
  - Combinational feedback
- Interface timing (will need additional vectors)
- Bugs in tools or libraries

11-14 • Comprehensive Verilog: Test Fixtures

Notes:

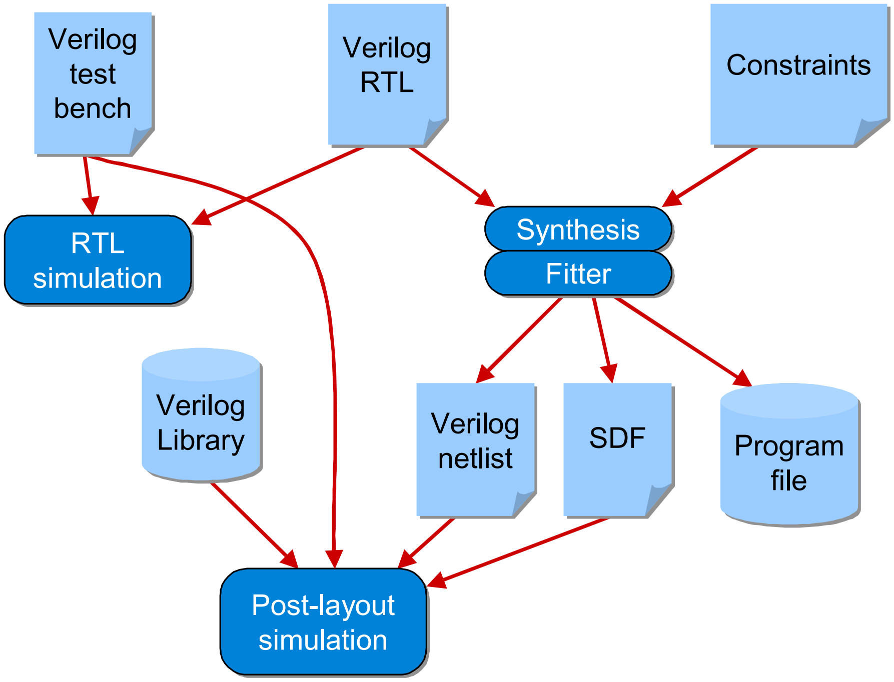
Ideally it might be thought that synthesis is "correct by construction", so the synthesized design will preserve the functionality of the original Verilog source, making functional simulation of the gates unnecessary. This is definitely not true in practice!

For one thing, simulators and synthesis tools can actually interpret the Verilog code differently. We have covered the main cases during this course. For example, synthesis tools ignore initial statements.

Secondly, the RTL simulation only represents the timing of the circuit at the clock cycle level, and does not include any of the detailed timing information present at the implementation level. Static timing analysis tools are very useful for locating problems with critical timing paths in synchronous circuits, but gate level simulation is the only way to verify the timing across asynchronous interfaces.

Verilog Flow for PLD Design

Verilog Flow for PLD Design



11-15 • Comprehensive Verilog: Test Fixtures

Notes:

Many FPGA and CPLD vendors support a Verilog design flow, which means that they provide Verilog gate-level models for their technology, and generate a Verilog netlist and an SDF file from their fitter or place and route tools.

The diagram opposite shows the tool flow for the simulation of a PLD or FPGA before and after fitting or place and route. Note the following:

- Only the RTL Verilog code is common to simulation and synthesis. The test fixture is for simulation only. The constraints are for synthesis/fitting only.

- With some tool combinations, constraints are input to the synthesis tool, and automatically passed through to the fitter. With other tool combinations, constraints are input directly to the fitter.

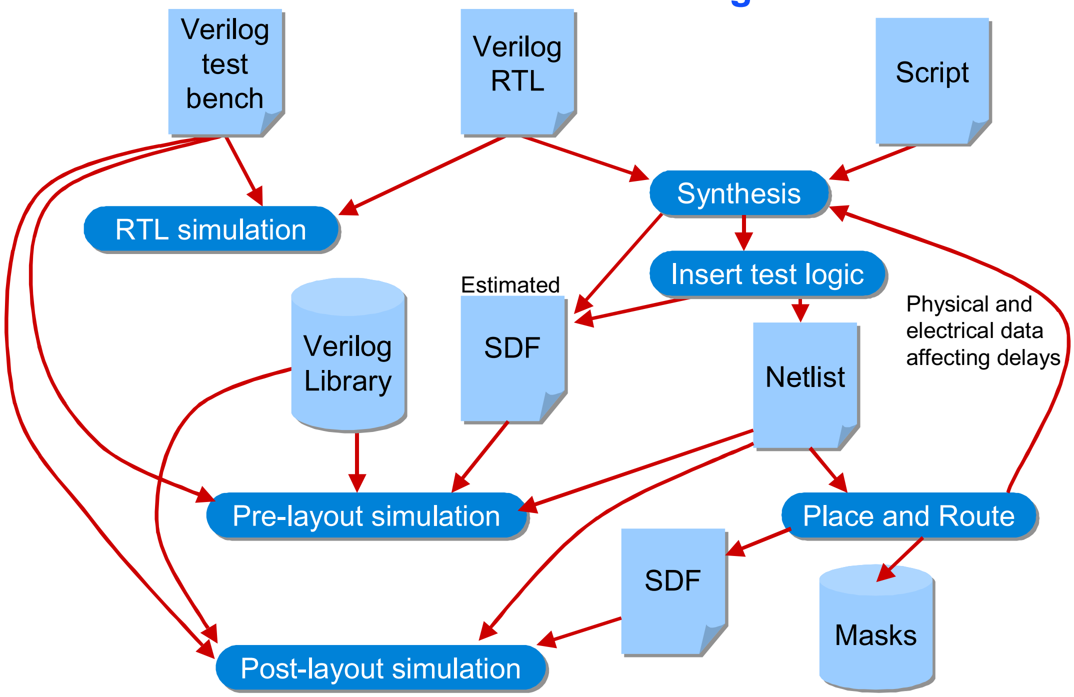
- The same test fixture is used for both simulations.

- Gate-level simulation is performed after fitting, not before fitting.

- The Verilog netlist from the fitter is for simulation only. The fitter directly produces the files necessary for programming the device.

Tool Flow for ASIC Design

Tool Flow for ASIC Design



11-16 • Comprehensive Verilog: Test Fixtures

Notes:

The tool flow for ASIC design is an extension of that for PLD design. Note the following:

· Three simulations are performed: RTL, pre-layout and post-layout.

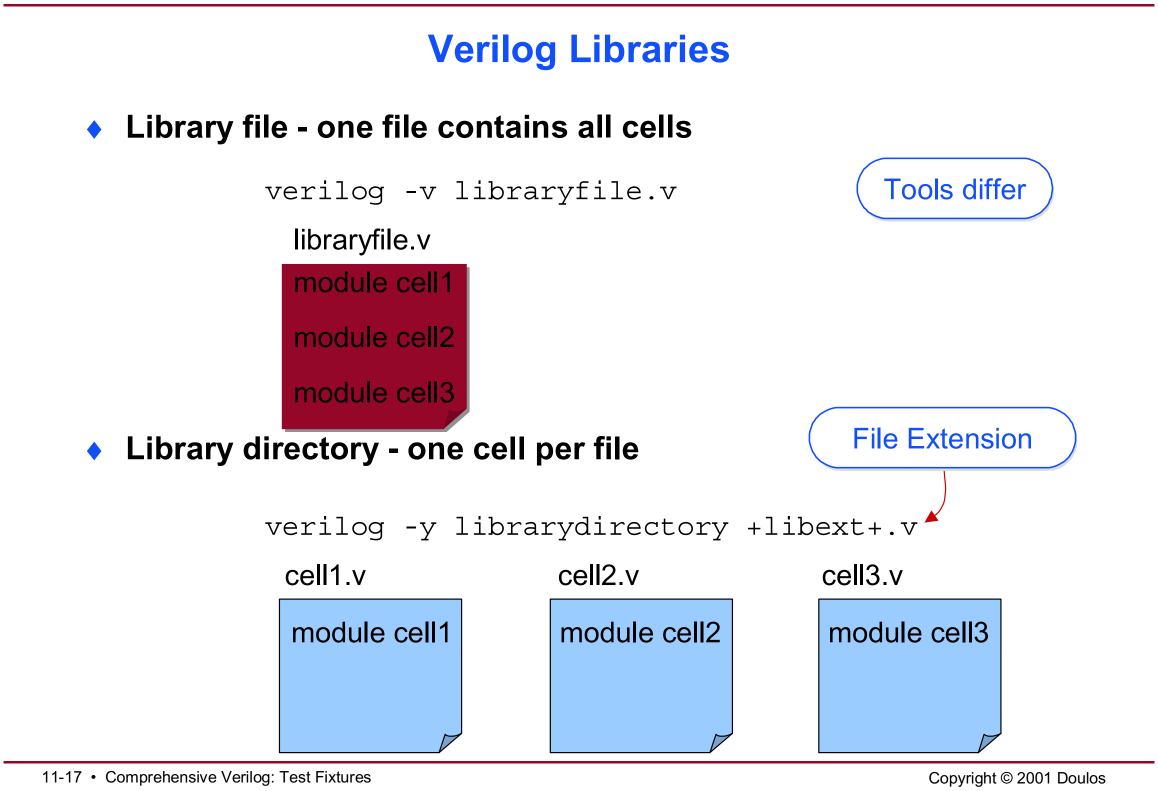
· The pre-layout simulation uses estimated delays calculated by the synthesis tool.

· Inserting logic to increase the testability of the design and performing automatic test pattern generation is an important part of many ASIC design flows, and impacts the simulation methodology. You may want to simulate before test logic insertion, after test logic insertion, or both.

· The pre- and post-layout simulations use the same Verilog netlists: only the SDF files are different.

· Physical and electrical data affecting delays can be fed back from layout to synthesis. Sophisticated design flows are being developed which try to meet timing constraints by iterating around the synthesis-layout loop a small number of times, thus achieving good timing closure.

Verilog Libraries



Verilog Libraries

- **Library file - one file contains all cells**

```
verilog -v libraryfile.v
```

- **Library directory - one cell per file**

```
verilog -y librarydirectory +libext+.v
```

11-17 • Comprehensive Verilog: Test Fixtures

Copyright © 2001 Doulos

Notes:

A Verilog library can be a collection of modules in source code form. Many Verilog tools support such libraries. Other tools have proprietary library formats.

The significance of Verilog libraries is that they are scanned by the compiler to resolve module references i.e. to find modules that have not been defined elsewhere, the point being that only library modules actually used in the design will be compiled.

-v

Verilog libraries come in two forms. The -v command line flag indicates that the library is a single text file.

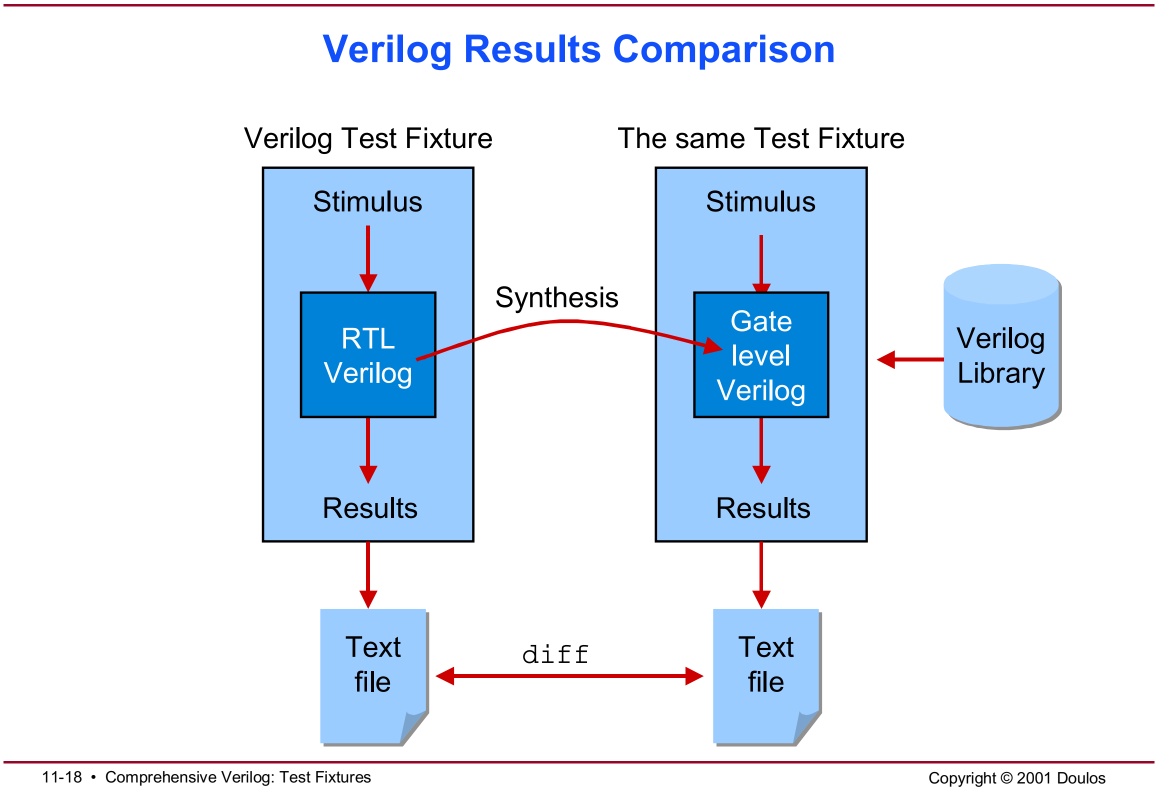
-y

The -y command line flag indicates that the library is a directory containing one source text file per module.

+libext+

The +libext+ command line flag is used together with -y and indicates the extension to be added to the module name by the compiler in order to construct the corresponding filename. So a module cell1 is expected to be found in the file cell1.v in the directory librarydirectory.

Verilog Results Comparison



Verilog Results Comparison

Verilog Test Fixture

Stimulus

RTL Verilog

Results

Synthesis

The same Test Fixture

Stimulus

Gate level Verilog

Results

Verilog Library

Text file

diff

Text file

11-18 • Comprehensive Verilog: Test Fixtures

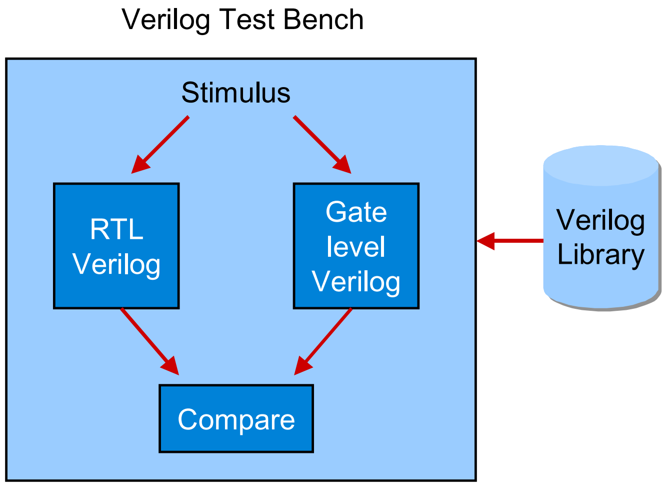
Copyright © 2001 Doulos

Notes:

The diagram shows a simple scenario for gate level verification, where simulation is entirely within the Verilog environment. The Verilog test fixture contains code to dump simulation results to a text file. The gate level description is plugged into the same Verilog test fixture. The resulting text files can be compared using UNIX diff or a similar utility.

Comparison On-the-fly

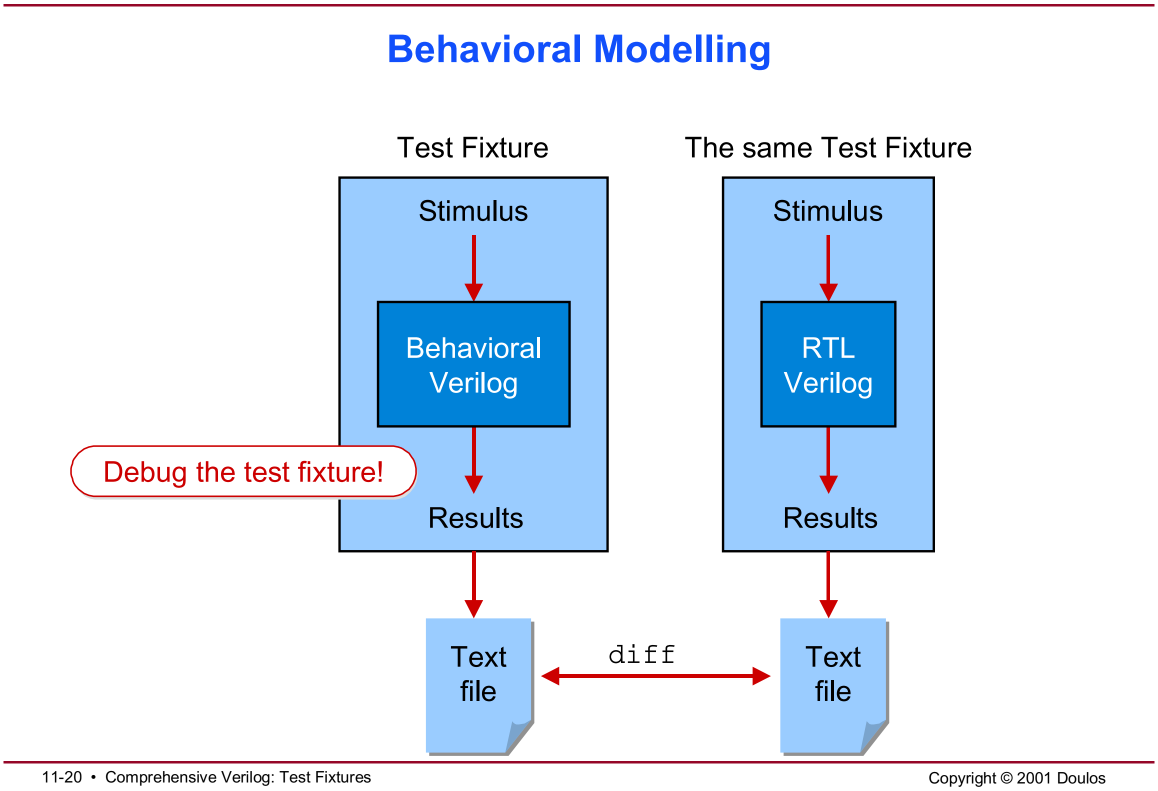
Comparison On-the-fly



Notes:

This diagram shows an alternative verification scenario, where two description of the same design are simulated side by side in the same test fixture, and the results compared on the fly. This has the advantage that simulation results do not have to be dumped to disk. The simulation can be set up such that it stops when there is a difference. The user can then investigate the reason for the difference.

Behavioral Modeling



Behavioral Modelling

Test Fixture

Stimulus

Behavioral Verilog

Results

Debug the test fixture!

The same Test Fixture

Stimulus

RTL Verilog

Results

Text file

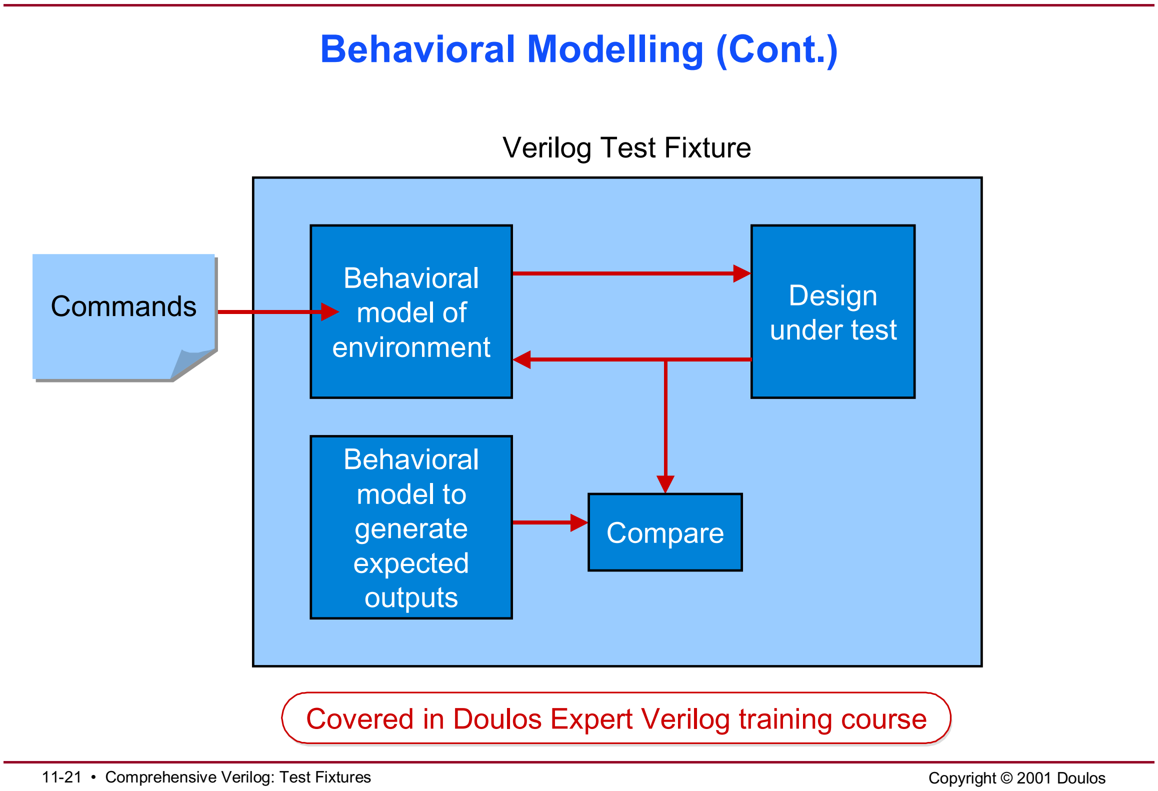
diff

Text file

Notes:

The test fixture usually takes longer to develop than the RTL code, but how do you debug the test fixture? One way is to start by writing a functionally accurate behavioral model of the design at the highest possible level of abstraction, and use this model to accelerate test bench development. You then have a higher quality test fixture at the start of the RTL debug cycle, so you can then concentrate on debugging the RTL code.

Behavioral Modeling (Cont.)



Behavioral Modelling (Cont.)

Verilog Test Fixture

Commands

Behavioral model of environment

Design under test

Behavioral model to generate expected outputs

Compare

Covered in Doulos Expert Verilog training course

11-21 • Comprehensive Verilog: Test Fixtures

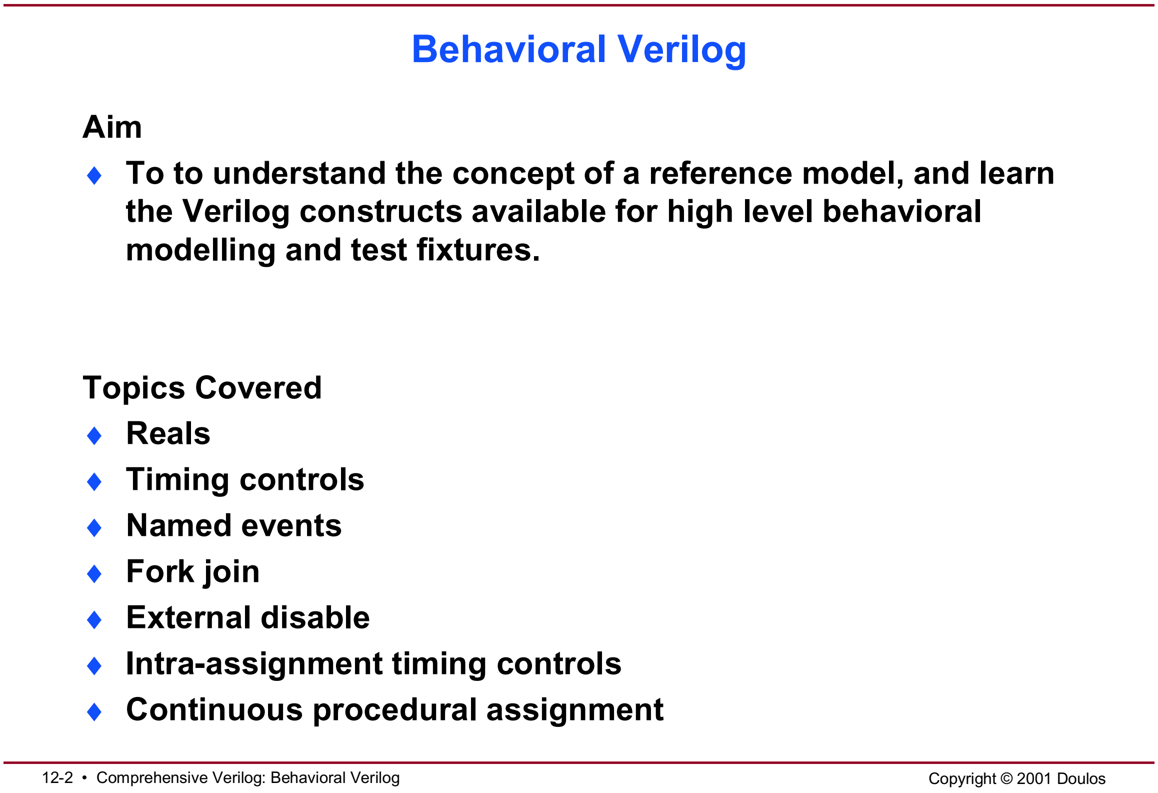
Notes:

The advantages of comparison-on-the fly and automated test fixtures can also be achieved by using behavioral models. For example, the stimulus generation block in a test fixture could be replaced with a behavioral model of the hardware interface connected to the design-under-test that is able to interact bi-directionally with the design, or a behavioral model could be used to generate the expected outputs of the design.

This subject is explored in full in the Doulos training course "Expert Verilog Verification". Behavioral modeling in Verilog is the subject of the next section in the present course.

Module 12
Behavioral Verilog

Behavioral Verilog



Behavioral Verilog

Aim

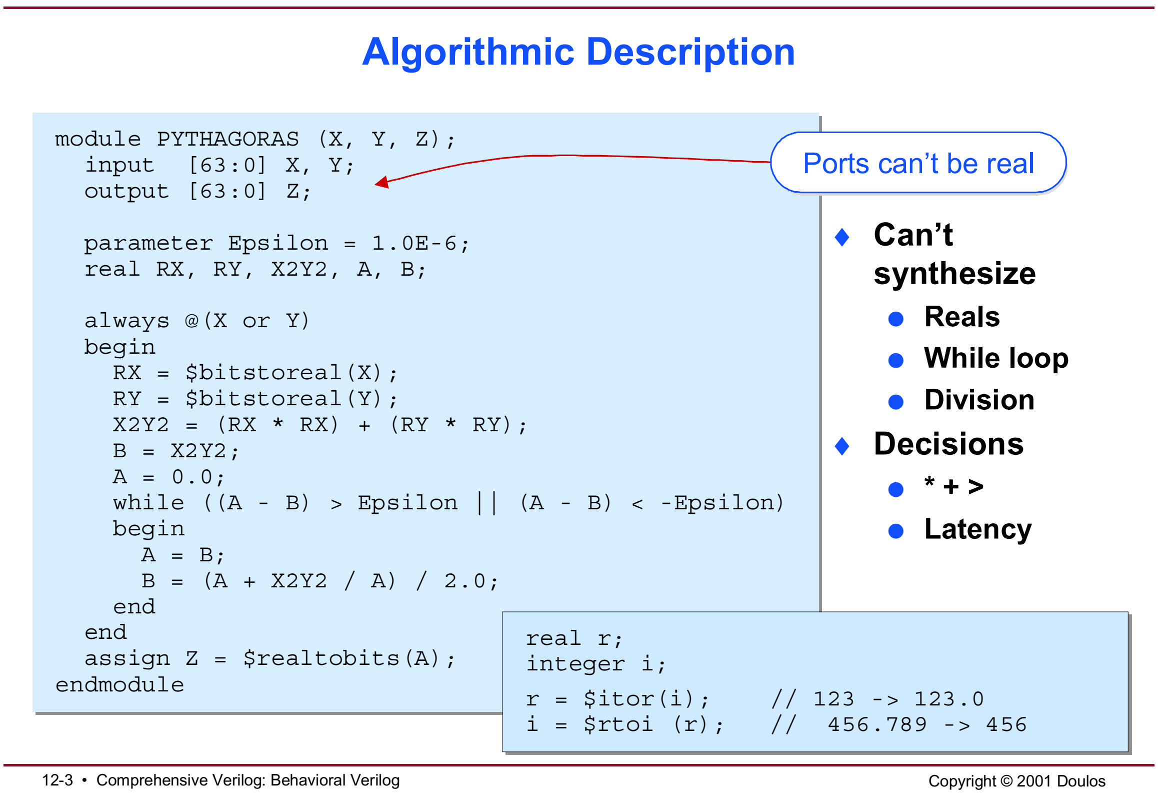
- To to understand the concept of a reference model, and learn the Verilog constructs available for high level behavioral modelling and test fixtures.

Topics Covered

- Reals
- Timing controls
- Named events
- Fork join
- External disable
- Intra-assignment timing controls
- Continuous procedural assignment

Notes:

Algorithmic Description



Algorithmic Description

```
module PYTHAGORAS (X, Y, Z);
  input  [63:0] X, Y;
  output [63:0] Z;

  parameter Epsilon = 1.0E-6;
  real RX, RY, X2Y2, A, B;

  always @(X or Y)
  begin
    RX = $bitstoreal(X);
    RY = $bitstoreal(Y);
    X2Y2 = (RX * RX) + (RY * RY);
    B = X2Y2;
    A = 0.0;
    while ((A - B) > Epsilon || (A - B) < -Epsilon)
    begin
      A = B;
      B = (A + X2Y2 / A) / 2.0;
    end
  end
  assign Z = $realtobits(A);
endmodule
```

Ports can't be real

- Can't synthesize
  - Reals
  - While loop
  - Division
- Decisions
  - * + >
  - Latency

```
real r;
integer i;

r = $itor(i);    // 123 -> 123.0
i = $rtoi (r);   //  456.789 -> 456
```

12-3 • Comprehensive Verilog: Behavioral Verilog

Copyright © 2001 Doulos

Notes:

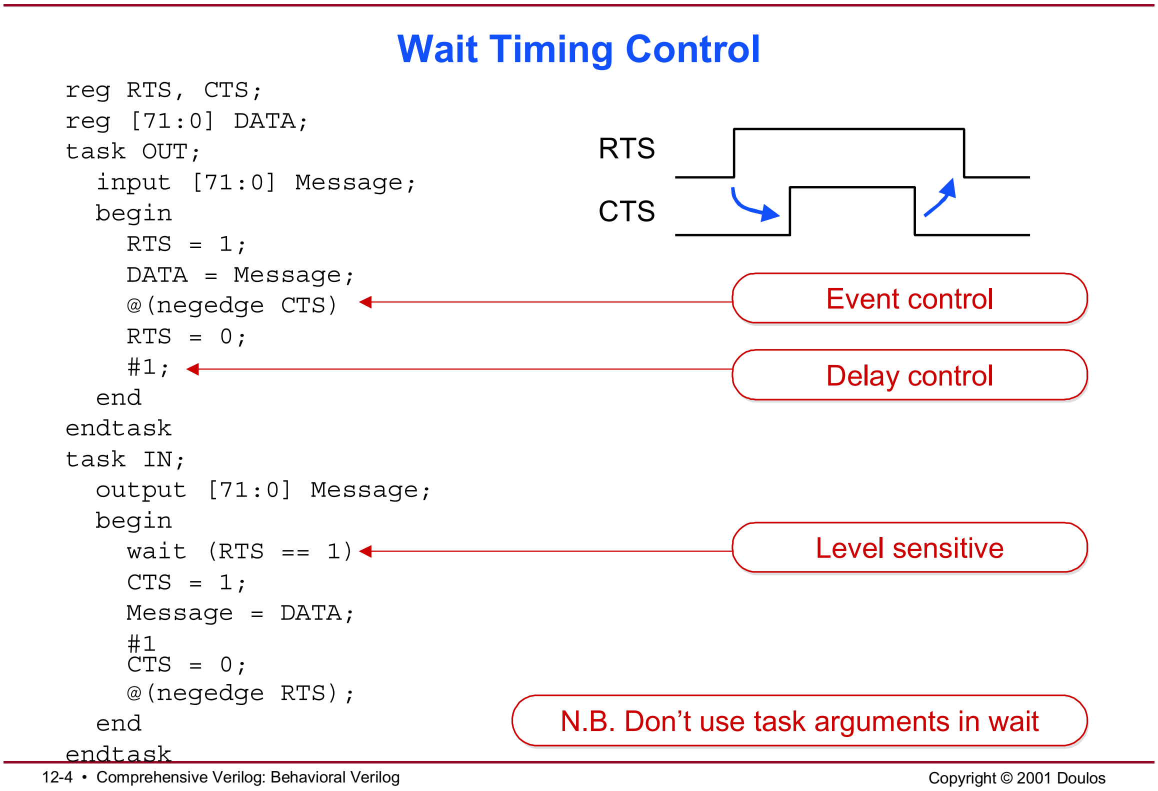
This example shows a module which performs an abstract arithmetical calculation. The module takes in the lengths (X and Y) of the two shortest sides of a right-angle triangle and calculates the length of the hypotenuse (Z) using Pythagorean's theorem (X2 + Y2 = Z2). The lengths of the sides are represented as real numbers.

The significant feature of the procedural block that models the behavior of the module is that it uses a successive approximation algorithm to calculate the square root function. The number of iterations of the while loop is not fixed in advance, but depends on the values of X and Y each time the always statement is executed. This description is abstracted from any hardware implementation because issues of hardware resource allocation and cycle level timing have still to be decided.

This module cannot be mapped directly into hardware - there are a lot of design decisions to make first, such as: How many multipliers? What multiplier design? How many comparators? How much parallelism? What bit level representation for floating point numbers?

Descriptions involving multiple communicating Verilog procedures, where each procedure represents a pure algorithm of this kind, can be used to describe systems at a high level of abstraction.

Wait Timing Control



Wait Timing Control

```
reg RTS, CTS;
reg [71:0] DATA;
task OUT;
  input [71:0] Message;
  begin
    RTS = 1;
    DATA = Message;
    @(negedge CTS)
    RTS = 0;
    #1;
  end
endtask
task IN;
  output [71:0] Message;
  begin
    wait (RTS == 1)
    CTS = 1;
    Message = DATA;
    #1
    CTS = 0;
    @(negedge RTS);
  end
endtask
```

Event control

Delay control

Level sensitive

N.B. Don’t use task arguments in wait

12-4 • Comprehensive Verilog: Behavioral Verilog

Copyright © 2001 Doulos

Notes:

Now let's consider communication and synchronization between procedural blocks. At RTL, each procedural block either has a single explicit clock, or represents combinational logic. If the restrictions of RTL descriptions are lifted, then procedural blocks can have any synchronous or asynchronous behavior. For example, the two tasks shown opposite each contain multiple timing controls, and model an asynchronous handshake.

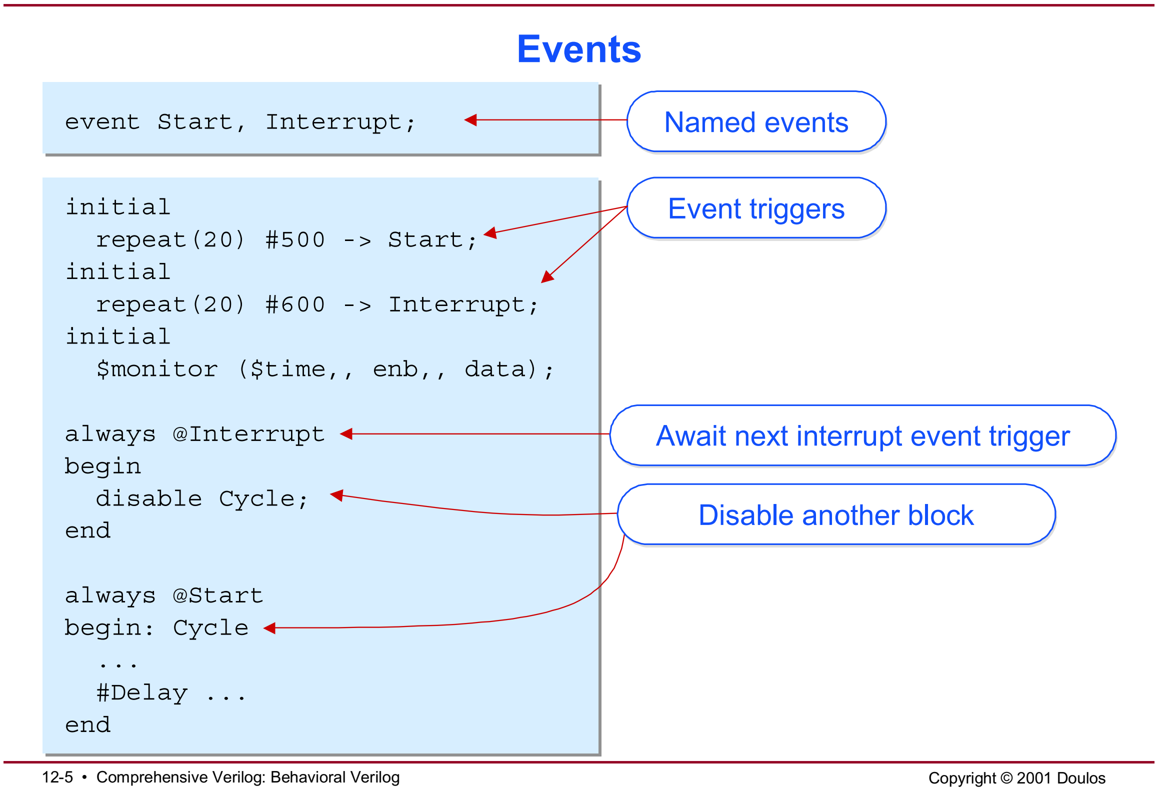
Where a procedural block contains multiple timing controls, then each timing control represents a different state in the control circuit being modelled. Since each timing control can be sensitive to a different variable or can even be a simple delay, it is possible to write very abstract descriptions a long way removed from the hardware implementation.

Wait

This example introduces the third kind of timing control; the wait timing control. Unlike the event control (@...) and the delay control (#...), the wait timing control is level sensitive. The procedure suspends until the expression in parenthesis becomes true. If the expression is already true, the procedure is not suspended but simply continues its execution past the wait timing control.

Note that task inputs should not be used in wait expressions, because a value is only copied into the input when the task is called. The task will not see a subsequent change in the wire or reg whose value was copied.

Events



Events

```
event Start, Interrupt;
```

```
initial
  repeat(20) #500 -> Start;
initial
  repeat(20) #600 -> Interrupt;
initial
  $monitor ($time,, enb,, data);

always @Interrupt
begin
  disable Cycle;
end

always @Start
begin: Cycle
  ...
  #Delay ...
end
```

Named events
Event triggers
Await next interrupt event trigger
Disable another block

12-5 • Comprehensive Verilog: Behavioral Verilog

Copyright © 2001 Doulos

Notes:

Verilog contains a rich set of language constructs for describing concurrency and synchronization. These constructs cannot be used at the register transfer level, but can be exploited when coding at higher levels of abstraction to express the design in a simpler and sometimes a more natural way.

Named event

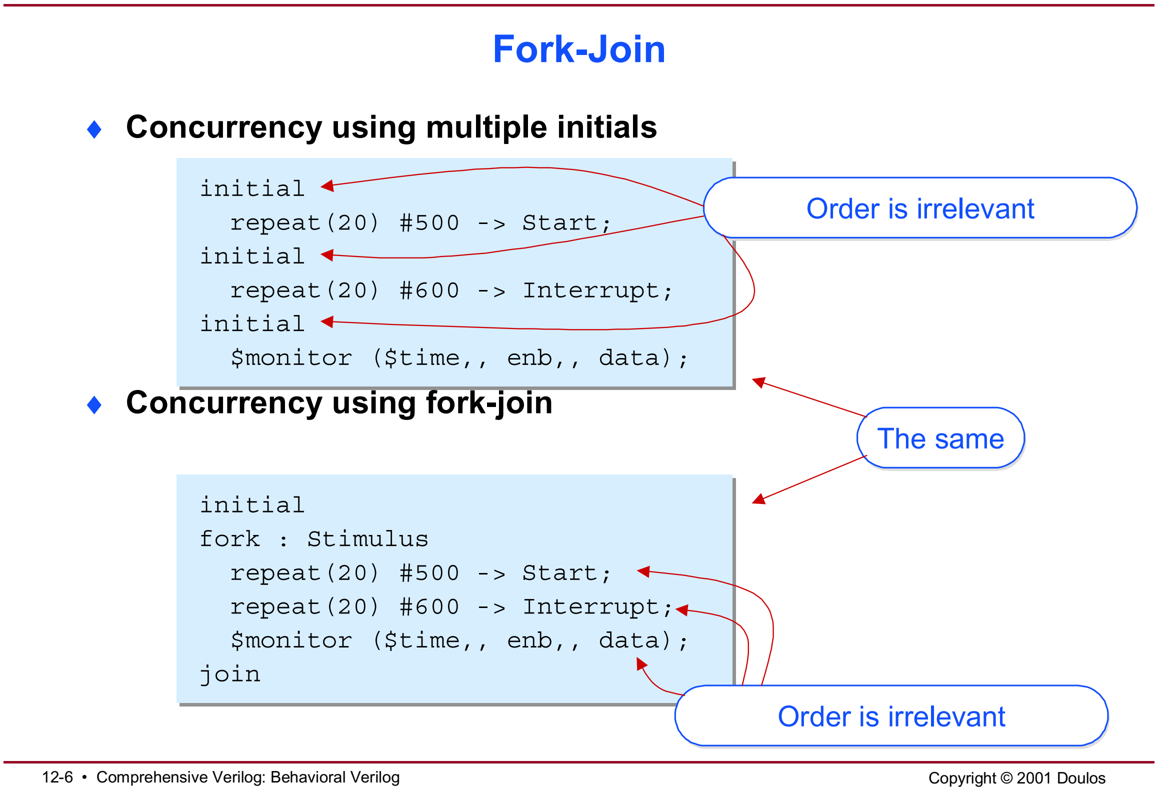
A named event is used to synchronize procedures; it can be thought of as a zero bit register that doesn't have a value - like a reg, only simpler! Instead of being assigned, a named event is triggered (or "made to happen") using the syntax

->name. The name of the event can appear in an event control, and thus the named event can cause other procedural blocks to wake up. Technically, a named event is another kind of data type. In the example opposite, the named events Start and Interrupt are generated at regular intervals by the initial statements, and are used to kick off the second and third blocks respectively.

Disable

This example also illustrates an the external disable. We have previously used the disable statement to jump out of an enclosing named block (e.g. to break a loop from within the loop). It can also be used to jump out of a named block that is part of another procedural block (an external disable as opposed to a self disable). This can provide a high level way to model an interrupt, as shown opposite, or a reset (see later). In practice, it can be quite difficult to model an interrupt with a disable statement, because no state information is saved.

Fork-Join



Fork-Join

- Concurrency using multiple initials

```
initial
  repeat(20) #500 -> Start;
initial
  repeat(20) #600 -> Interrupt;
initial
  $monitor ($time,, enb,, data);
```

Order is irrelevant

- Concurrency using fork-join

The same

```
initial
fork : Stimulus
  repeat(20) #500 -> Start;
  repeat(20) #600 -> Interrupt;
  $monitor ($time,, enb,, data);
join
```

Order is irrelevant

12-6 • Comprehensive Verilog: Behavioral Verilog

Notes:

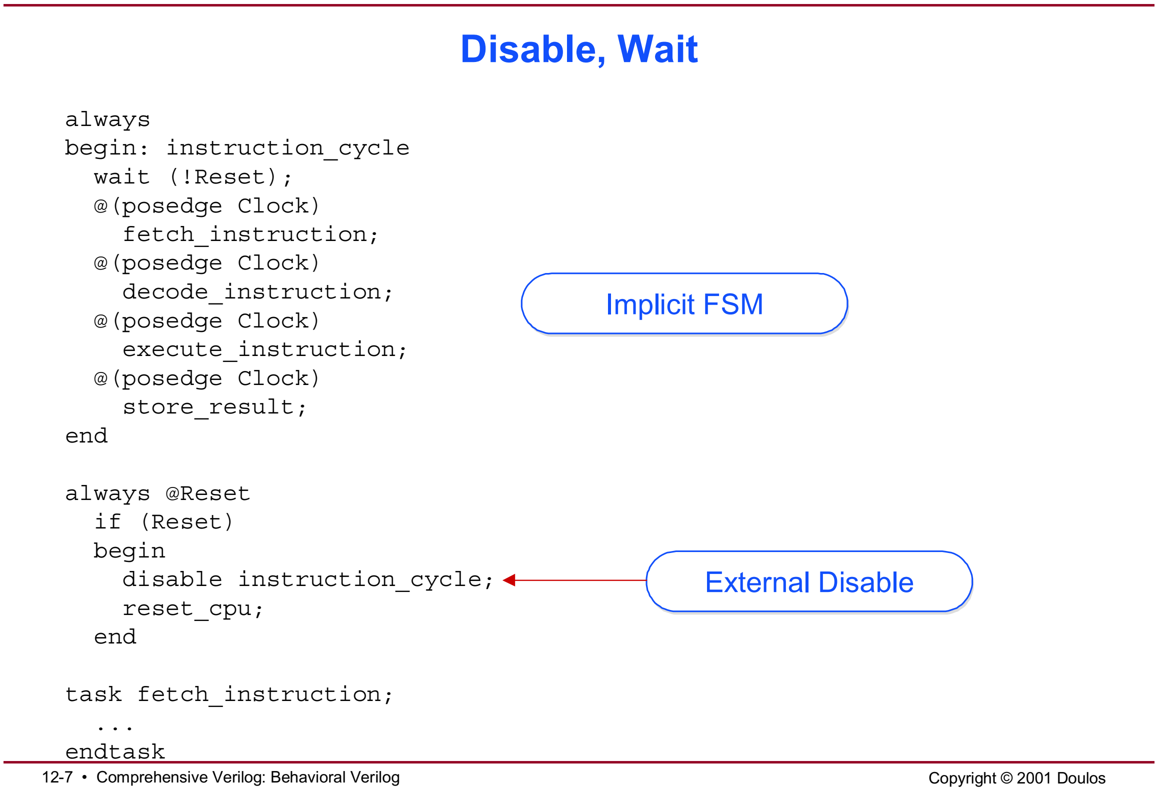
In the previous example, there were three concurrent initial statements. This can be modeled in an alternative way by using a single initial statement and enclosing the three procedural statements in a fork-join block instead of a begin-end block.

Fork-join

So far, we have grouped statements within a procedure using the begin-end keywords, causing the statements to be executed in sequence. The fork-join keywords are used to enclose statements which execute in parallel, the entire construct terminating only when the last enclosed statement terminates. Begin-end and fork-join constructs can be nested to any depth, making this a very powerful way to describe concurrency. It is often useful when generating stimulus, as shown here.

In this example, the two repeat loops run in parallel with events being triggered at times 500, 600, 1000, 1200, ... The $monitor process runs from the start of simulation.

Disable, Wait



Disable, Wait

```
always
begin: instruction_cycle
  wait (!Reset);
  @(posedge Clock)
    fetch_instruction;
  @(posedge Clock)
    decode_instruction;
  @(posedge Clock)
    execute_instruction;
  @(posedge Clock)
    store_result;
end

always @Reset
  if (Reset)
  begin
    disable instruction_cycle;
    reset_cpu;
  end

task fetch_instruction;
  ...
endtask
```

Implicit FSM

External Disable

12-7 • Comprehensive Verilog: Behavioral Verilog

Notes:

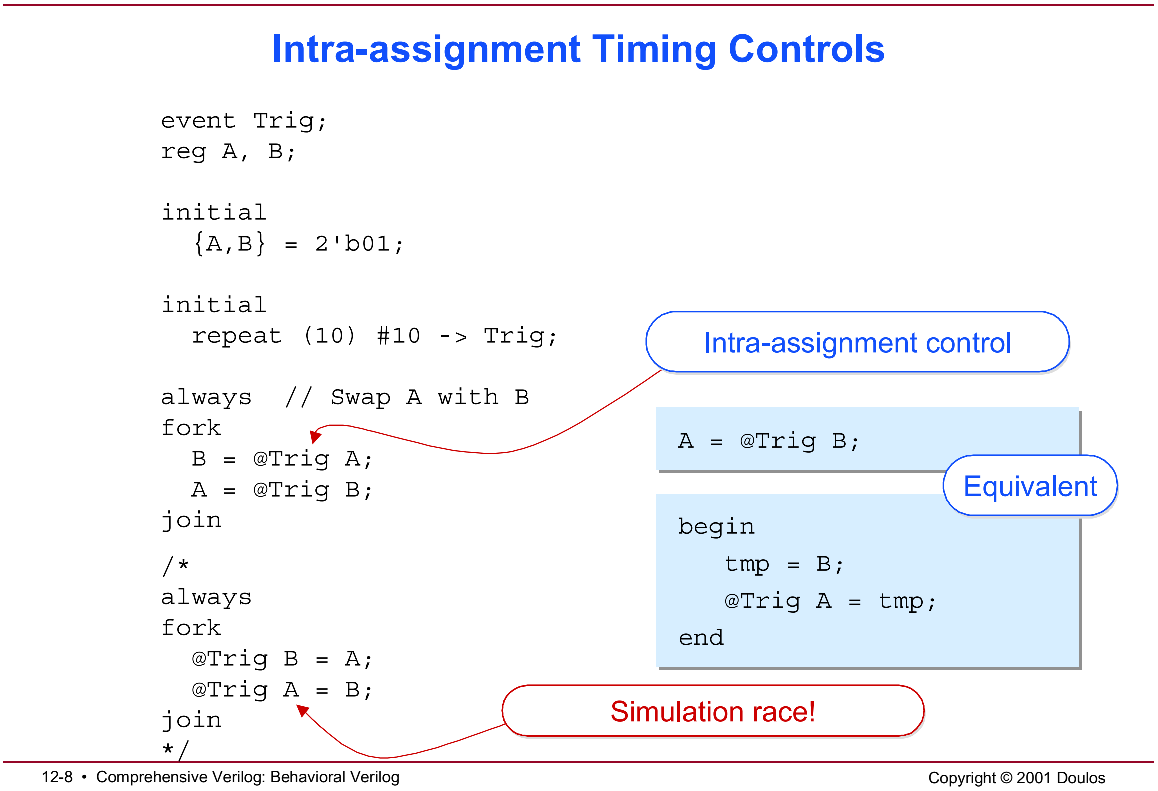
Here is another example illustrating the use of multiple timing controls, the wait timing control and the external disable.

The first procedural block describes the behavior of a synchronous circuit implementing a fetch-execute instruction cycle. On each tick of the Clock, the procedure performs part of the instruction cycle (by calling a task) then moves on to the next timing control.

The second procedural block describes the behavior of an asynchronous reset. When Reset is asserted, the instruction_cycle block is disabled (i.e. the currently executing instruction is interrupted), and the first block then waits at the top until the Reset is deasserted.

Note that some synthesis tools are able to synthesize descriptions with multiple clock events, like the top always statement. There is an implicit finite state machine in the description, so at least two flip-flops would be inferred from the four @(posedge Clock) statements. There is no corresponding reg for these flip-flops.

Intra-assignment Timing Controls



Intra-assignment Timing Controls

```
event Trig;
reg A, B;

initial
  {A,B} = 2'b01;

initial
  repeat (10) #10 -> Trig;

always  // Swap A with B
fork
  B = @Trig A;
  A = @Trig B;
join

/*
always
fork
  @Trig B = A;
  @Trig A = B;
join
*/
```

Intra-assignment control

```
A = @Trig B;
```

Equivalent

```
begin
   tmp = B;
   @Trig A = tmp;
end
```

Simulation race!

12-8 • Comprehensive Verilog: Behavioral Verilog

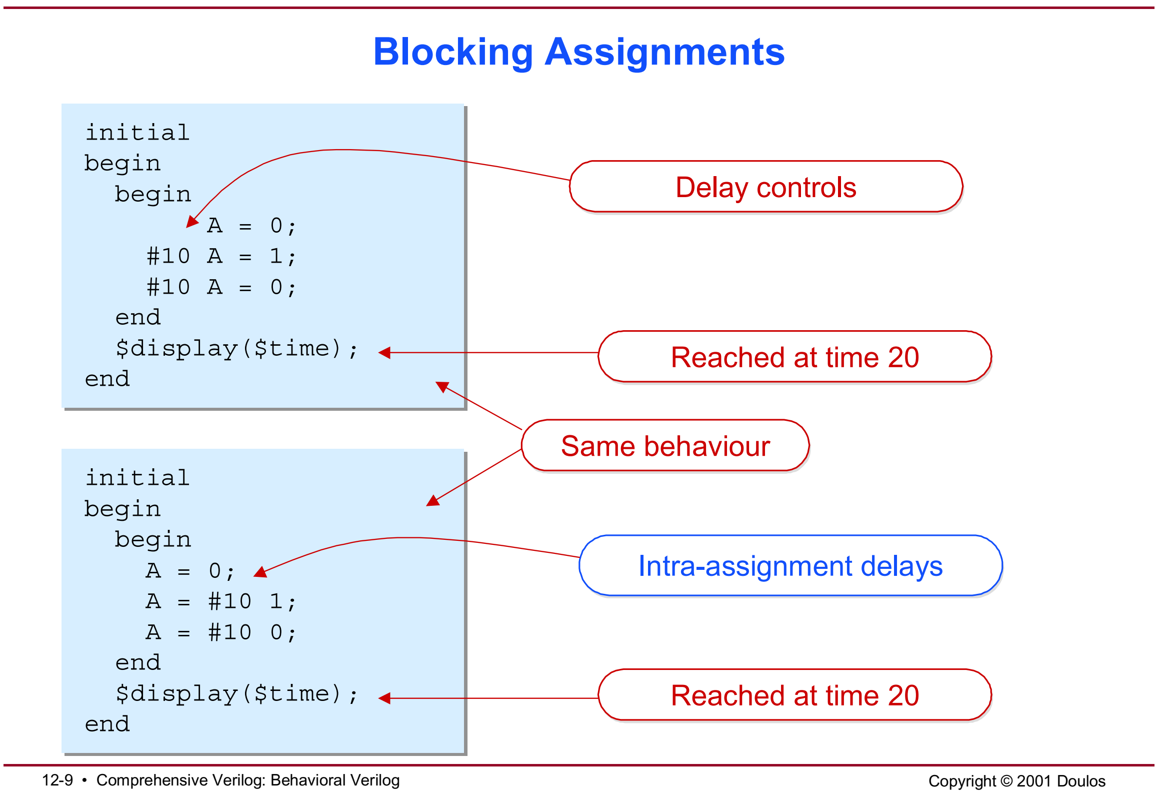
Notes:

In behavioral descriptions, it is sometimes convenient to write a timing control in the middle of a procedural assignment - an intra assignment timing control. This has the effect of suspending the procedural block after evaluating the expression on the right hand side of the assignment, but before updating the register on the left hand side. Effectively, the value being assigned is stored in a temporary variable while the procedure is suspended.

In the example opposite, the intra-assignment timing controls

(B = @Trig A; A = @Trig B;) are used to prevent a simulation race condition when swapping the values of A and B. The comment at the bottom of the example shows a bad fragment of code where there is a race between the two assignments.

Blocking Assignments



Blocking Assignments

```
initial
begin
  begin
        A = 0;
    #10 A = 1;
    #10 A = 0;
  end
  $display($time);
end
```

Delay controls

Reached at time 20

Same behaviour

```
initial
begin
  begin
    A = 0;
    A = #10 1;
    A = #10 0;
  end
  $display($time);
end
```

Intra-assignment delays

Reached at time 20

12-9 • Comprehensive Verilog: Behavioral Verilog

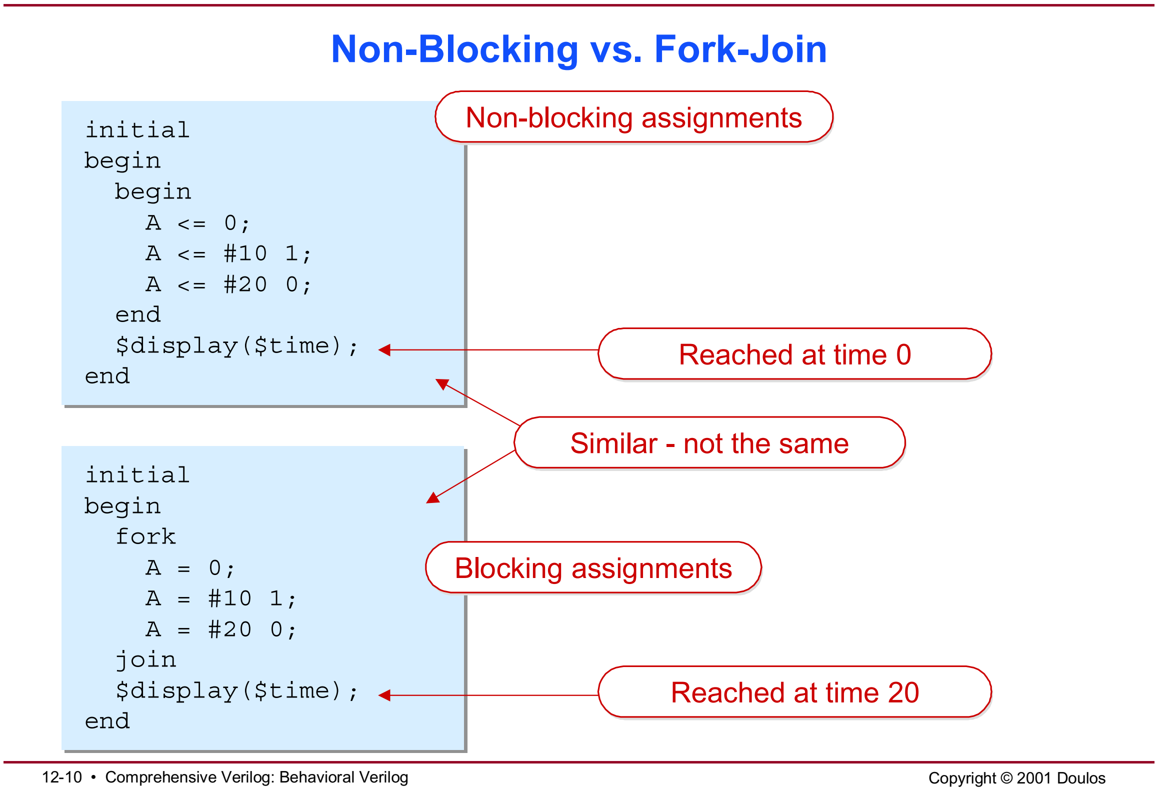
Copyright © 2001 Doulos

Notes:

Timing controls block the flow of control through a procedural block. For example, in a begin-end block with delays, the simulator waits for each delay before proceeding with the following statement. The begin-end completes when the final statement has been executed (after any delay).

When an intra-assignment timing control is used within a blocking assignment, the assignment also blocks the flow of control. This is illustrated by the example at the bottom, which has exactly the same simulation behavior as the example at the top of the page.

Non-Blocking vs. Fork-Join



Non-Blocking vs. Fork-Join

```
initial
begin
  begin
    A <= 0;
    A <= #10 1;
    A <= #20 0;
  end
  $display($time);
end
```

```
initial
begin
  fork
    A = 0;
    A = #10 1;
    A = #20 0;
  join
  $display($time);
end
```

Non-blocking assignments

Reached at time 0

Similar - not the same

Blocking assignments

Reached at time 20

12-10 • Comprehensive Verilog: Behavioral Verilog

Copyright © 2001 Doulos

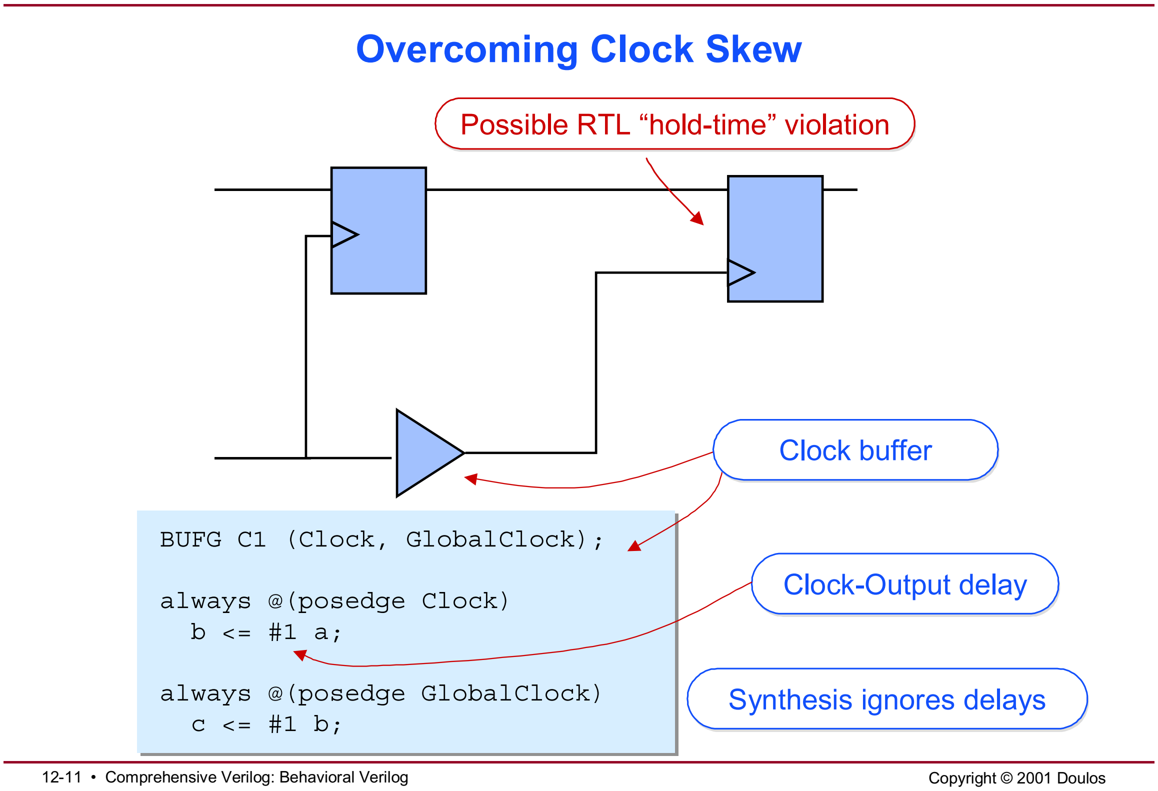
Notes:

A procedural assignment using the symbol <= is a non-blocking procedural assignment. The assignment schedules an event to occur under the control of the timing control, but the execution of the procedure is not blocked.

Thus the begin-end block opposite is executed and completes at time 0, and schedules A to change at times 0, 10 and 20.

There are similarities between using non-blocking assignments in a begin-end and blocking statements in a fork-join statement, though they are certainly not equivalent. The difference is that blocking assignments still block in a fork-join. All the statements are executed in parallel, so they don't block each other, but the simulator won't exit from the fork-join until all the blocking assignments have completed. In this example, that will be at time 20, when the third assignment to A completes.

Overcoming Clock Skew



Overcoming Clock Skew

Possible RTL “hold-time” violation

Clock buffer

Clock-Output delay

Synthesis ignores delays

```
BUFG C1 (Clock, GlobalClock);

always @(posedge Clock)
  b <= #1 a;

always @(posedge GlobalClock)
  c <= #1 b;
```

12-11 • Comprehensive Verilog: Behavioral Verilog

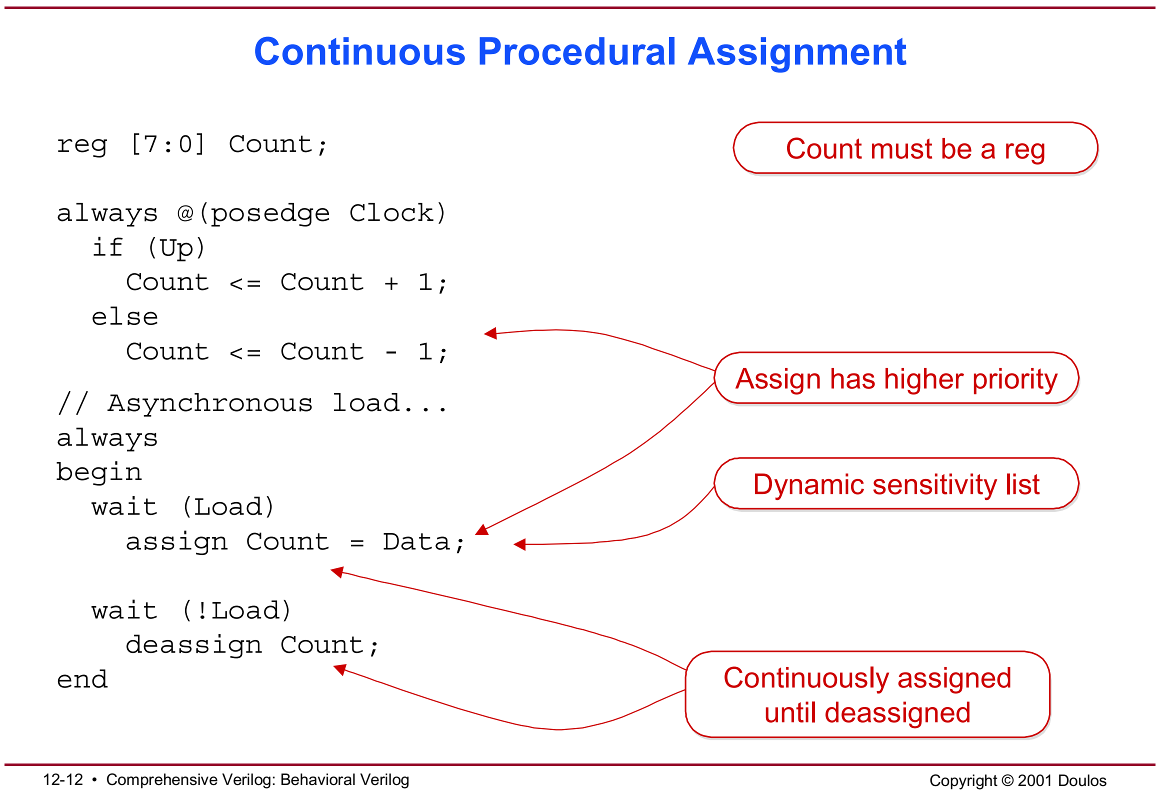
Notes:

There is one important application of intra-assignment delays in RTL design. It concerns the problem of clock skew.

Clock skew can be a problem in real hardware, causing hold time violations at the flip-flops. It can also be a problem in RTL code! The problem occurs when there are delays (even zero-length delays) in the clock net. This might happen when clock buffers are instanced explicitly in a design, rather than being inferred by the synthesis tool. (Instancing clock buffers may sometimes be necessary in both ASIC design and FPGA design.)

The problem of RTL clock skew can be solved using intra-assignment delays, as shown. The delay must be bigger than any clock delays. In effect, a nominal clock to output delay is being modeled.

Continuous Procedural Assignment



Continuous Procedural Assignment

```
reg [7:0] Count;

always @(posedge Clock)
  if (Up)
    Count <= Count + 1;
  else
    Count <= Count - 1;

// Asynchronous load...
always
begin
  wait (Load)
    assign Count = Data;

  wait (!Load)
    deassign Count;
end
```

Count must be a reg

Assign has higher priority

Dynamic sensitivity list

Continuously assigned until deassigned

12-12 • Comprehensive Verilog: Behavioral Verilog

Copyright © 2001 Doulos

Notes:

The continuous procedural assignment is another very powerful statement for controlling concurrent access to Verilog registers.

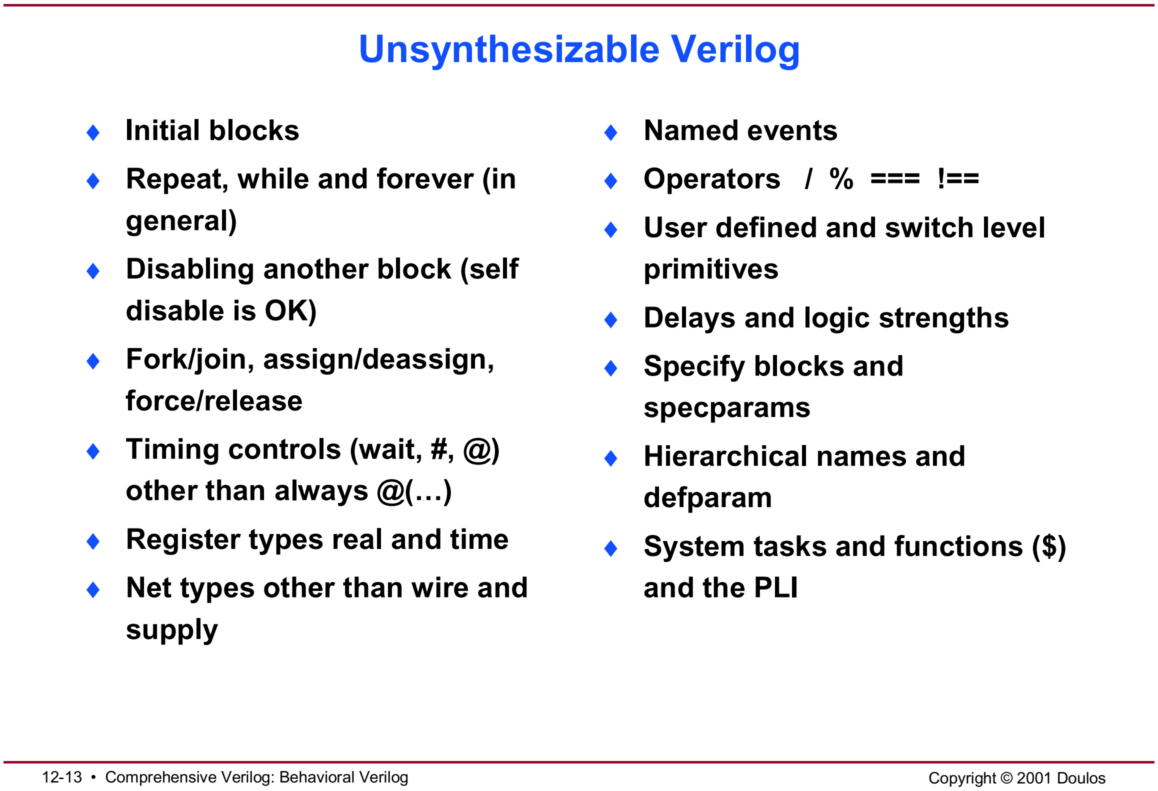
The continuous procedural assignment is a kind of procedural assignment i.e. an assignment to a register written within a procedural block. It differs in two very significant ways from the usual procedural assignment.

Firstly, it has a higher priority than the usual procedural assignment. In the example above, the assignment assign Count = Data; takes priority over the procedural assignments in the first procedural block when the asynchronous load is active.

Secondly, it is continuous, i.e. the register on the left hand side is continuously updated with the value of the expression on the right hand side. When the assignment is executed, a dynamic sensitivity list is created from the names of nets and registers on the right hand side, and the assignment is triggered whenever an event occurs on an item in the sensitivity list.

The register on the left hand side stays continuously assigned until another continuous procedural assignment is made to that register, or until the register is explicitly deassigned as shown.

Unsynthesizable Verilog



Unsynthesizable Verilog

- Initial blocks
- Repeat, while and forever (in general)
- Disabling another block (self disable is OK)
- Fork/join, assign/deassign, force/release
- Timing controls (wait, #, @) other than always @(…)
- Register types real and time
- Net types other than wire and supply
- Named events
- Operators / % === !==
- User defined and switch level primitives
- Delays and logic strengths
- Specify blocks and specparams
- Hierarchical names and defparam
- System tasks and functions ($) and the PLI

12-13 • Comprehensive Verilog: Behavioral Verilog

Notes:

The Verilog HDL statements that are not synthesizable are shown opposite. Unsynthesizable means that the construct is not supported universally. Some constructs mentioned here are not supported by any tools. Others are supported by some tools but not by others. In either case, these statements should not be used for synthesis, as, at best, the code will not be easily portable between tools.

Simply knowing which statements can be synthesized and which not is only part of the story. It is important also to know how to use the synthesizable constructs to achieve the desired result in a predictable way.

Unbounded Verilog constructs are not synthesizable...

Indefinite loops

Indefinite loops (such as while loops) are unbounded because the number of loop iterations is not fixed, so the synthesis tool cannot create hardware for each loop iteration. An exception is loops explicitly synchronized to a clock edge in an implicit FSM description.

/ %

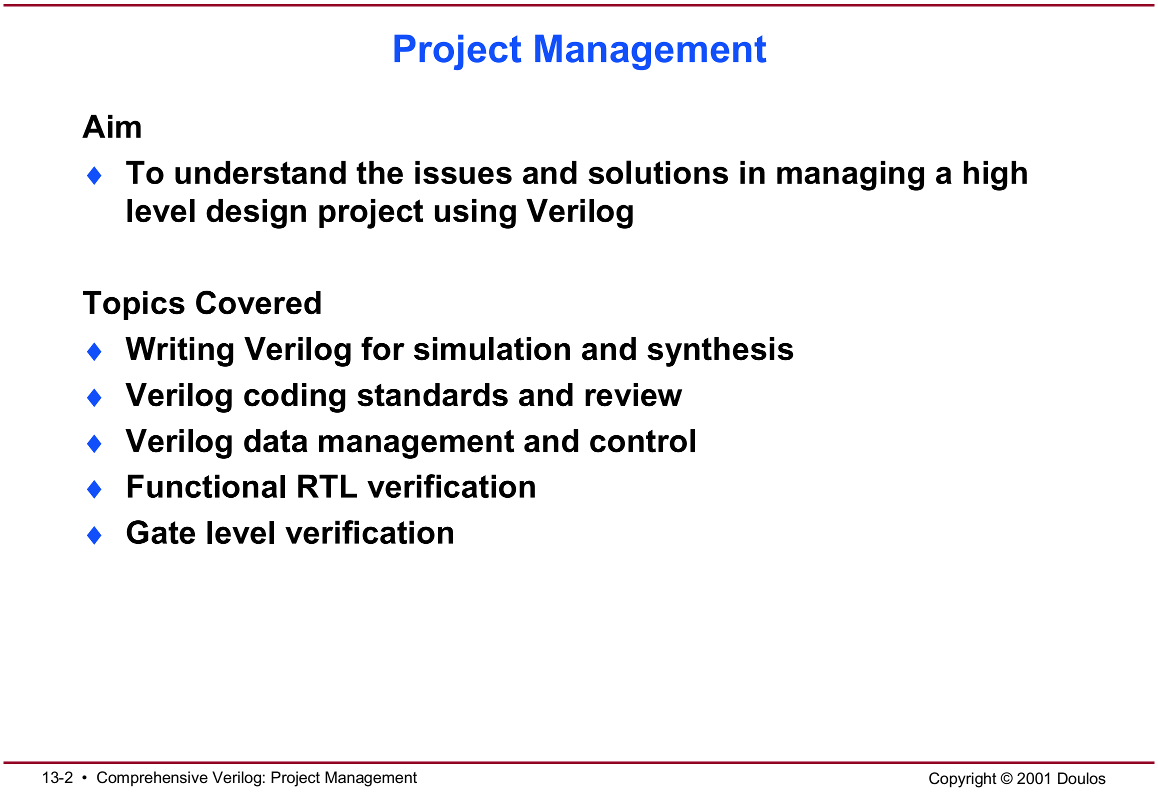
The division and remainder operators are unbounded, because the number of hardware clock cycles required to produce a result is unknown.

real

Real registers are unbounded, because the accuracy, range and rounding of the representation of real numbers is undefined.

Module 13
Project Management

Project Management



Project Management

Aim

- To understand the issues and solutions in managing a high level design project using Verilog

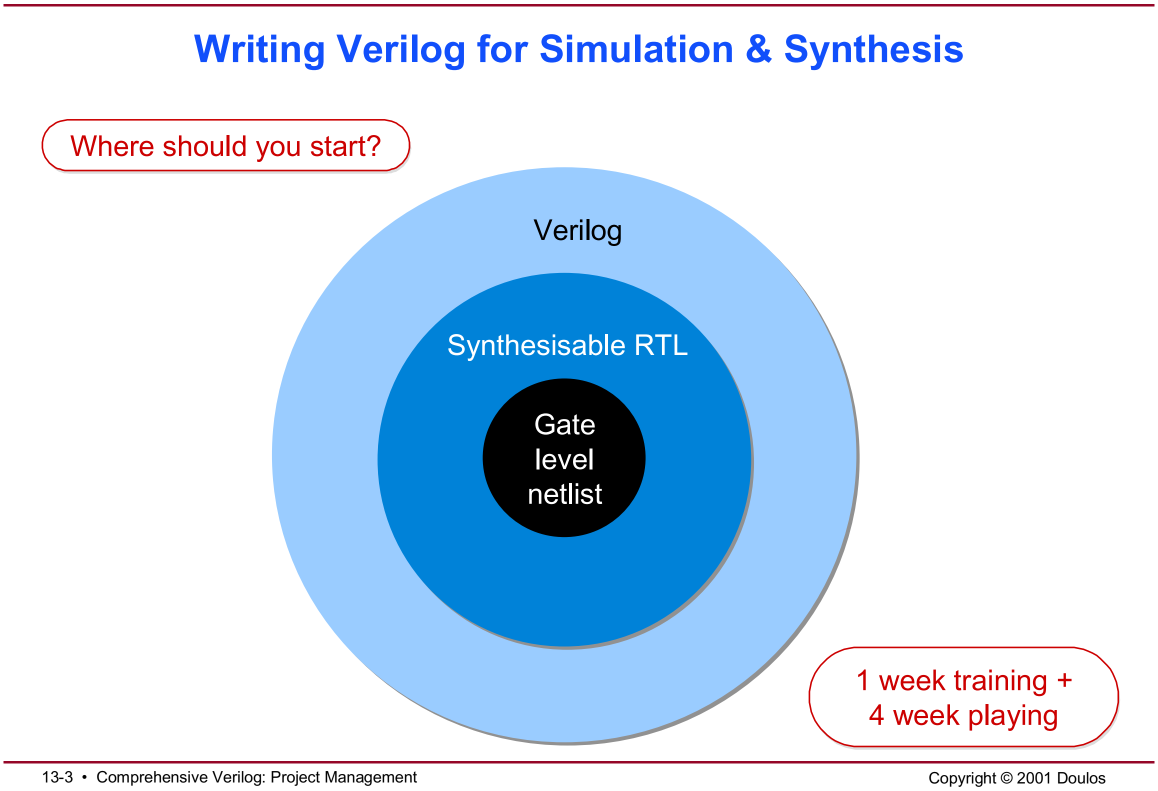
Topics Covered

- Writing Verilog for simulation and synthesis
- Verilog coding standards and review
- Verilog data management and control
- Functional RTL verification
- Gate level verification

13-2 • Comprehensive Verilog: Project Management

Notes:

Writing Verilog for Simulation & Synthesis



Writing Verilog for Simulation & Synthesis

Where should you start?

1 week training +
4 week playing

13-3 • Comprehensive Verilog: Project Management

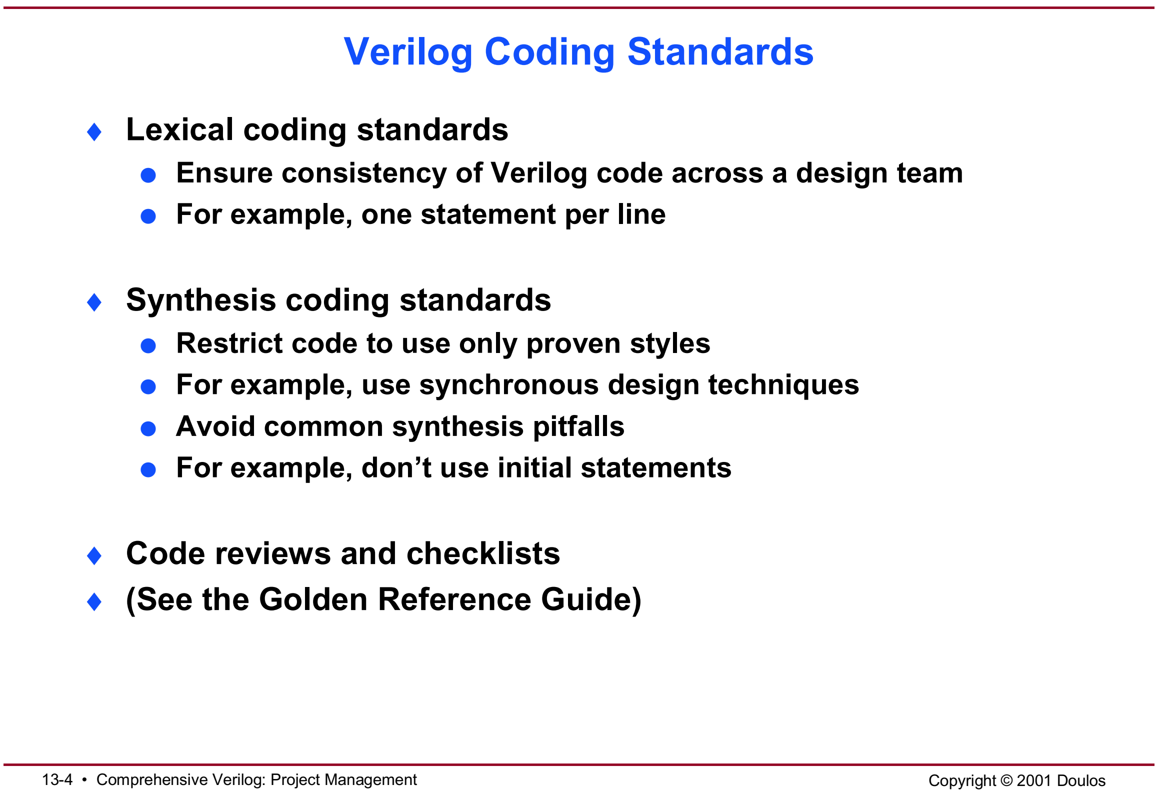
Notes:

You can think of the Verilog language as organized into a number of proper subsets. Within the outer circle, the entire language as defined in the 1364 LRM can be simulated with any Verilog simulator, so it is appropriate for abstract behavioral modeling. A proper subset of Verilog, the RTL subset, is appropriate for designing hardware at the register transfer level and can be synthesized down to gate level by a synthesis tool. Within the inner circle, a proper subset of RTL Verilog is a gate level netlist using logical primitives available in the target implementation technology.

Where is the right point to start coding Verilog? There are two common mistakes. The first mistake is to attempt to do detailed hardware design by writing Verilog at too high a level of abstraction (the outer circle). The result is that the Verilog code is either not synthesizable, or else generates a very inefficient hardware implementation. The second mistake is to attempt to design hardware with a Verilog style that lies inside or close to the inner circle. By coding at a level that lies close to the hardware you can maximize your control over the details of the hardware design, but sacrifice abstraction and productivity in the process.

These mistakes can only be overcome by in-depth training and by experience. For example, a one week training course followed by four weeks of playing with the language and tools might be an appropriate mix.

Verilog Coding Standards



Verilog Coding Standards

- Lexical coding standards
  - Ensure consistency of Verilog code across a design team
  - For example, one statement per line
- Synthesis coding standards
  - Restrict code to use only proven styles
  - For example, use synchronous design techniques
  - Avoid common synthesis pitfalls
  - For example, don't use initial statements
- Code reviews and checklists
- (See the Golden Reference Guide)

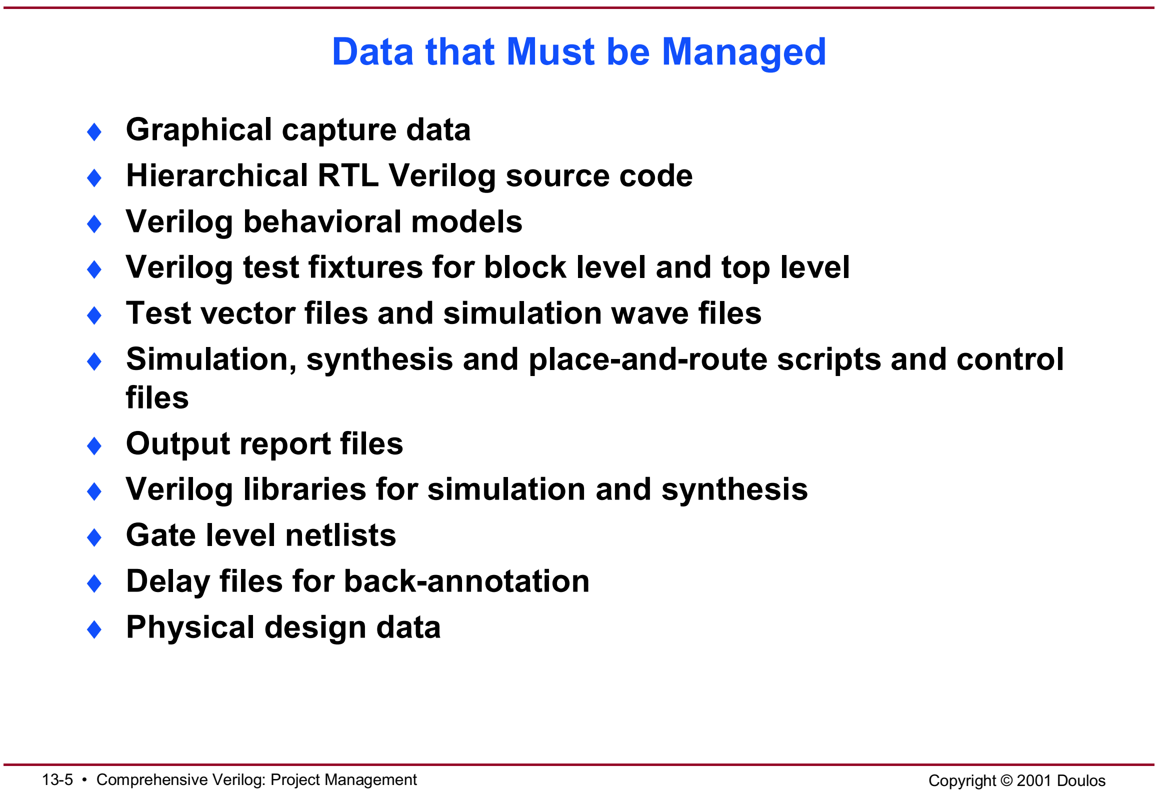
13-4 • Comprehensive Verilog: Project Management

Notes:

Verilog coding standards come in two kinds. Lexical coding standards specify the choices to be made in laying out the source code and choosing the case of identifiers, naming conventions, inserting comments and so on. Lexical coding standards are important in enforcing consistency of layout and style across a design team. Synthesis coding standards specify the choice of hardware design style and coding style to get the Verilog successfully through the RTL synthesis tool. Synthesis coding standards avoid everyone having to re-invent a successful coding strategy from scratch, and increase the likelihood of achieving a high quality design on the first iteration.

Code reviews are a good idea, especially in the early days when a team is new to Verilog-based designs. Code reviews can help to reinforce the lessons learned during formal training sessions.

Data that Must be Managed



Data that Must be Managed

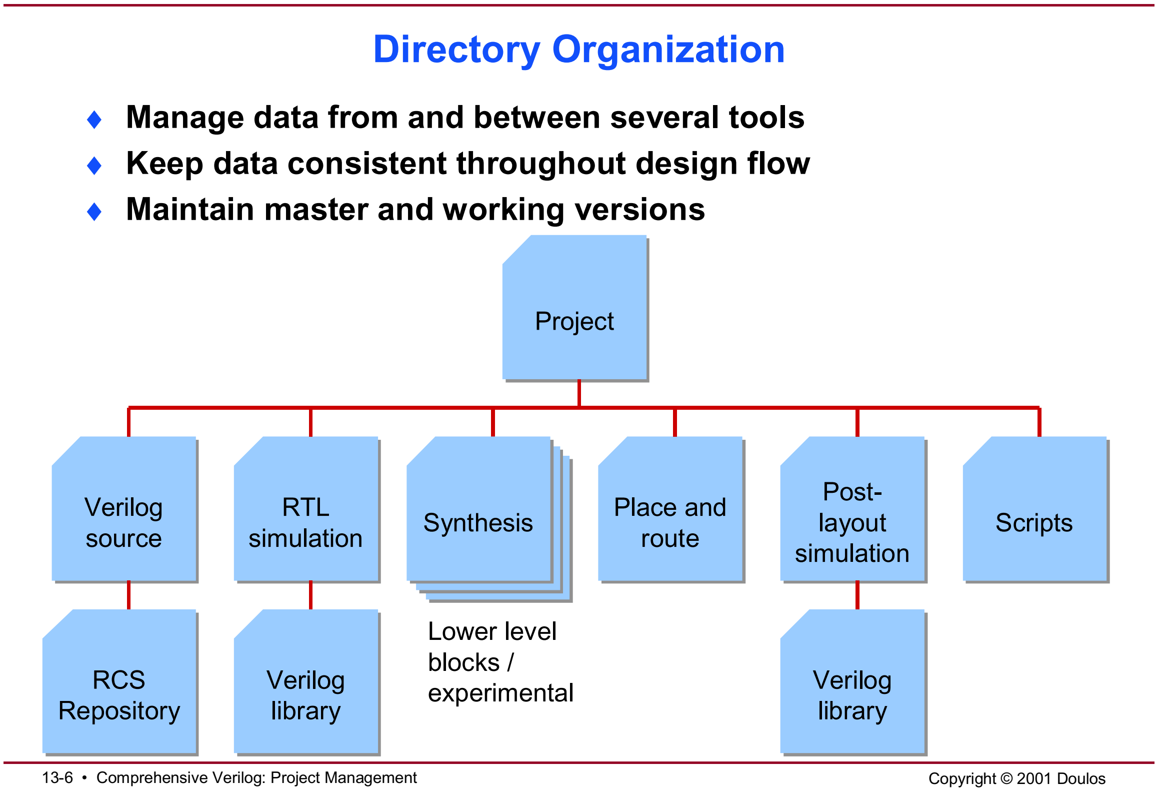
- Graphical capture data
- Hierarchical RTL Verilog source code
- Verilog behavioral models
- Verilog test fixtures for block level and top level
- Test vector files and simulation wave files
- Simulation, synthesis and place-and-route scripts and control files
- Output report files
- Verilog libraries for simulation and synthesis
- Gate level netlists
- Delay files for back-annotation
- Physical design data

13-5 • Comprehensive Verilog: Project Management

Notes:

Organizing the Verilog source files is only the first step towards a structured design flow; there are many more kinds of files that need to be controlled and maintained, as shown by the list.

Directory Organization



Directory Organization

- Manage data from and between several tools
- Keep data consistent throughout design flow
- Maintain master and working versions

13-6 • Comprehensive Verilog: Project Management

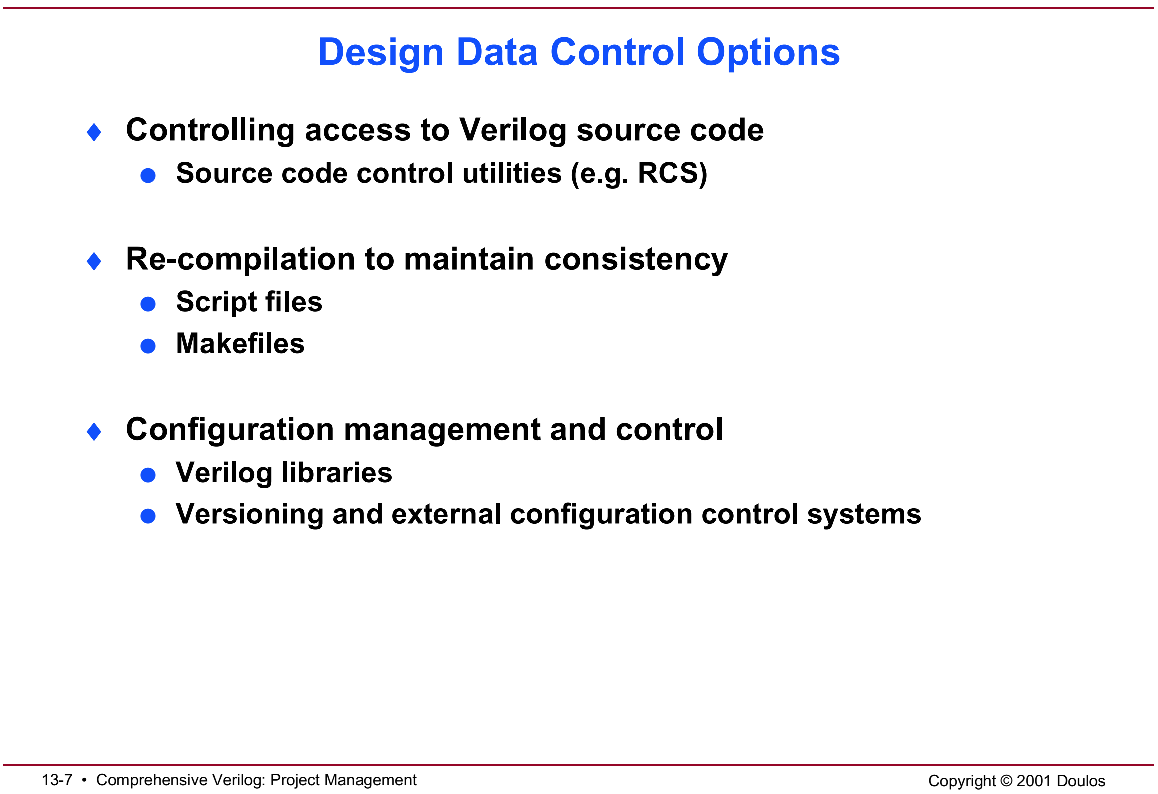
Copyright © 2001 Doulos

Notes:

Design data from several different tools will need to be managed. The data must be kept consistent throughout the design flow. Support must be provided for hierarchical verification, which may require simulating or synthesizing a lower level block rather than the entire design, and for block-level experimentation, which may require the creation of a temporary version of all or part of the design. Back-copies of old versions may need to be kept.

It is important to define a directory organization at the outset of a project, and stick to it. A possible organization is suggested above. The actual organization used will depend to a large extent on the tools and procedures being used.

Design Data Control Options



Design Data Control Options

- Controlling access to Verilog source code
  - Source code control utilities (e.g. RCS)
- Re-compilation to maintain consistency
  - Script files
  - Makefiles
- Configuration management and control
  - Verilog libraries
  - Versioning and external configuration control systems

13-7 • Comprehensive Verilog: Project Management

Notes:

There are a number of distinct issues in managing and controlling the source data:

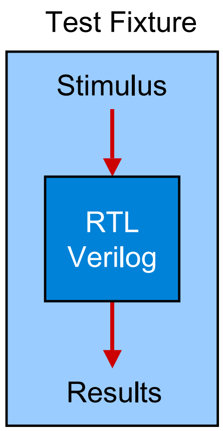
- Controlling access to the source code so that two parties cannot try to edit the same file at once, and so that master data can be protected from change.
- Supporting the automatic re-compilation of the Verilog source code when a design unit is changed (where tools require files to be compiled separately).
- Configuration management and control so that a particular version of the complete design can be identified, frozen, and re-created at a later date.

Source code control is sometimes provided as part of the functionality of a Design Automation Framework. Otherwise, general-purpose source code control utilities can be used, or custom in-house utilities developed at low cost.

Most Verilog tool sets have a re-compilation facility, although sometimes with less than the required degree of sophistication. A practical alternative is to build an in-house re-compilation utility based on Unix make. Script files are important throughout the design process to ensure that the process is repeatable.

Functional Verification

Functional Verification



- Extensive functional simulation is essential
- Depends on the quality of the test fixture
- Be methodical
- Create a verification plan
- Review the test bench
- Maybe use code coverage analysis tools
- Different testing methods find different kinds of bugs

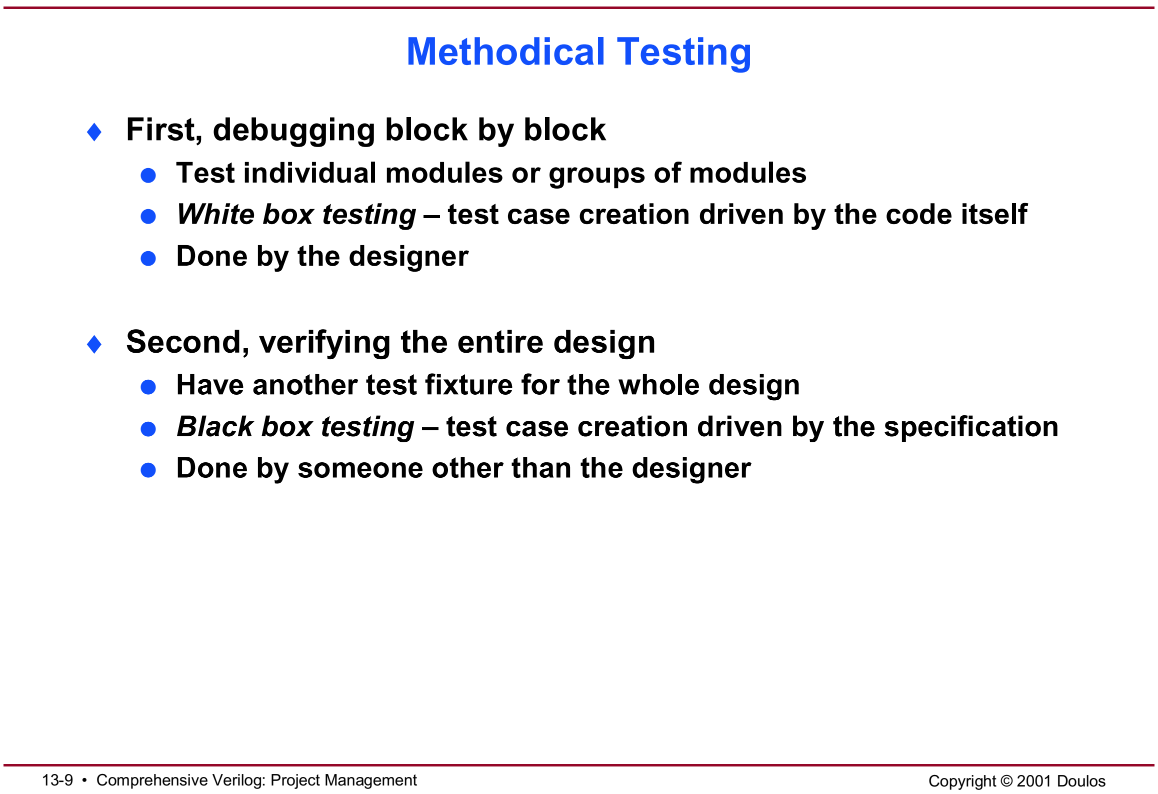
Notes:

The High Level Design methodology depends on validating the functionality of the design at each level of abstraction before refining the design to lower levels of abstraction. In practice, this is achieved through extensive simulation. However, validation through simulation is reliant on the quality of the test cases, and so writing (high level) test cases becomes one of the major creative engineering tasks.

RTL simulation is much faster than gate level simulation, and experience has shown that this speed up is best exploited by simulating more test cases in the time available, not shortening the time allotted to simulation.

Software engineers have learned from long and bitter experience that proving code through testing is not easy. A structured and methodical approach is necessary.

Methodical Testing



Methodical Testing

- First, debugging block by block
  - Test individual modules or groups of modules
  - *White box testing* – test case creation driven by the code itself
  - Done by the designer
- Second, verifying the entire design
  - Have another test fixture for the whole design
  - *Black box testing* – test case creation driven by the specification
  - Done by someone other than the designer

13-9 • Comprehensive Verilog: Project Management

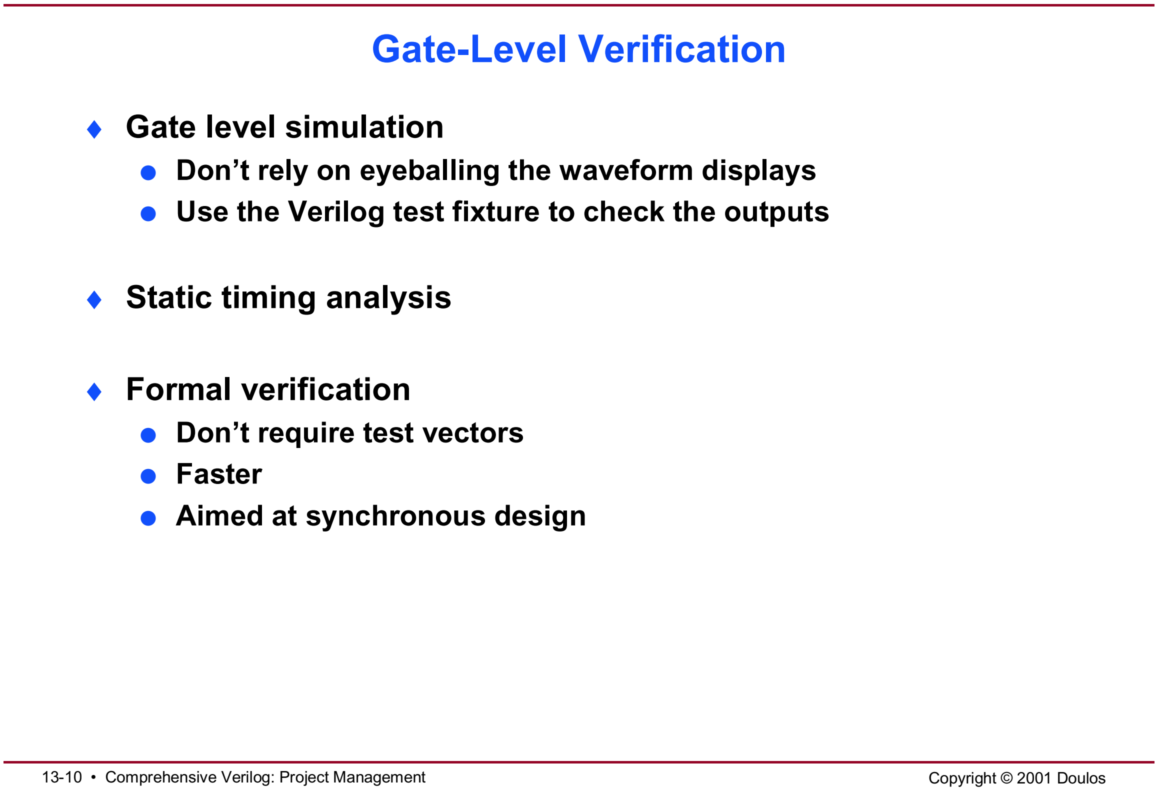
Notes:

When testing Verilog code, it is important to distinguish debugging from testing. The aim of debugging is to find bugs in the source code, whereas the aim of testing is to prove the absence of bugs.

Debugging is done by the designer himself on a block-by-block basis, using temporary test fixtures as scaffolding. It's best to use so-called white box testing, where you look at the fine details of the code itself both to choose test cases and to observe the effects of test cases (using a debugger).

Testing is done on the entire design. It's best to use so-called black box testing, where the internal details of the code are hidden during test case generation and output checking. Test case creation should be driven from the specification, and it is beneficial to have different parties develop the hardware design and the test cases to ensure compliance with the specification.

Gate-Level Verification



Gate-Level Verification

- Gate level simulation
  - Don’t rely on eyeballing the waveform displays
  - Use the Verilog test fixture to check the outputs
- Static timing analysis
- Formal verification
  - Don’t require test vectors
  - Faster
  - Aimed at synchronous design

13-10 • Comprehensive Verilog: Project Management

Notes:

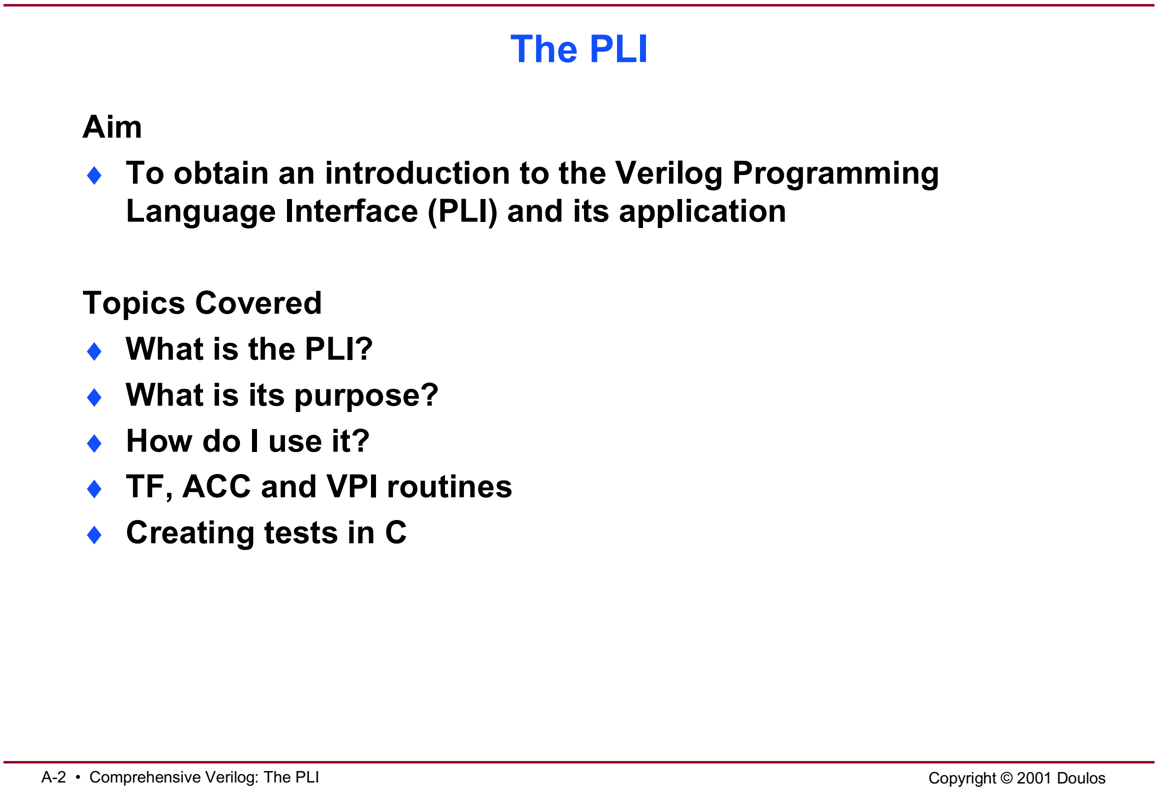
After synthesis and/or place-and-route, the final functional verification step is to compare the gate-level simulation results with the RTL simulation results, usually by putting the gate-level netlist back into the RTL test fixture. The thing not to do is to rely on using the waveform display to confirm that the simulation results are still correct. It is notoriously easy to miss errors when scanning long waveforms by eye. You should always automate the comparison between simulation results, as discussed in the previous section.

Gate-level simulation is not the only method of verification. Static timing analysis is rightly increasing in popularity as a method of verifying the timing, because it does not rely on the completeness of a set of simulation test cases, and runs much faster than simulation. The downside of static timing analysis is that it works on the assumption of synchronous design. If you have to deal with asynchronous interfaces, then static timing analysis will not give you the whole truth.

Formal verification is also growing in popularity as an alternative to gate-level simulation in some situations. For example, equivalence checking can be used to prove the equivalence of two netlists before and after inserting test logic or clock trees, or before and after making some minor modification to a netlist by hand.

Appendix A
The PLI

The PLI



The PLI

Aim

- To obtain an introduction to the Verilog Programming Language Interface (PLI) and its application

Topics Covered

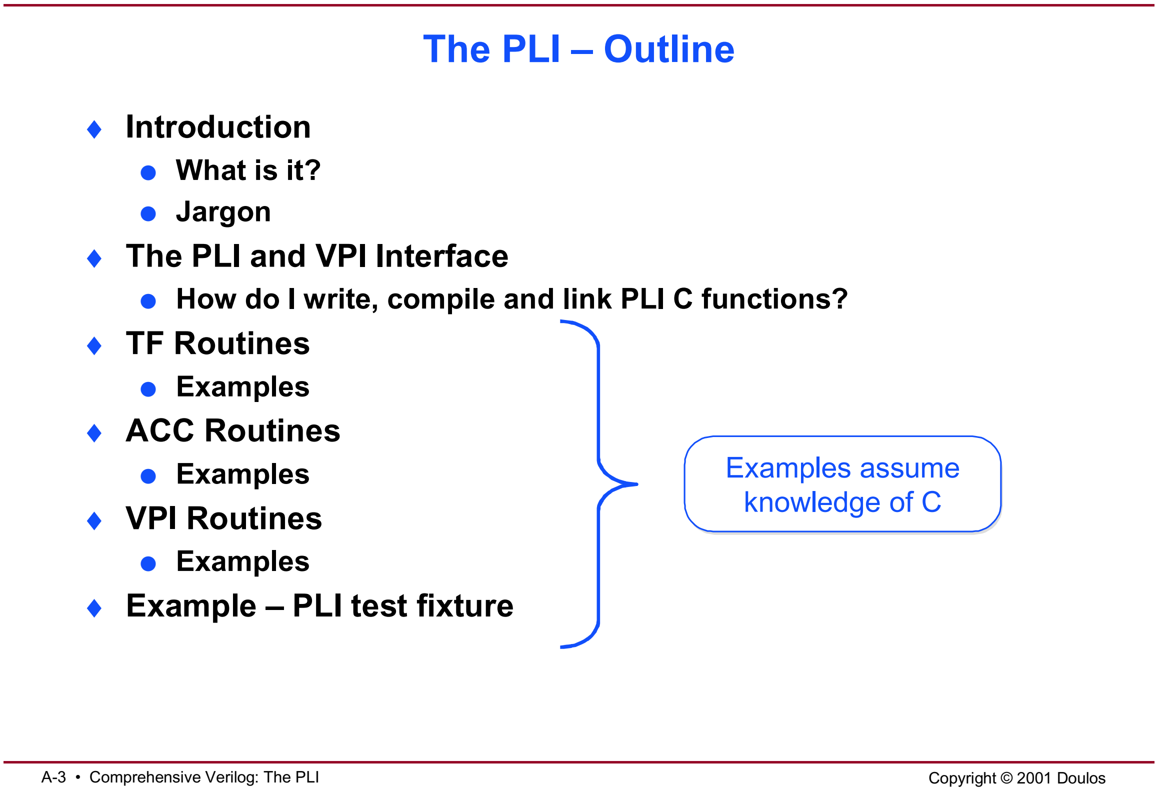
- What is the PLI?
- What is its purpose?
- How do I use it?
- TF, ACC and VPI routines
- Creating tests in C

A-2 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

The PLI – Outline



The PLI – Outline

- Introduction
  - What is it?
  - Jargon
- The PLI and VPI Interface
  - How do I write, compile and link PLI C functions?
- TF Routines
  - Examples
- ACC Routines
  - Examples
- VPI Routines
  - Examples
- Example – PLI test fixture

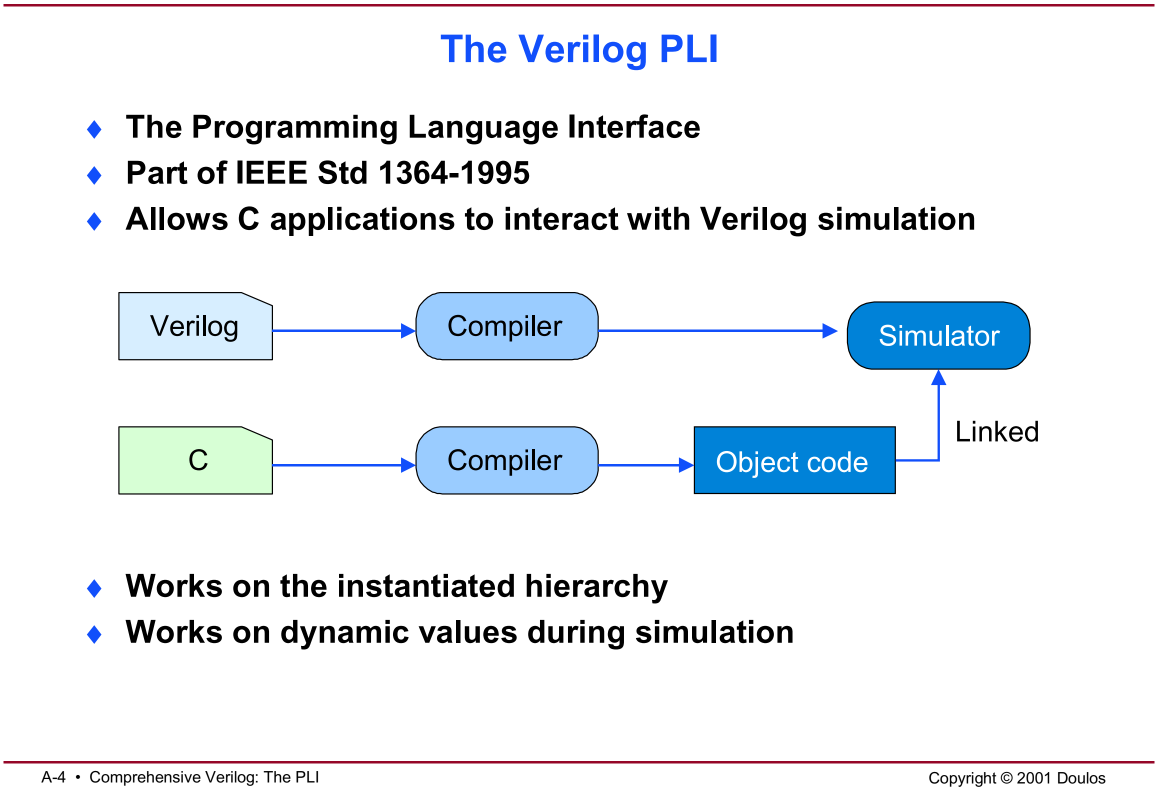
Examples assume knowledge of C

A-3 • Comprehensive Verilog: The PLI

Notes:

Above is the outline of this section. To understand most of the examples, you need to have a working knowledge of the C programming language.

The Verilog PLI



The Verilog PLI

- The Programming Language Interface
- Part of IEEE Std 1364-1995
- Allows C applications to interact with Verilog simulation

- Works on the instantiated hierarchy
- Works on dynamic values during simulation

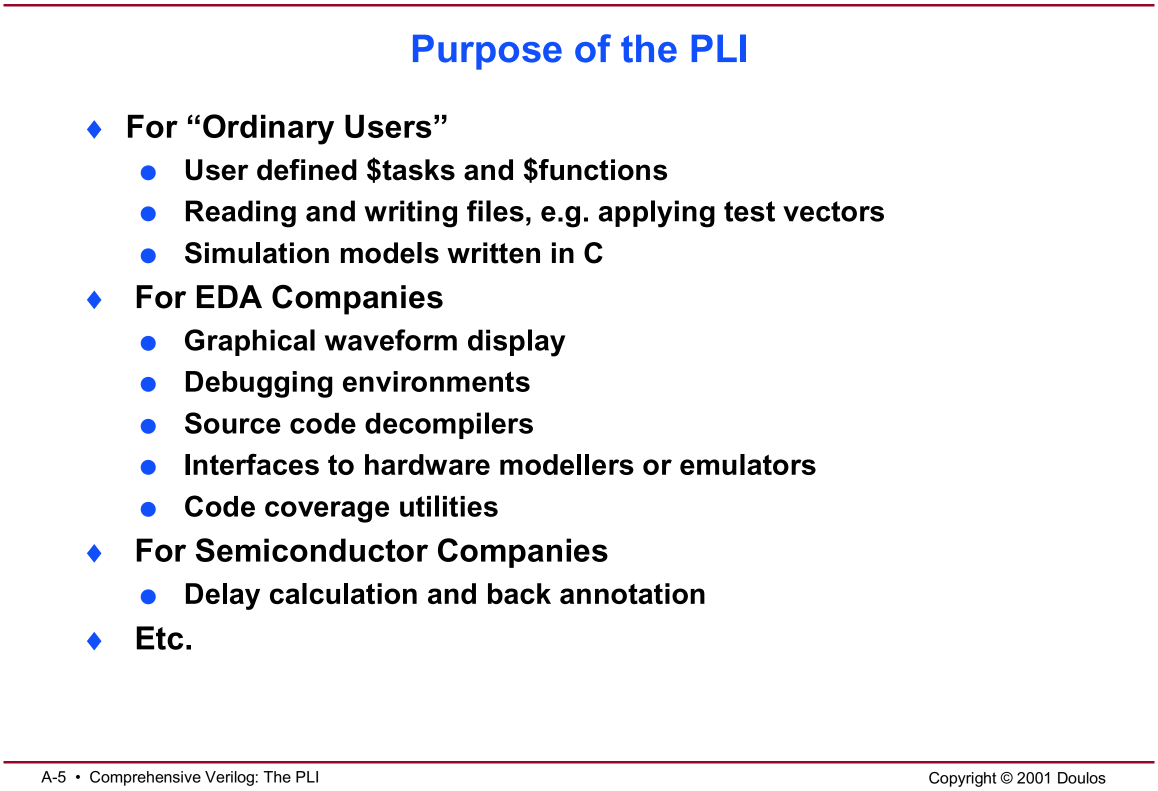
A-4 • Comprehensive Verilog: The PLI

Notes:

The Verilog Programming Language Interface (PLI) is part of the IEEE 1364 Verilog standard. It allows functions written in C to be called from and to interact with a Verilog simulation.

Using the PLI, you can access the data structures created from the hierarchy of instances in the design being simulated, including the structural and specify block delays and the values of wires and registers as they change during the simulation.

Purpose of the PLI



Purpose of the PLI

- For “Ordinary Users”
  - User defined $tasks and $functions
  - Reading and writing files, e.g. applying test vectors
  - Simulation models written in C
- For EDA Companies
  - Graphical waveform display
  - Debugging environments
  - Source code decompilers
  - Interfaces to hardware modellers or emulators
  - Code coverage utilities
- For Semiconductor Companies
  - Delay calculation and back annotation
- Etc.

A-5 • Comprehensive Verilog: The PLI

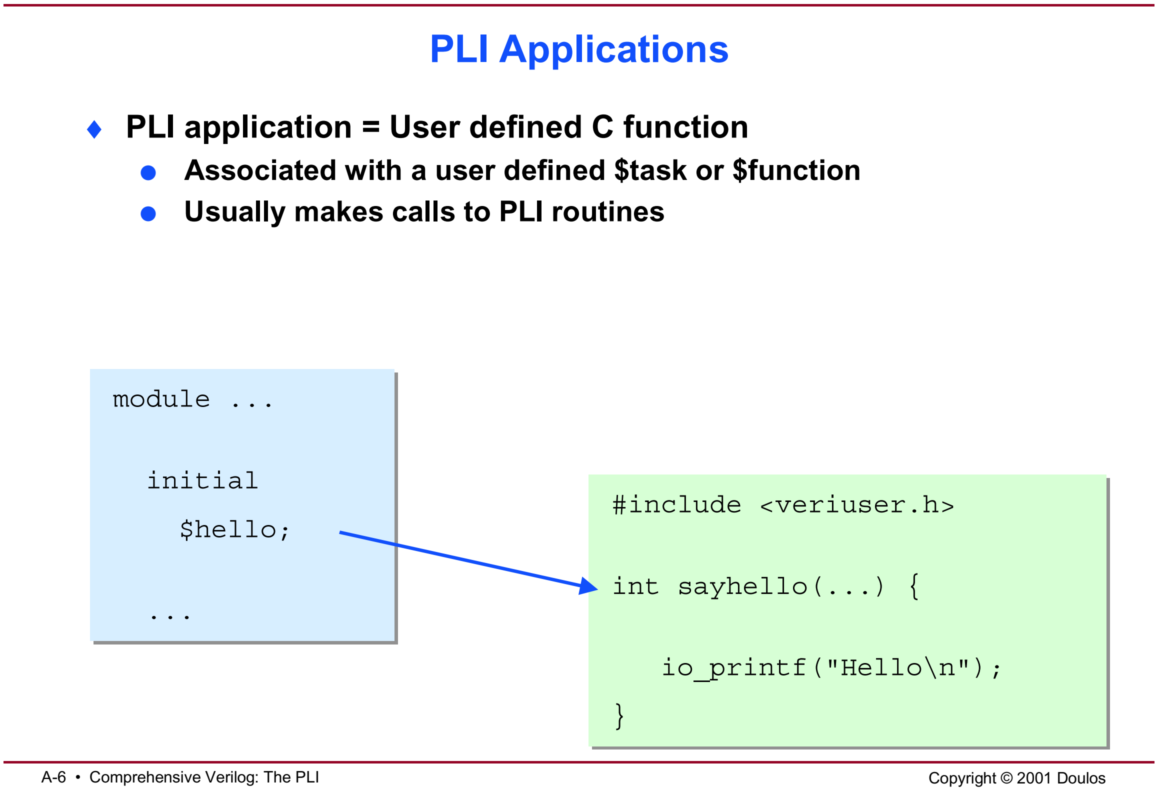
Notes:

The PLI is used to extend the capabilities of the Verilog Hardware Description Language and of the Verilog simulator being used.

Examples of PLI applications include user-defined utilities, for example to read and write text files; custom delay calculation and back-annotation; interfacing graphical waveform display utilities, and providing graphical debugging environments. The PLI can also be used as an interface to allow models written in C or other hardware description languages such as VHDL to co-simulate with Verilog modules.

Because the PLI is part of the Verilog standard, applications such as these can easily be ported to work with any Verilog simulator that complies with the Verilog standard.

PLI Applications



PLI Applications

- PLI application = User defined C function
  - Associated with a user defined $task or $function
  - Usually makes calls to PLI routines

```
module ...

  initial
    $hello;

  ...
```

```
#include <veriuser.h>

int sayhello(...) {

    io_printf("Hello\n");
}
```

A-6 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

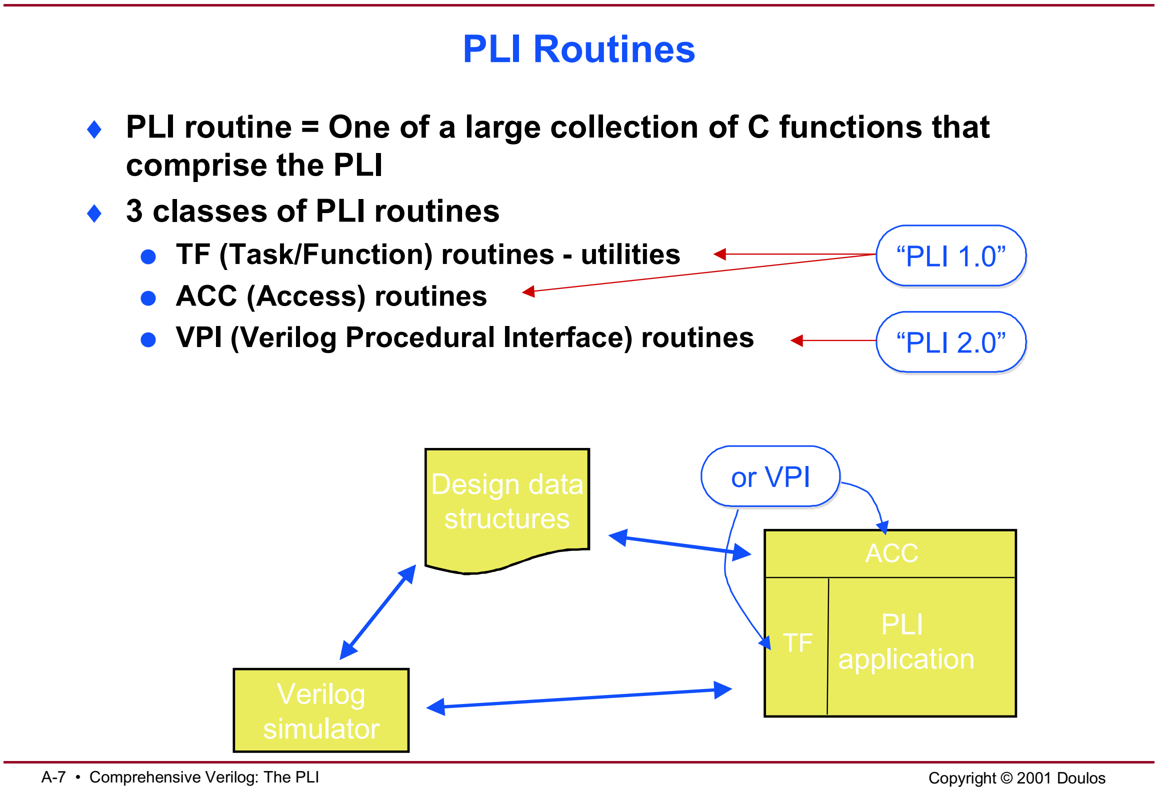
Notes:

PLI Application

The user creates PLI applications, which are C functions associated with user-defined system tasks and functions, which are called from Verilog modules. PLI applications normally call various PLI routines. (See the next page.)

The user will provide information that tells the Verilog simulator which C function to call for a specific user-defined system task or function. This information is necessary, because there is no default function name. For example, naming a user-defined task $hello doesn't necessarily imply that the corresponding C function will be called hello.

PLI Routines



PLI Routines

- PLI routine = One of a large collection of C functions that comprise the PLI
- 3 classes of PLI routines
  - TF (Task/Function) routines - utilities
  - ACC (Access) routines
  - VPI (Verilog Procedural Interface) routines

“PLI 1.0”

“PLI 2.0”

A-7 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

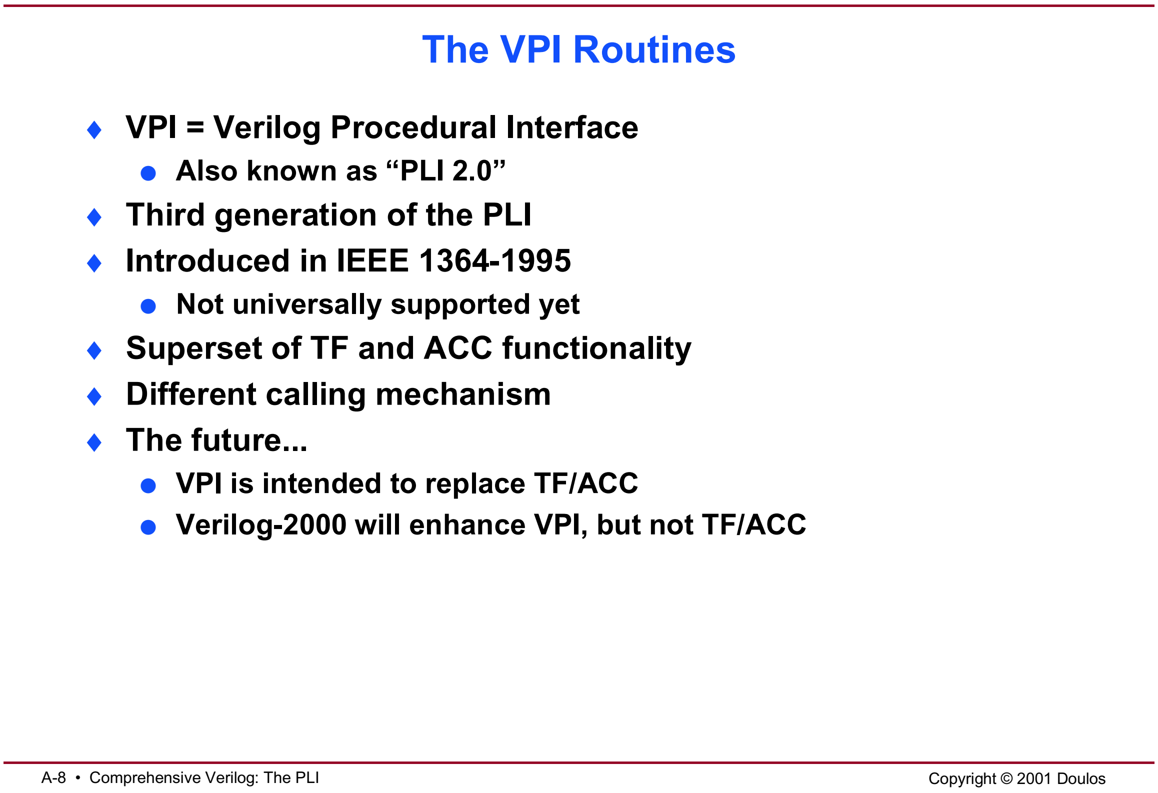
PLI routines

The PLI is comprised of a number of PLI routines. PLI routines are themselves C functions that can be called from a user's PLI application. The PLI routines are described in the IEEE 1364 standard.

PLI Routine Classes

There are three classes of PLI routine: the TF (task/function) routines, which are utilities for use in PLI applications; the ACC (access) routines, which provide access to the simulation data structures; and the VPI (Verilog Procedural Interface) routines, which provide an object-oriented interface to a Verilog simulation. The VPI routines are intended to supersede the TF and ACC routines and provide a superset of their functionality.

The VPI Routines



The VPI Routines

- VPI = Verilog Procedural Interface
  - Also known as “PLI 2.0”
- Third generation of the PLI
- Introduced in IEEE 1364-1995
  - Not universally supported yet
- Superset of TF and ACC functionality
- Different calling mechanism
- The future...
  - VPI is intended to replace TF/ACC
  - Verilog-2000 will enhance VPI, but not TF/ACC

A-8 • Comprehensive Verilog: The PLI

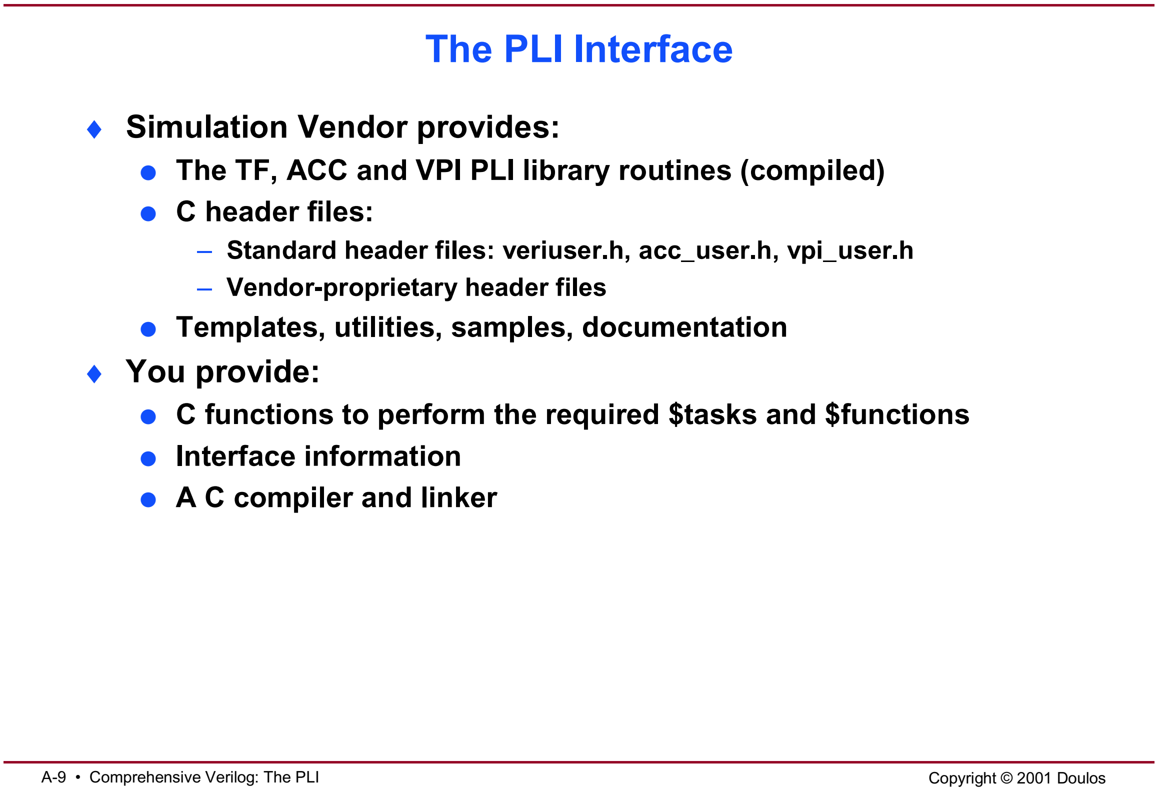
Notes:

The Verilog Procedural Interface (VPI) routines are the newest part of the PLI. Although they are part of the IEEE 1364 standard, they are not yet supported by all commercial simulators. They are sometimes called PLI 2.0, because they first appeared in version 2.0 of the proposed PLI standard, as part of the Verilog standardization process (The TF and ACC routines are sometimes known as PLI 1.0).

The intention is that the TF/ACC routines will be replaced by the VPI routines. There will not be any further enhancements to the TF/ACC routines, and any new functionality that is added as the Verilog language evolves (for example in the proposed Verilog-2000 standard) will only be accessible using the VPI routines.

The VPI routines provide a superset of the functionality of the TF and ACC routines, but have a different calling mechanism.

The PLI Interface



The PLI Interface

- Simulation Vendor provides:
  - The TF, ACC and VPI PLI library routines (compiled)
  - C header files:
    - Standard header files: veriuser.h, acc_user.h, vpi_user.h
    - Vendor-proprietary header files
  - Templates, utilities, samples, documentation
- You provide:
  - C functions to perform the required $tasks and $functions
  - Interface information
  - A C compiler and linker

A-9 • Comprehensive Verilog: The PLI

Notes:

The principles of the PLI interface are the same for all tool vendors. However, the details do vary between simulators, and are described separately for various simulators on the following pages.

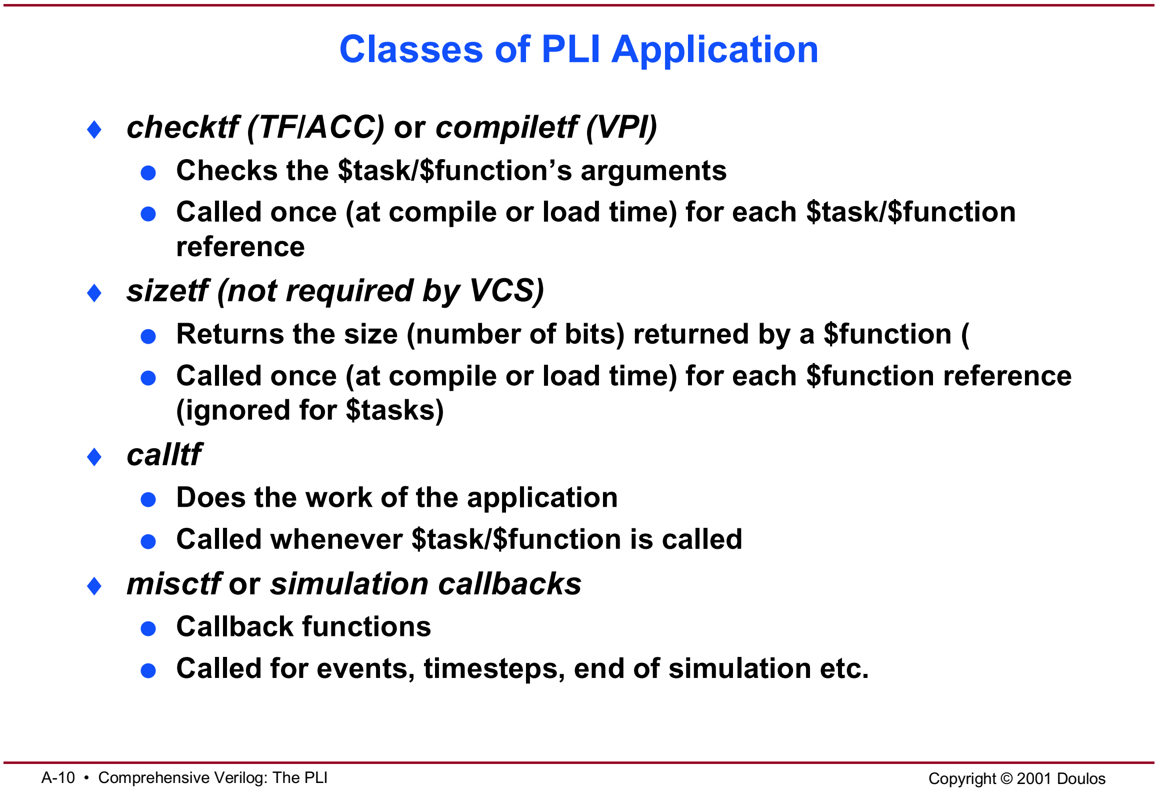
- The simulation vendor provides the PLI library (i.e. TF, ACC and VPI) routines in a compiled form, and the standard PLI C header files. Vendors may also provide additional, proprietary header files and various templates and utilities to help create PLI applications. Full documentation and sample PLI applications may also be provided.

- The user provides C functions to perform the required Verilog user-defined system tasks and functions, and information about these so that the simulator knows how and when to call them.

- The user is also responsible for compiling the C functions and possibly for linking them to create a new, customized Verilog simulator executable. Alternatively, the C functions may be compiled to form a shareable library or object file. Again, the details will depend on which simulator is being used.

Although basic information on creating PLI applications for some popular Verilog simulators is presented in this section, you should refer to the documentation that came with your simulator for full details.

Classes of PLI Application



Classes of PLI Application

- ***checktf (TF/ACC)*** **or** ***compiletf (VPI)***
  - **Checks the $task/$function's arguments**
  - **Called once (at compile or load time) for each $task/$function reference**
- ***sizetf (not required by VCS)***
  - **Returns the size (number of bits) returned by a $function (**
  - **Called once (at compile or load time) for each $function reference (ignored for $tasks)**
- ***calltf***
  - **Does the work of the application**
  - **Called whenever $task/$function is called**
- ***misctf*** **or** ***simulation callbacks***
  - **Callback functions**
  - **Called for events, timesteps, end of simulation etc.**

A-10 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

Each user-defined system task or function may be associated with a number of different C functions. Each of these plays a different role.

checktf, compiletf

A checktf (TF and ACC) or compiletf (VPI) function is used to check the number and types of the arguments used in calls to the corresponding user-defined system task or function in the Verilog code. In other words, it is used for syntax checking prior to simulation. It is called once at compile-time (or when the design is loaded for simulation) for each reference to the user-defined system task or function in the Verilog code.

sizetf

The sizetf function is only required for user-defined system functions. It is ignored for user-defined tasks. It returns the number of bits returned by the corresponding user-defined system task or function. Like a checktf or compiletf function, a sizetf function is called once at compile-time for each reference to the corresponding user-defined function in the Verilog code.

Synopsys VCS does not support sizetf functions. Instead, you supply the number of bits that the user-defined system function returns as a number in the VCS PLI table. This is described later.

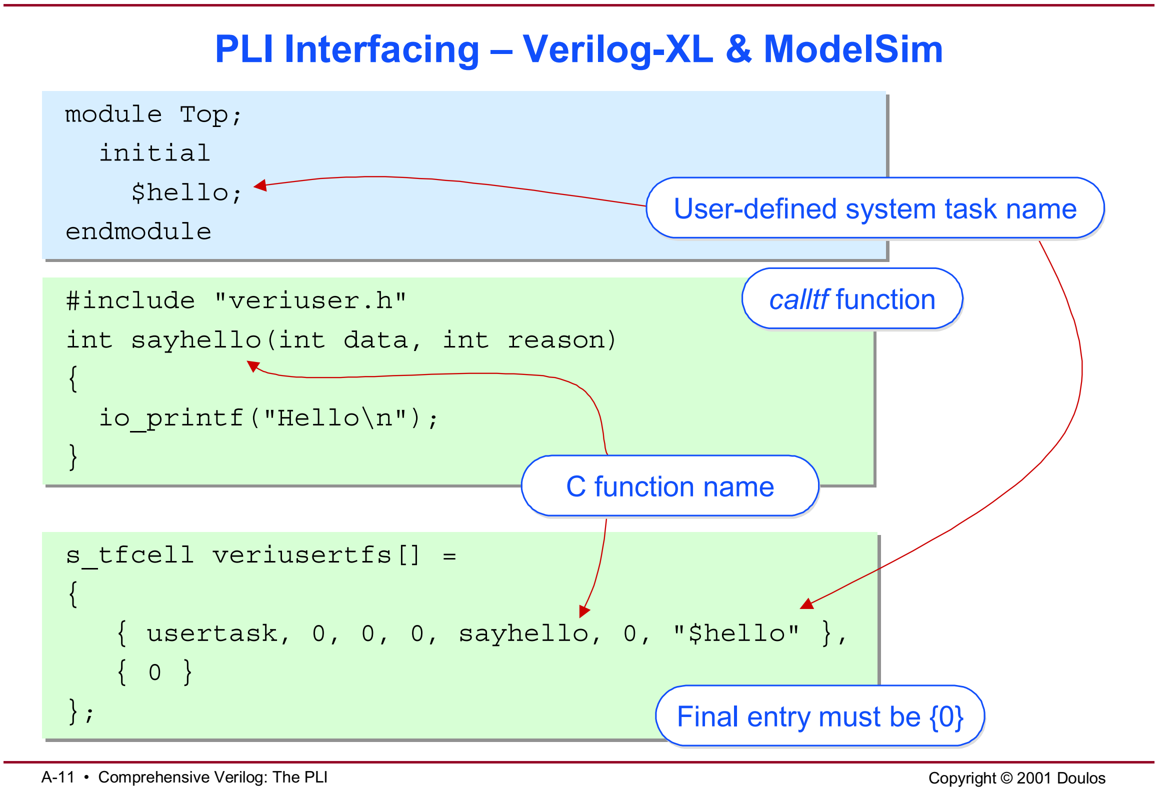
calltf

The calltf function is the C function that is called when the corresponding user-defined system task or function is called during simulation. This is the function that does the work or the user-defined system task or function. It usually calls other user-defined C functions and/or PLI library routines.

misctf, Simulation Callbacks

A misctf (TF and ACC) callback or Simulation Callback (VPI) function is called implicitly before, during or after simulation, when certain simulation events occur, such as the start or end of simulation. The simulator passes a reason flag to the function, so that the event which caused it to be called can be determined.

PLI Interfacing – Verilog-XL & ModelSim



PLI Interfacing – Verilog-XL & ModelSim

```
module Top;
  initial
    $hello;
endmodule
```

User-defined system task name

```
#include "veriuser.h"
int sayhello(int data, int reason)
{
  io_printf("Hello\n");
}
```

calltf function

C function name

```
s_tfcell veriusertfs[] =
{
    { usertask, 0, 0, 0, sayhello, 0, "$hello" },
    { 0 }
};
```

Final entry must be {0}

A-11 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

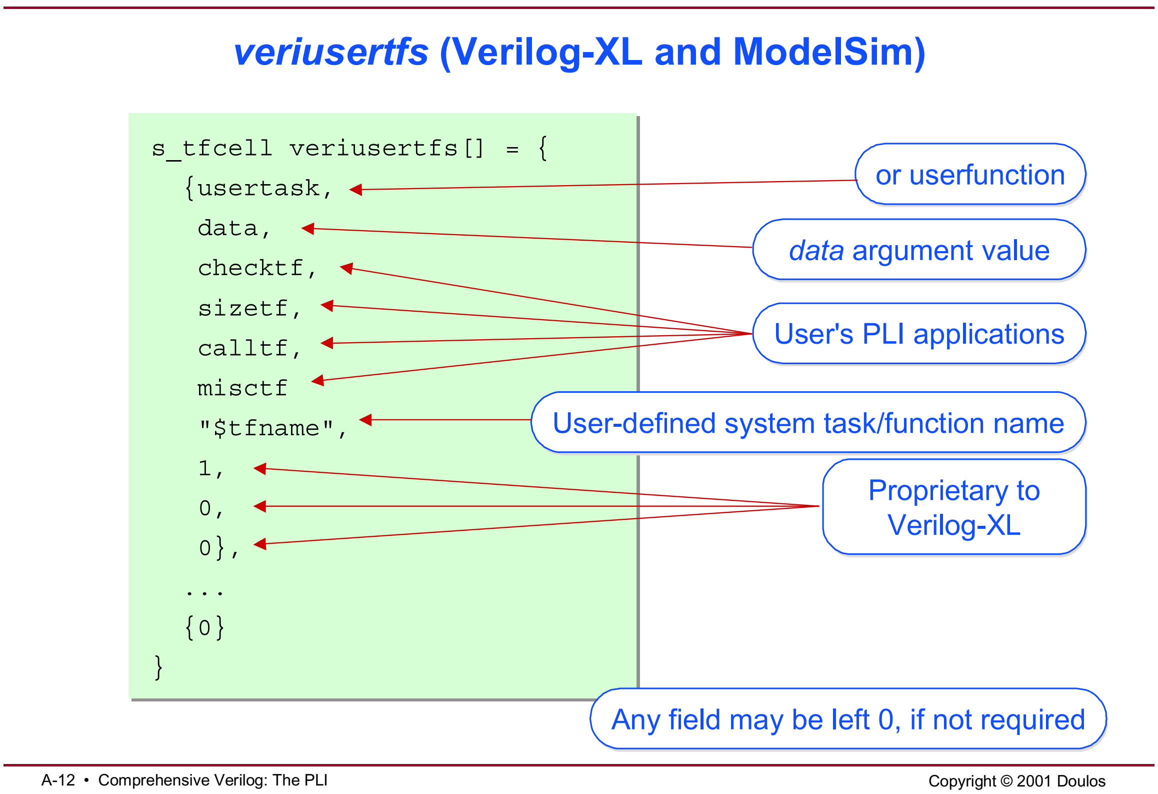
Notes:

For Cadence Verilog-XL and Model Technology ModelSim, the user creates an entry in a table veriusertfs. (Cadence provides a veriuser.c file for you to copy and edit: the veriusertfs table goes in this file.) An entry must be created in the veriusertfs table for each user-defined system task or function that can be called from your Verilog code. The entry associates the user-defined system task or function name with the user's PLI application(s). The final entry in veriusertfs must be {0}.

The definition of veriusertfs is not part of the IEEE 1364-1995 standard.

In the example shown, the user-defined system task $hello is associated in the veriusertfs table with the calltf application sayhello. This in turn calls the TF library routine, io_printf.

veriusertfs (Verilog-XL and ModelSim)



veriusertfs (Verilog-XL and ModelSim)

```
s_tfcell veriusertfs[] = {
  {usertask,
   data,
   checktf,
   sizetf,
   calltf,
   misctf
   "$tfname",
   1,
   0,
   0},
  ...
  {0}
}
```

or userfunction

data argument value

User's PLI applications

User-defined system task/function name

Proprietary to Verilog-XL

Any field may be left 0, if not required

A-12 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

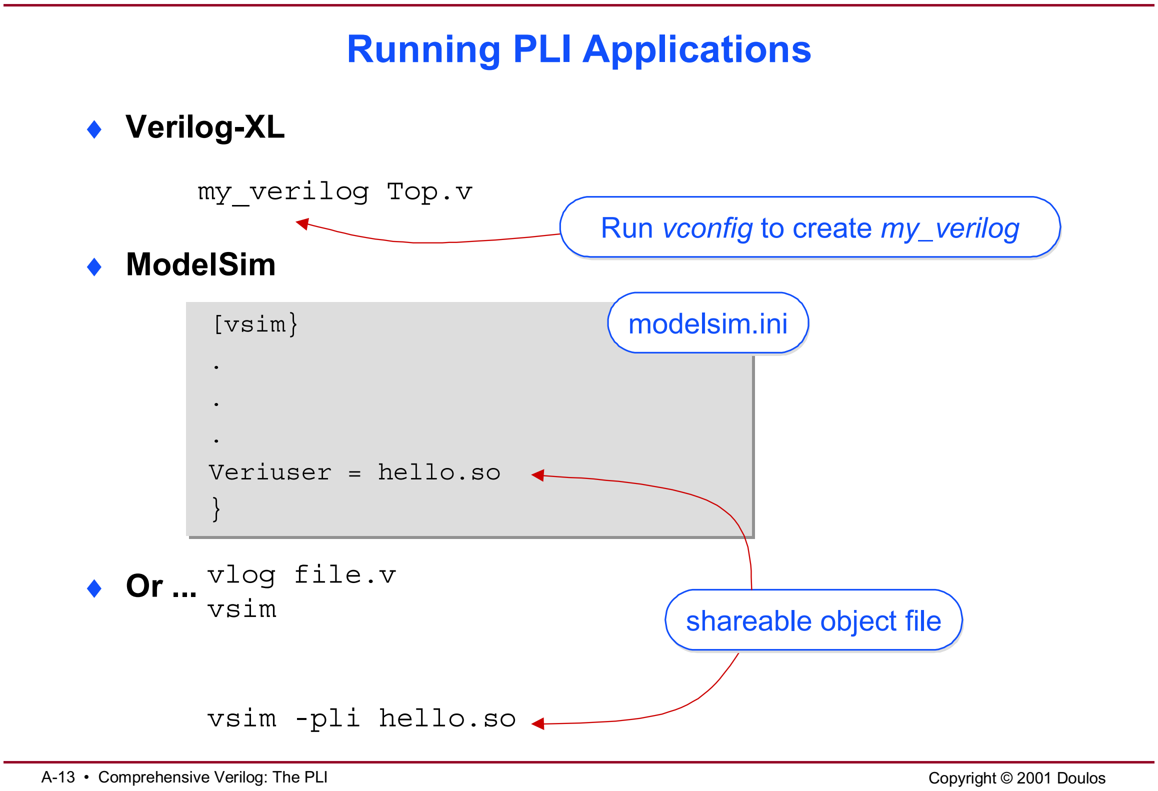
The slide above shows the format of the veriusertfs table, which is used by Verilog-XL and ModelSim.

If a field is not required, use the value 0. The final record in the table must be {0}.

The data field can be used if several user-defined system tasks or functions call the same C function: the function's data argument can then indicate which $task or $function was called.

The three final entries in each field should usually be set to 1, 0 and 0 respectively. These fields are proprietary to Cadence. For details, see Cadence's documentation.

Running PLI Applications



Running PLI Applications

- Verilog-XL

```
my_verilog Top.v
```

Run *vconfig* to create *my_verilog*

- ModelSim

```
[vsim}
.
.
.
Veriuser = hello.so
}
```

modelsim.ini

- Or ...

```
vlog file.v
vsim
```

shareable object file

```
vsim -pli hello.so
```

A-13 • Comprehensive Verilog: The PLI

Notes:

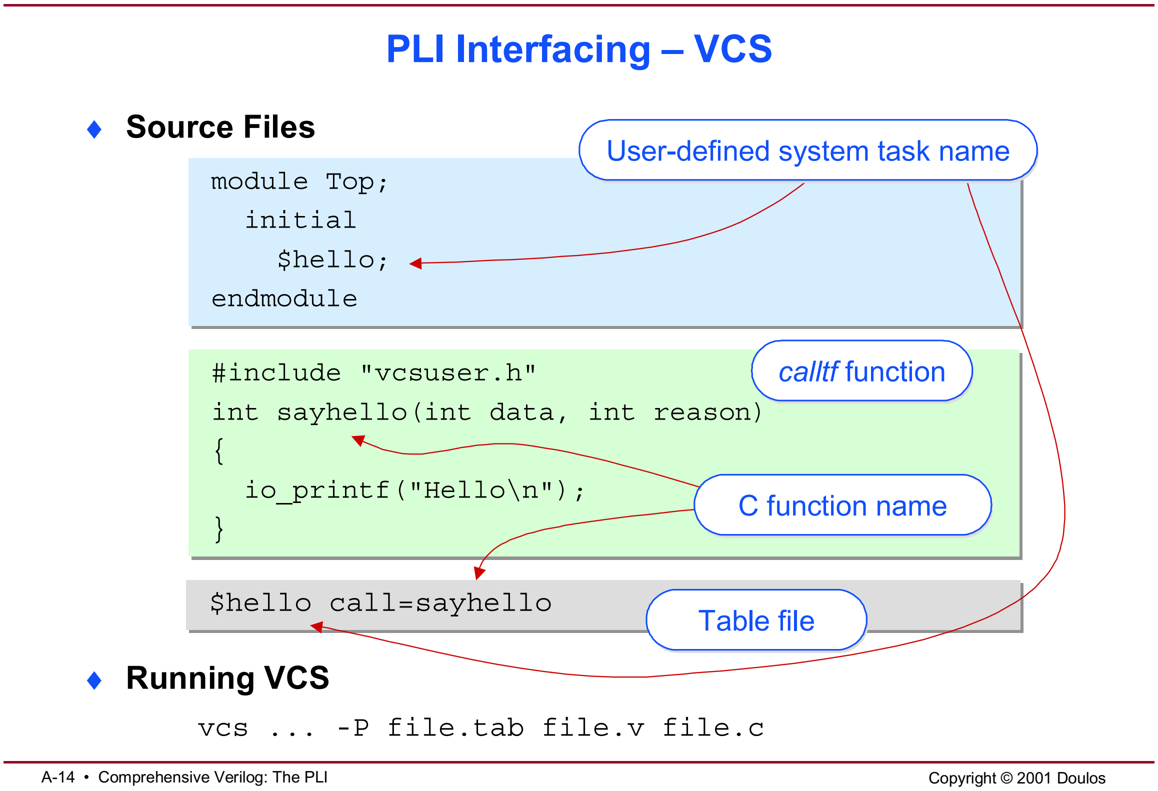
Verilog-XL

Cadence provides a utility, vconfig, which is used to create a script cr_vlog, which in turn is used to compile the user's C functions and customized veriuser.c and link them with object files supplied by Cadence to form a custom Verilog-XL simulator. This custom simulator is then used to perform simulations. In the example, the customized simulator is called my_verilog. This program is run with the exactly the same command-line options and switches as the standard Verilog-XL program.

ModelSim

With ModelSim, you must compile your C code and create a shareable object file (Solaris and Linux), shared object library (HP-UX) or Dynamic Link Library (Microsoft Windows). You must tell ModelSim about this file when you run the simulator. You can do this using the command-line switch -pli, or by editing the modelsim.ini file and including the Veriuser field. You compile your Verilog code and run the simulator in the normal way.

PLI Interfacing – VCS



PLI Interfacing – VCS

- **Source Files**

```
module Top;
  initial
    $hello;
endmodule
```

```
#include "vcsuser.h"
int sayhello(int data, int reason)
{
  io_printf("Hello\n");
}
```

```
$hello call=sayhello
```

- **Running VCS**

```
vcs ... -P file.tab file.v file.c
```

A-14 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

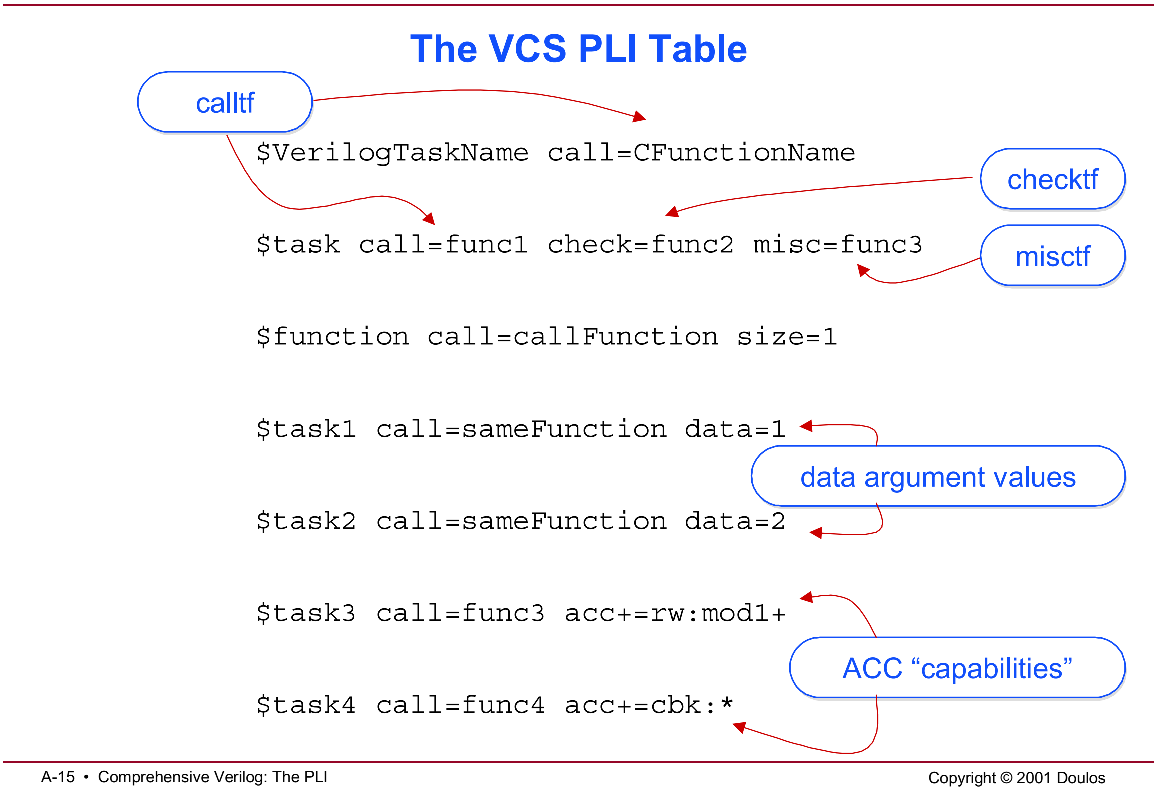
Notes:

The mechanism for linking PLI routines with Synopsys VCS is slightly different from that for Verilog-XL and ModelSim. (If you have existing PLI applications that were written for use with these tools, there is a utility supplied with VCS to convert existing veriusertfs tables to VCS tables).

For VCS, the user creates an entry in a PLI table. Unlike the veriusertfs table, this is not C syntax, but a separate text file. There is one table entry for each user-defined system task or function that can be called from Verilog code. The entry associates the task or function name with the user's PLI application(s).

The table is then referenced on the VCS command line using the option -P. Also included on the command line is the name of the file(s) containing the user's C function(s). The C functions are compiled along with the Verilog code.

The VCS PLI Table



The VCS PLI Table

calltf

checktf

misctf

data argument values

ACC “capabilities”

```
$VerilogTaskName call=CFunctionName

$task call=func1 check=func2 misc=func3

$function call=callFunction size=1

$task1 call=sameFunction data=1

$task2 call=sameFunction data=2

$task3 call=func3 acc+=rw:mod1+

$task4 call=func4 acc+=cbk:*
```

A-15 • Comprehensive Verilog: The PLI

Notes:

The slide above shows examples of VCS PLI table entries.

Each entry starts with the name of the user-defined system task or function. There follows one or more PLI or ACC specifications.

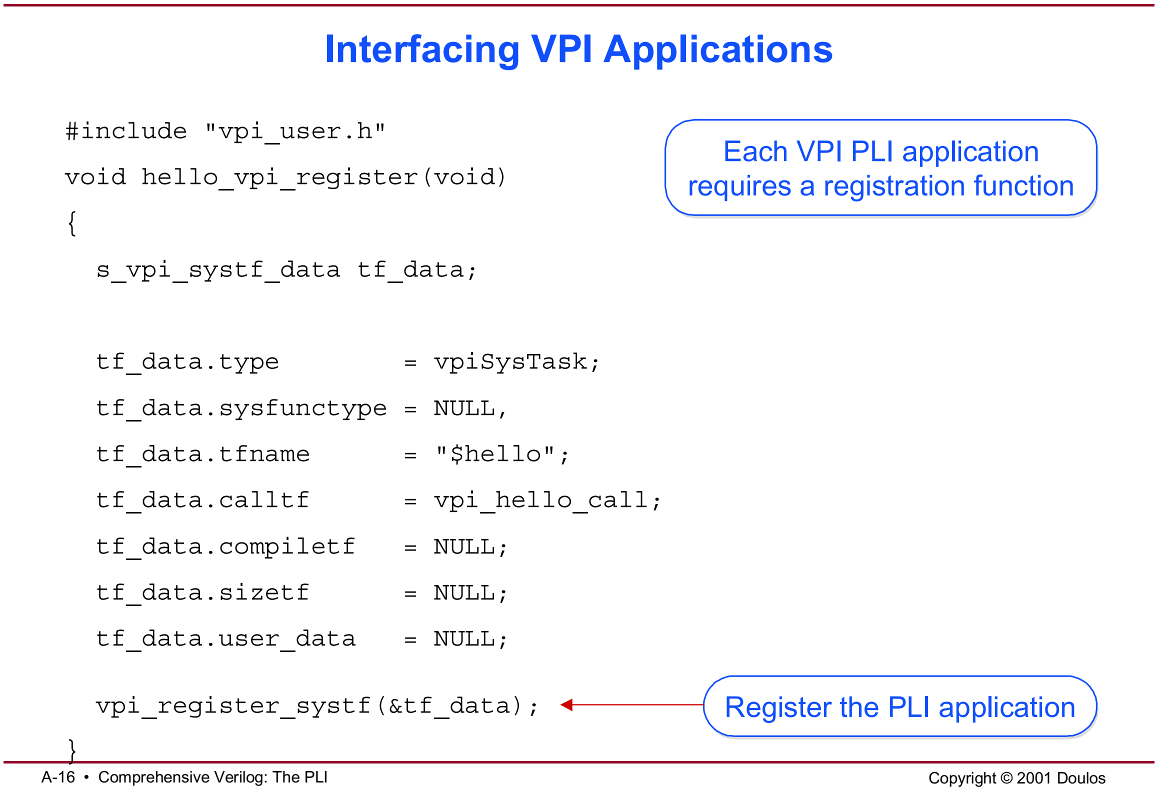
call=function_name identifies the calltf function corresponding to the user-defined system task or function. Similarly, check=function_name, size=function_name and misc=function_name identify the checktf, sizetf and misctf functions respectively.

data=value provides a value for the data argument for the PLI application. This could be used if the same C function is called for two different user-defined system tasks, as described earlier.

The ACC specifications enable or disable ACC access to specified objects in the design. For example, acc+=rw:mod1+ enables ACC read/write access to registers and nets in the module mod1 and its children. acc+=cbk:* enables callbacks on all objects in the design. These and other ACC specifications are described fully in the VCS documentation.

ACC capabilities can also be specified on the VCS command-line. This is described in the VCS documentation.

Interfacing VPI Applications



Interfacing VPI Applications

```
#include "vpi_user.h"
void hello_vpi_register(void)
{
  s_vpi_systf_data tf_data;

  tf_data.type        = vpiSysTask;
  tf_data.sysfunctype = NULL,
  tf_data.tfname      = "$hello";
  tf_data.calltf      = vpi_hello_call;
  tf_data.compiletf   = NULL;
  tf_data.sizetf      = NULL;
  tf_data.user_data   = NULL;

  vpi_register_systf(&tf_data);
}
```

Each VPI PLI application requires a registration function

Register the PLI application

A-16 • Comprehensive Verilog: The PLI

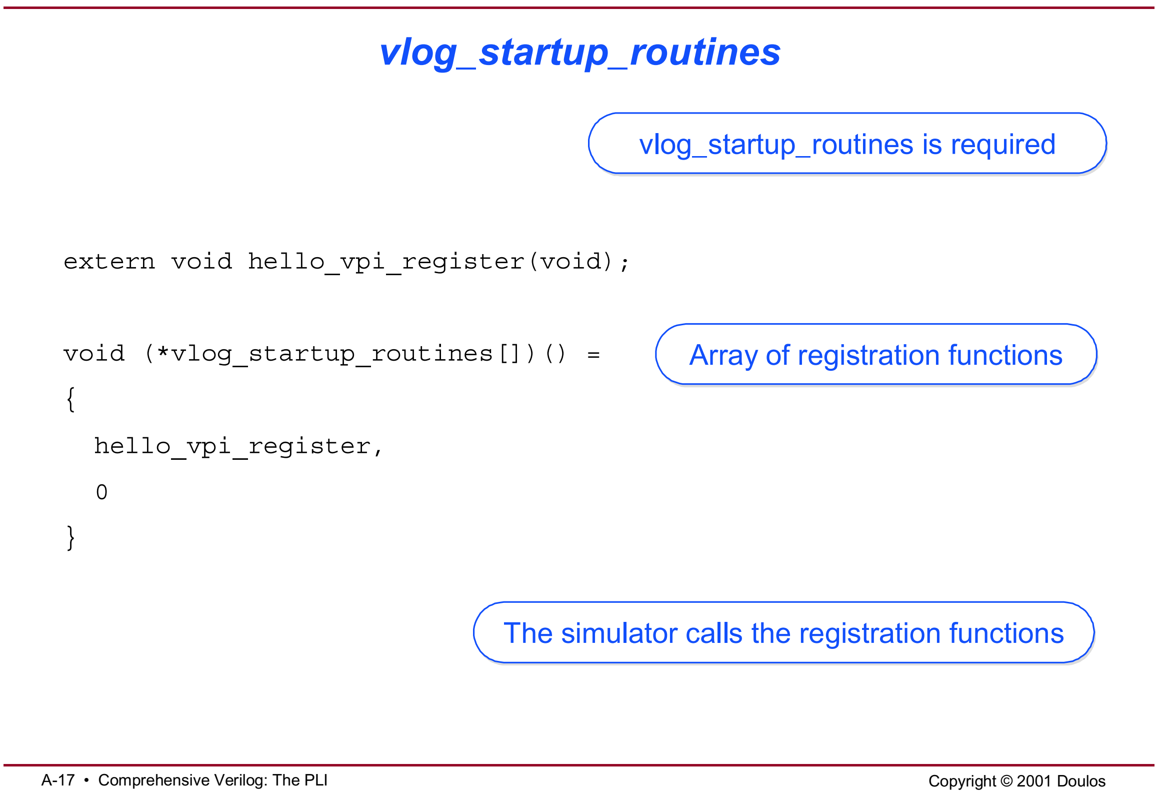
Notes:

To interface a VPI PLI application to a Verilog simulator, you need to create a registration function for the application and include the registration function in the vlog_startup_routines table.

The registration function should create a s_vpi_systf_data struct and initialize it. This is analogous to creating an entry in the veriusertfs table.

The registration function registers the VPI application by calling vpi_register_systf and passing a pointer to the s_vpi_systf_data struct.

vlog_startup_routines



vlog_startup_routines

vlog_startup_routines is required

```
extern void hello_vpi_register(void);

void (*vlog_startup_routines[])() =
{
  hello_vpi_register,
  0
}
```

Array of registration functions

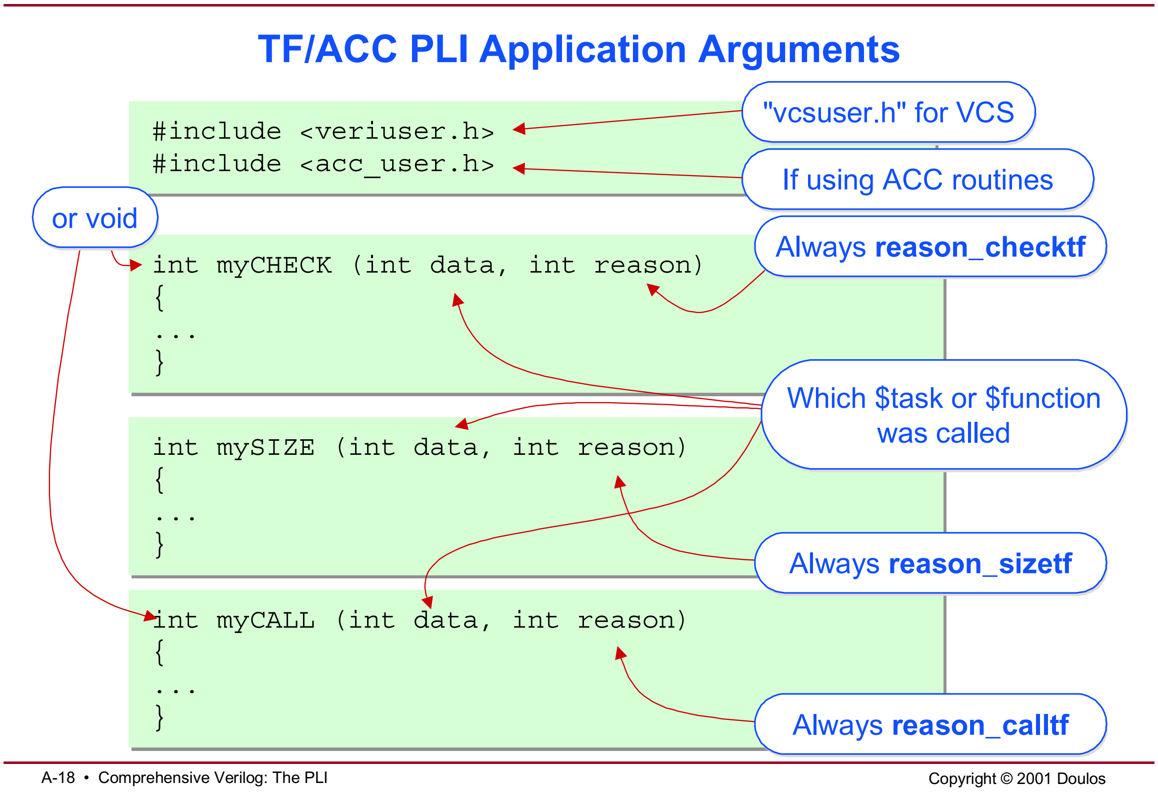
The simulator calls the registration functions

A-17 • Comprehensive Verilog: The PLI

Notes:

vlog_startup_routines is a zero-terminated array of functions. This is required by the IEEE standard. The simulator uses the array to call all the registration functions in it. It is therefore important that all registration functions are included in this array.

TF/ACC PLI Application Arguments



TF/ACC PLI Application Arguments

```
#include <veriuser.h>
#include <acc_user.h>
```

```
int myCHECK (int data, int reason)
{
...
}
```

```
int mySIZE (int data, int reason)
{
...
}
```

```
int myCALL (int data, int reason)
{
...
}
```

"vcsuser.h" for VCS

If using ACC routines

or void

Always **reason_checktf**

Which $task or $function was called

Always **reason_sizetf**

Always **reason_calltf**

A-18 • Comprehensive Verilog: The PLI

Notes:

This and the following page describe the arguments used by TF and ACC PLI applications.

C header files

PLI applications require certain C header files (these are provided by the simulation tool vendor.) For Verilog-XL and Modelsim, include veriuser.h. For VCS, include vcsuser.h instead. (vcsuser.h is the same as veriuser.h) If you are using the ACC routines with any vendor, include acc_user.h. Vendors may also have proprietary header files, for example to define proprietary extensions to the PLI.

PLI applications return an int and have two arguments, data and reason. (misctf functions also have a third argument, paramvc, which is described on the next page). The return value is only used by sizetf functions, so it is common practice to declare other application routines as returning void.

data

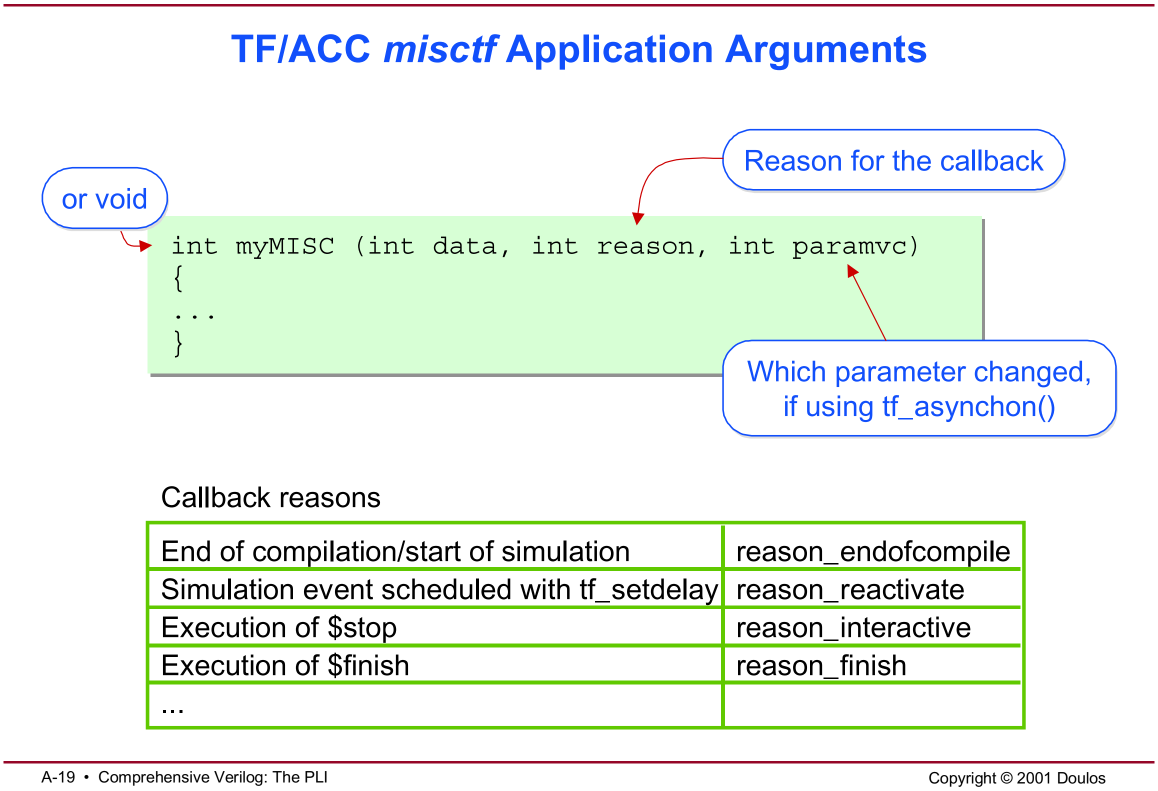
The data argument is used to determine which user-defined system task or function has caused the application to be called. This can be used when the same PLI application is called by a number of different user-defined system tasks and functions.

reason

The reason argument indicates why the PLI application was called. The most common use for the reason argument is so that a misctf PLI application can determine which simulation event has initiated the callback.

Note that for may PLI applications the arguments' values are not used. In particular, the reason argument is always reason_chectf for a checktf application; reason_sizetf for a sizetf application; and reason_calltf for a calltf application.

TF/ACC *misctf* Application Arguments



TF/ACC *misctf* Application Arguments

```
int myMISC (int data, int reason, int paramvc)
{
...
}
```

Callback reasons

End of compilation/start of simulation	reason_endofcompile
Simulation event scheduled with tf_setdelay	reason_reactivate
Execution of $stop	reason_interactive
Execution of $finish	reason_finish
...	

A-19 • Comprehensive Verilog: The PLI

Notes:

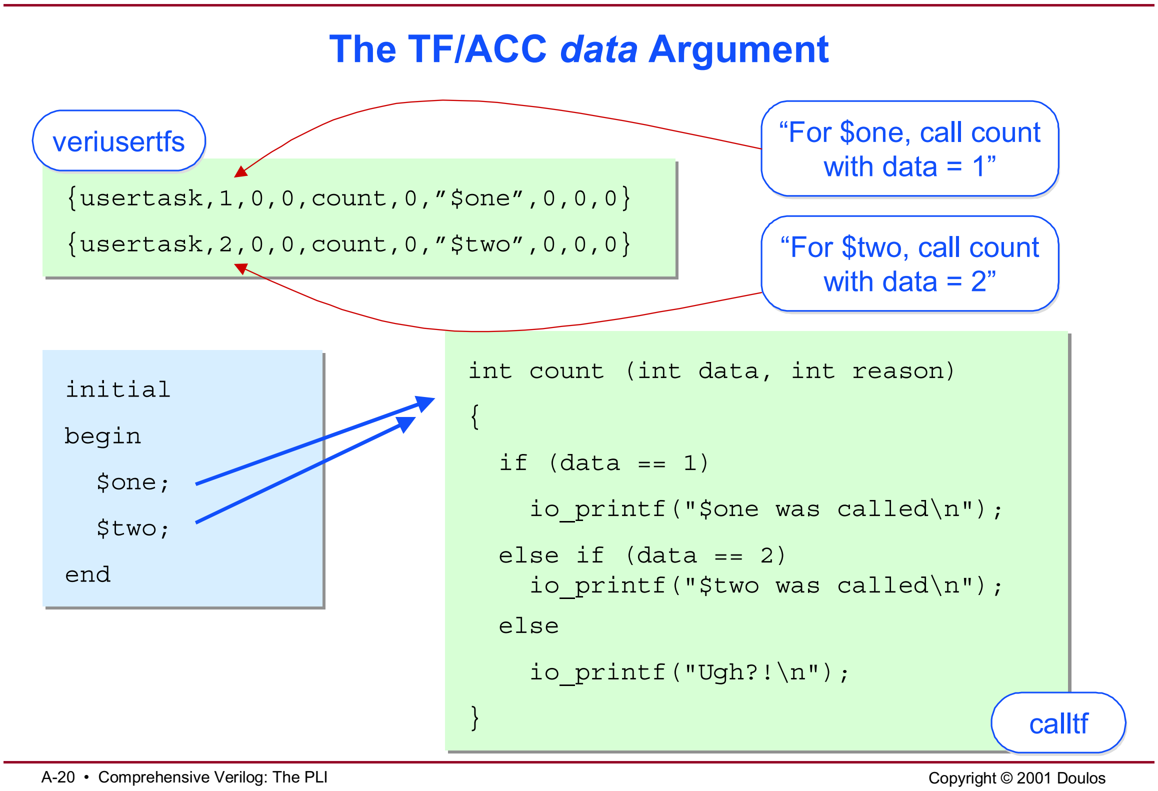
misctf applications have the data and reason arguments like other applications. They also have a third argument, paramvc.

The reason argument is important for a misctf application is because there are many different simulation events that could cause the callback and the intended reason must be detected. Note that every misctf application is called for every possible callback event - there is no way to filter out unwanted events, except within the misctf routine itself.

tf_asynchon

The PLI routine tf_asynchon is used to force callbacks when the values of specified wires and registers change during simulation. In this case a misctf application would be called with a reason reason_paramvc or reason_paramdrc, and the paramvc argument would indicate which object's value has changed. There is an example of this mechanism on a later slide.

The TF/ACC *data* Argument



The TF/ACC *data* Argument

veriusertfs

```
{usertask,1,0,0,count,0,"$one",0,0,0}
{usertask,2,0,0,count,0,"$two",0,0,0}
```

"For $one, call count with data = 1"

"For $two, call count with data = 2"

```
initial
begin
  $one;
  $two;
end
```

```
int count (int data, int reason)
{
  if (data == 1)
    io_printf("$one was called\n");
  else if (data == 2)
    io_printf("$two was called\n");
  else
    io_printf("Ugh?!\n");
}
```

calltf

A-20 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

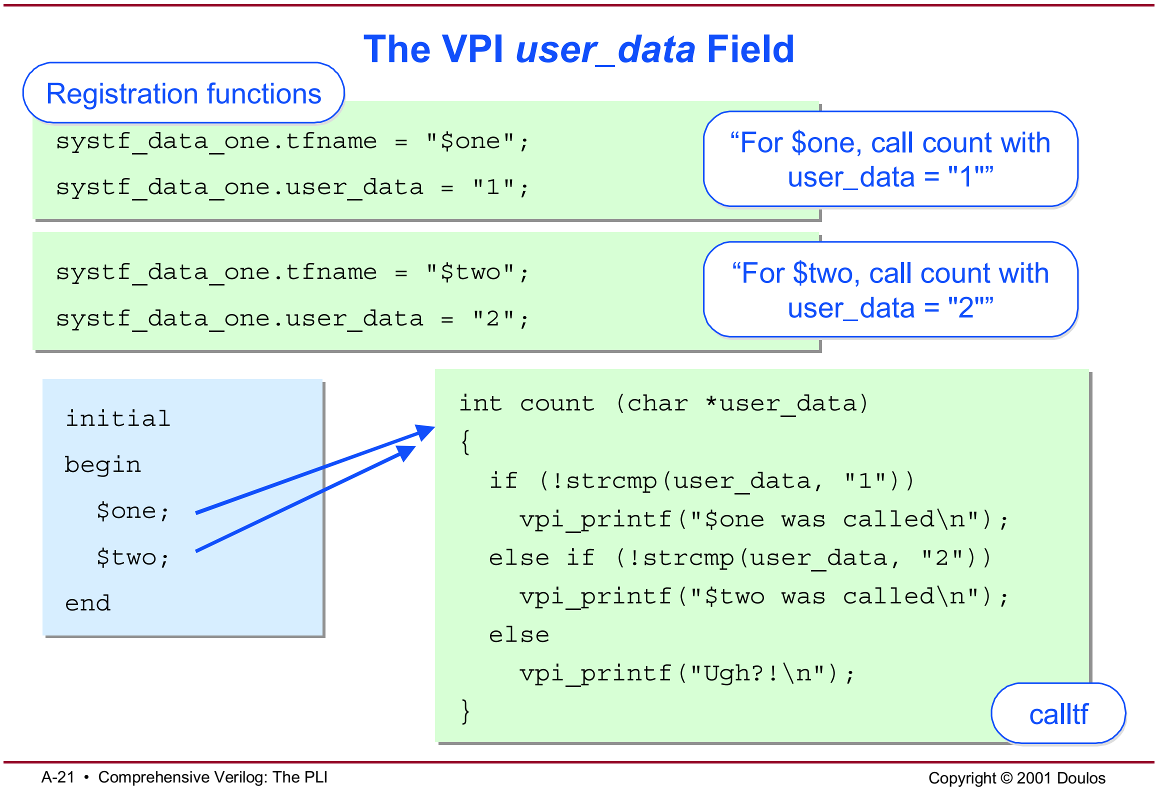
Notes:

In the Verilog code opposite there are calls to two user-defined system tasks, $one and $two. The same C function, count, is to be executed whenever either of these system tasks is called. Count is associated with $one and $two in the veriusertfs table (for Verilog-XL and ModelSim) or the PLI table (VCS). The table tells the simulator that the C function count should be called with a data argument value of 1 for $one and 2 for $two.

The VCS PLI table is not shown in the diagram. It would have entries like these:

```
$one call=count data=1
$two call=count data=2
```

The VPI *user_data* Field



The VPI *user_data* Field

Registration functions

```
systf_data_one.tfname = "$one";
systf_data_one.user_data = "1";
```

“For $one, call count with user_data = "1"”

```
systf_data_one.tfname = "$two";
systf_data_one.user_data = "2";
```

“For $two, call count with user_data = "2"”

```
initial
begin
  $one;
  $two;
end
```

```
int count (char *user_data)
{
  if (!strcmp(user_data, "1"))
    vpi_printf("$one was called\n");
  else if (!strcmp(user_data, "2"))
    vpi_printf("$two was called\n");
  else
    vpi_printf("Ugh?!\n");
}
```

calltf

A-21 • Comprehensive Verilog: The PLI

Notes:

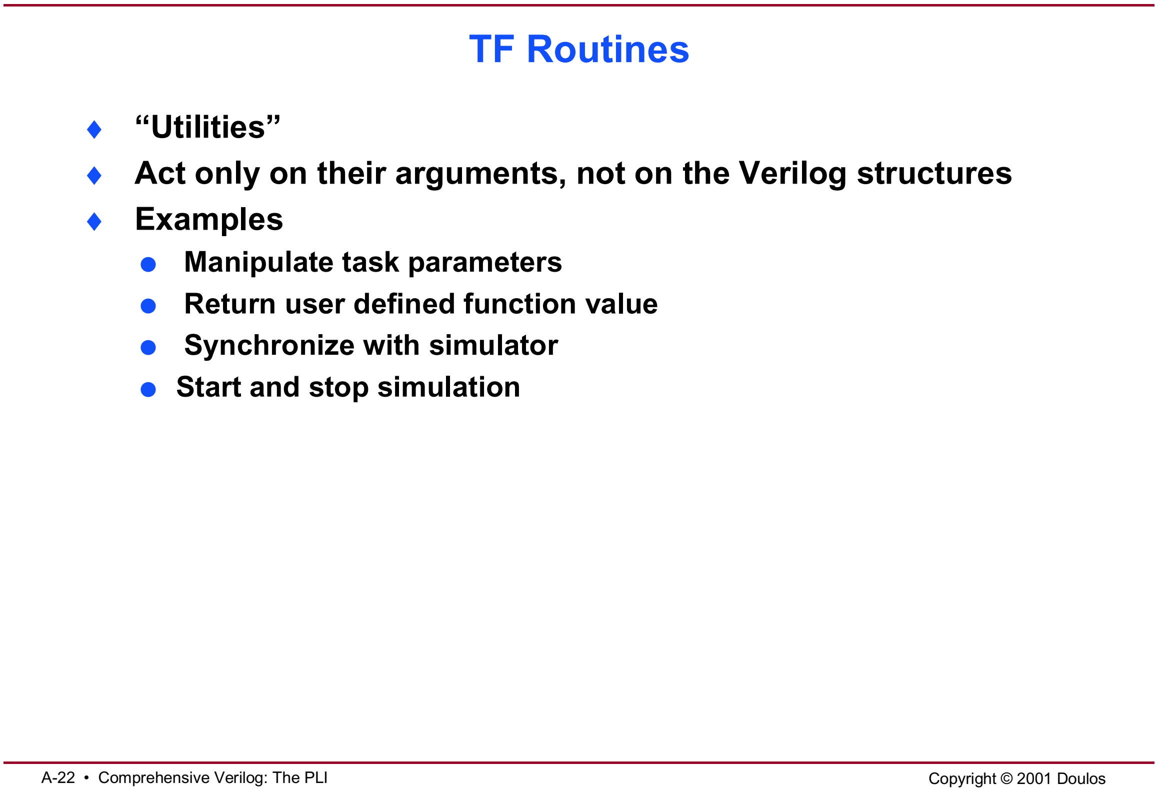
The VPI user_data field works in a similar way to the TF/ACC data argument. The main difference is that user_data is a pointer to a char. This means that, unlike the TF/ACC data field, which is limited to being a 32-bit integer, the VPI user_data field can be a string of arbitrary length.

In the example above, the same C function, count, is called whenever either of the user-defined systems $one or $two is called. If $one is called, the user_data argument will be the single-character string "1"; if $two is called user_data will be "2".

Note that because user_data is a string, we could have used the actual names of the user-defined system tasks, i.e. $one and $two respectively.

```
systf_data_one.user_data = "1";
systf_data_two.user_data = "2";
int count (char *user_data)
{
  if (!strcmp(user_data, "$one"))
    vpi_printf("$one was called\n");
  else if (!strcmp(user_data, "$two"))
    vpi_printf("$two was called\n");
  else
    vpi_printf("Ugh?!\n");
}
```

TF Routines



TF Routines

- “Utilities”
- Act only on their arguments, not on the Verilog structures
- Examples
  - Manipulate task parameters
  - Return user defined function value
  - Synchronize with simulator
  - Start and stop simulation

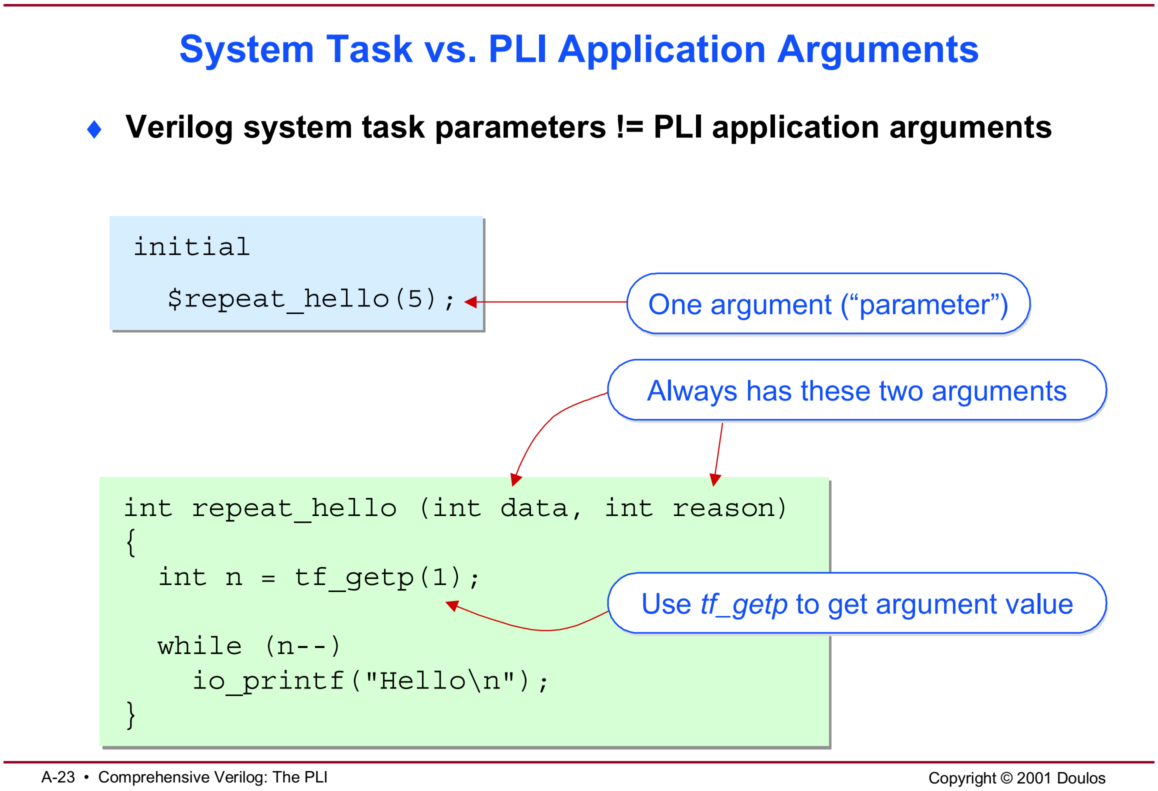
A-22 • Comprehensive Verilog: The PLI

Notes:

The TF (utility) PLI routines are used to manipulate the parameters of user-defined tasks and functions, and to synchronize interaction between a task and the simulator. They do not access directly the data structures containing information about the design.

The next few slides present examples of TF routines.

System Task vs. PLI Application Arguments



System Task vs. PLI Application Arguments

- Verilog system task parameters != PLI application arguments

```
initial
  $repeat_hello(5);
```

One argument (“parameter”)

Always has these two arguments

```
int repeat_hello (int data, int reason)
{
  int n = tf_getp(1);

  while (n--)
    io_printf("Hello\n");
}
```

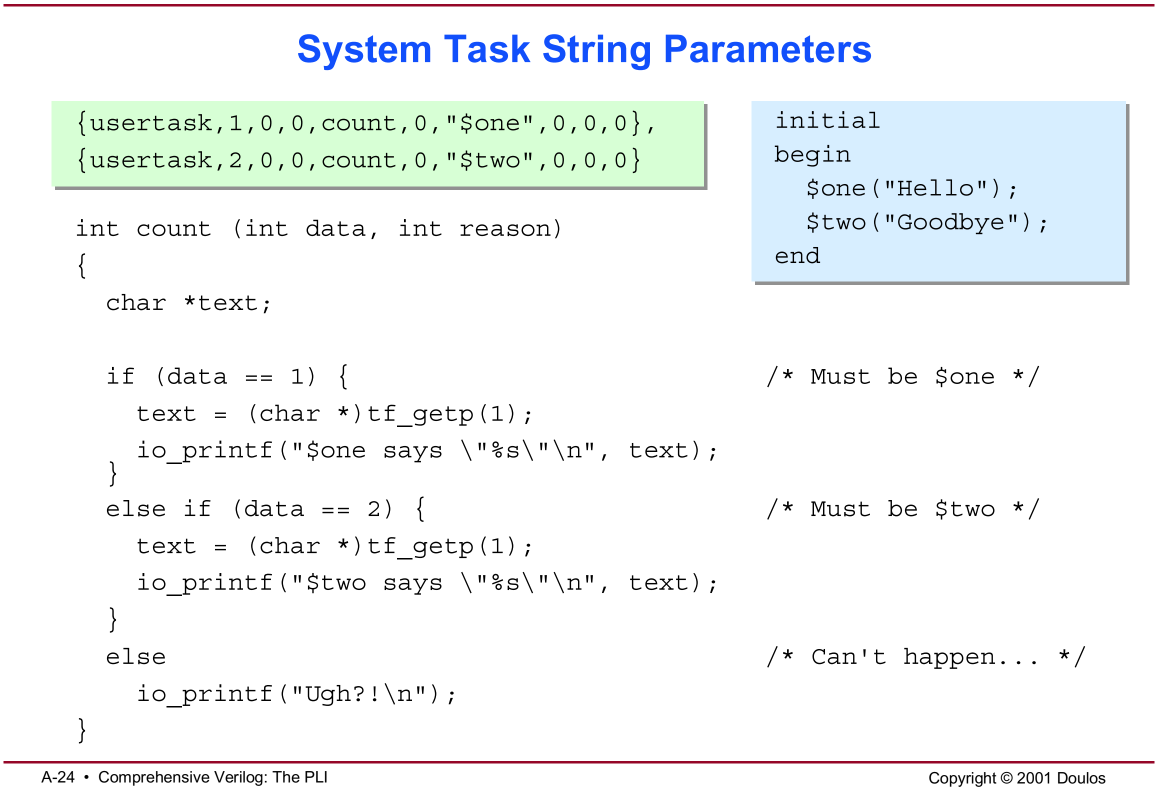
Use *tf_getp* to get argument value

A-23 • Comprehensive Verilog: The PLI

Notes:

Note that PLI applications (i.e. the C functions) always have two or three arguments (as described earlier), irrespective of the number of parameters of the corresponding user-defined system task. These parameters are passed to the C application using appropriate TF routines. For example the function tf_getp obtains the value of a parameter, as shown opposite. A further example is presented on the next slide.

System Task String Parameters



System Task String Parameters

```
{usertask,1,0,0,count,0,"$one",0,0,0},
{usertask,2,0,0,count,0,"$two",0,0,0}
```

```
initial
begin
  $one("Hello");
  $two("Goodbye");
end
```

```
int count (int data, int reason)
{
  char *text;

  if (data == 1) {                           /* Must be $one */
    text = (char *)tf_getp(1);
    io_printf("$one says \"%s\"\n", text);
  }
  else if (data == 2) {                      /* Must be $two */
    text = (char *)tf_getp(1);
    io_printf("$two says \"%s\"\n", text);
  }
  else                                       /* Can't happen... */
    io_printf("Ugh?!\n");
}
```

A-24 • Comprehensive Verilog: The PLI

Notes:

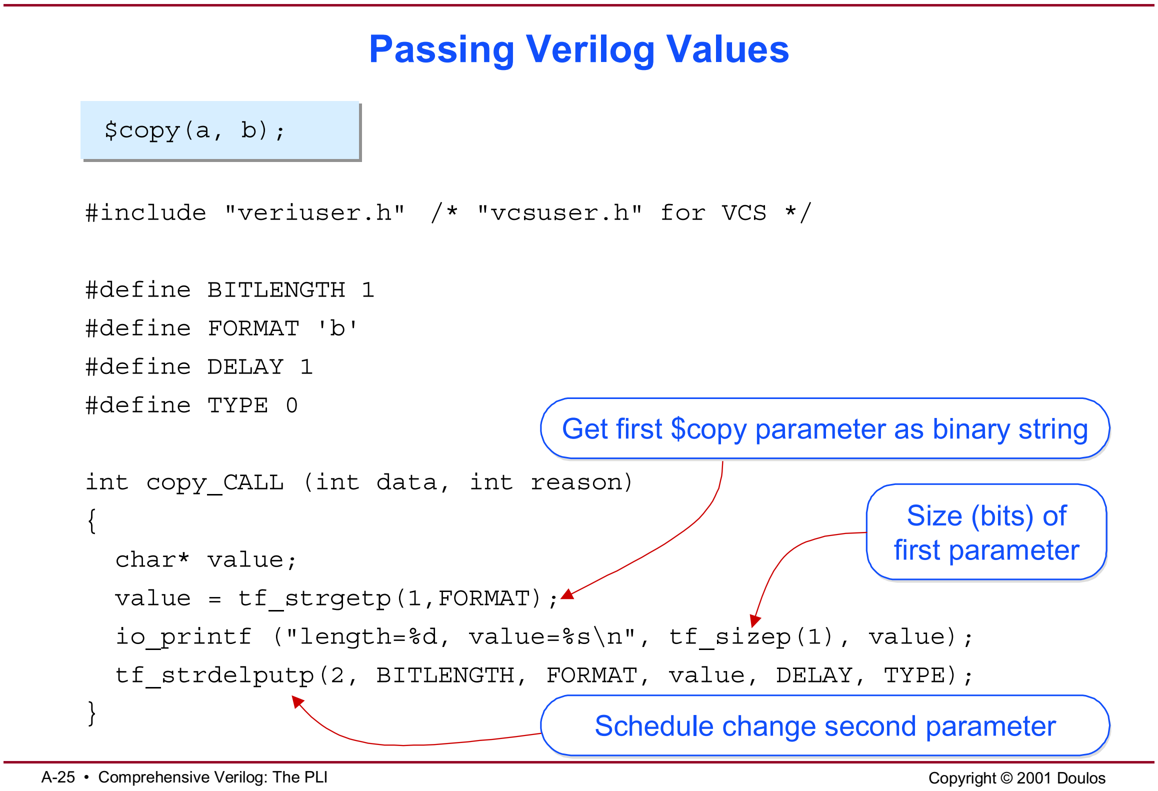
This example shows how strings can be passed from a call to a user-defined system task to a PLI application. The example also shows how the data argument can be used to identify which user-defined system task caused the PLI application to be called.

The TF routine tf_getp returns the value of a parameter to the user-defined system task that caused the C function to be called. In this example tf_getp returns a pointer to a string, which will be the value of the first parameter of $one or $two.

Parameter

Don't confuse the term "parameter" when used of the parameters of a task with Verilog parameters, which are runtime constants.

Passing Verilog Values



Passing Verilog Values

```
$copy(a, b);
```

```
#include "veriuser.h"  /* "vcsuser.h" for VCS */

#define BITLENGTH 1
#define FORMAT 'b'
#define DELAY 1
#define TYPE 0

int copy_CALL (int data, int reason)
{
  char* value;
  value = tf_strgetp(1,FORMAT);
  io_printf ("length=%d, value=%s\n", tf_sizep(1), value);
  tf_strdelputp(2, BITLENGTH, FORMAT, value, DELAY, TYPE);
}
```

Get first $copy parameter as binary string

Size (bits) of first parameter

Schedule change second parameter

A-25 • Comprehensive Verilog: The PLI

Notes:

This example shows how the TF routines are used to retrieve and modify the user-defined system task or function's parameters.

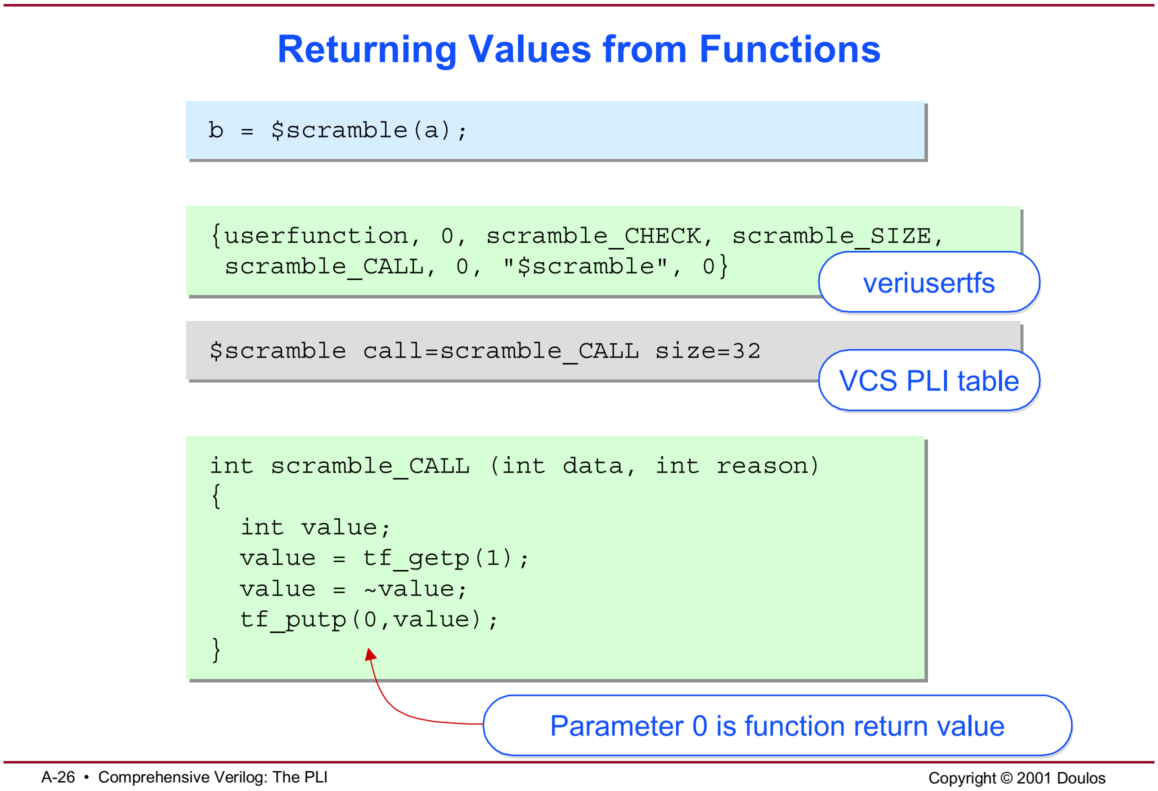
The user-defined system task, $copy, prints the value of the first parameter and copies it to the second parameter, after a delay.

The C function copy_CALL is called when $copy is called. tf_strgetp returns a pointer to a string containing the value of a, the first parameter of $copy. The string contains the value of a in binary format (c.f. the format specifier %b for $display). The format is specified by the value of the second argument to tf_strgetp, FORMAT, which has the value 'b'.

io_printf prints the number of bits in the first parameter, and the value as a string. The syntax of io_printf is identical to that of the standard C function printf, but it causes the text to be added to the Verilog log file as well as the standard output.

Finally, tf_strdelputp copies the value of a to the second parameter of $copy, b, after a delay of DELAY (1). The TYPE (0) indicates that inertial delay is to be used.

Returning Values from Functions



Returning Values from Functions

```
b = $scramble(a);
```

```
{userfunction, 0, scramble_CHECK, scramble_SIZE,
 scramble_CALL, 0, "$scramble", 0}
```

veriusertfs

```
$scramble call=scramble_CALL size=32
```

VCS PLI table

```
int scramble_CALL (int data, int reason)
{
  int value;
  value = tf_getp(1);
  value = ~value;
  tf_putp(0,value);
}
```

Parameter 0 is function return value

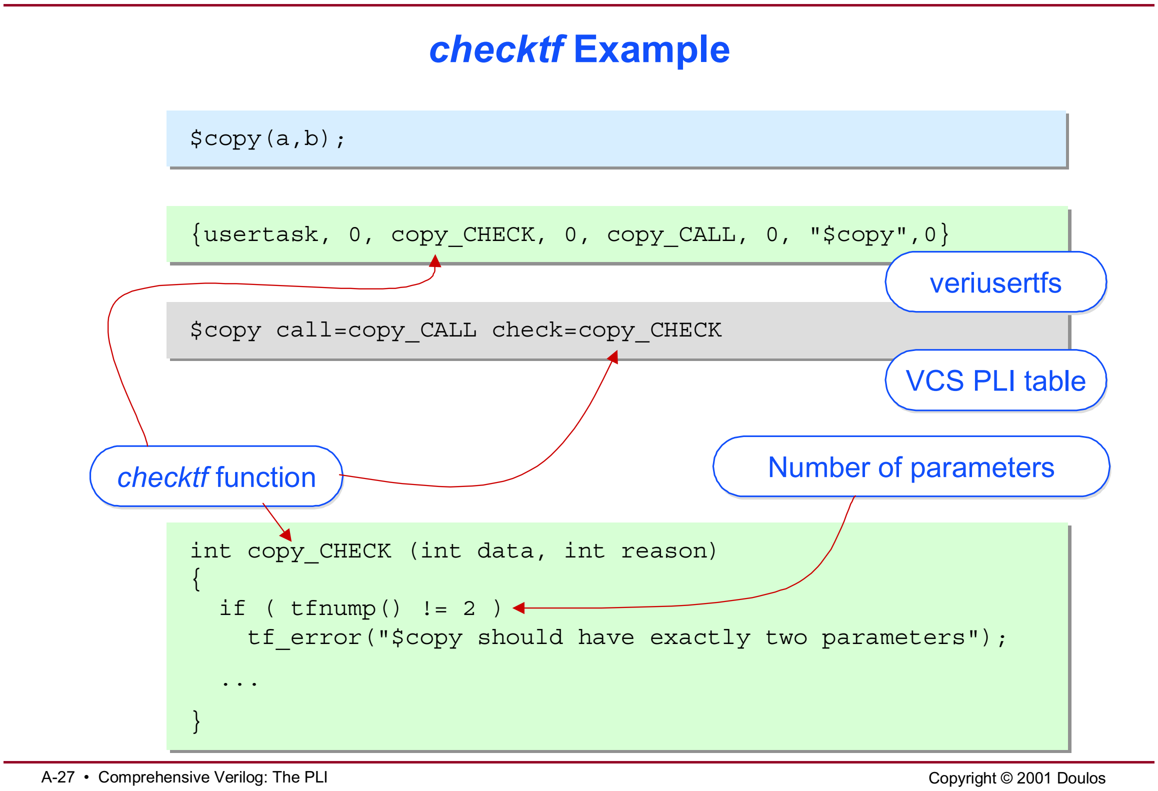
Notes:

This example shows the PLI routine tf_putp being used to return a value from a user-defined system function. The function $scramble returns the value of its parameter having undergone bitwise inversion.

Note that parameter 0 is the function's return value. Parameter 1 is the function's first parameter etc.

A sizetf function can be specified, so that the Verilog compiler can check that the function is being called correctly. (In VCS, you specify the number of bits in the PLI table.)

checktf Example



checktf Example

```
$copy(a,b);
```

```
{usertask, 0, copy_CHECK, 0, copy_CALL, 0, "$copy",0}
```

veriusertfs

```
$copy call=copy_CALL check=copy_CHECK
```

VCS PLI table

checktf function

Number of parameters

```
int copy_CHECK (int data, int reason)
{
  if ( tfnump() != 2 )
    tf_error("$copy should have exactly two parameters");
  ...
}
```

A-27 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

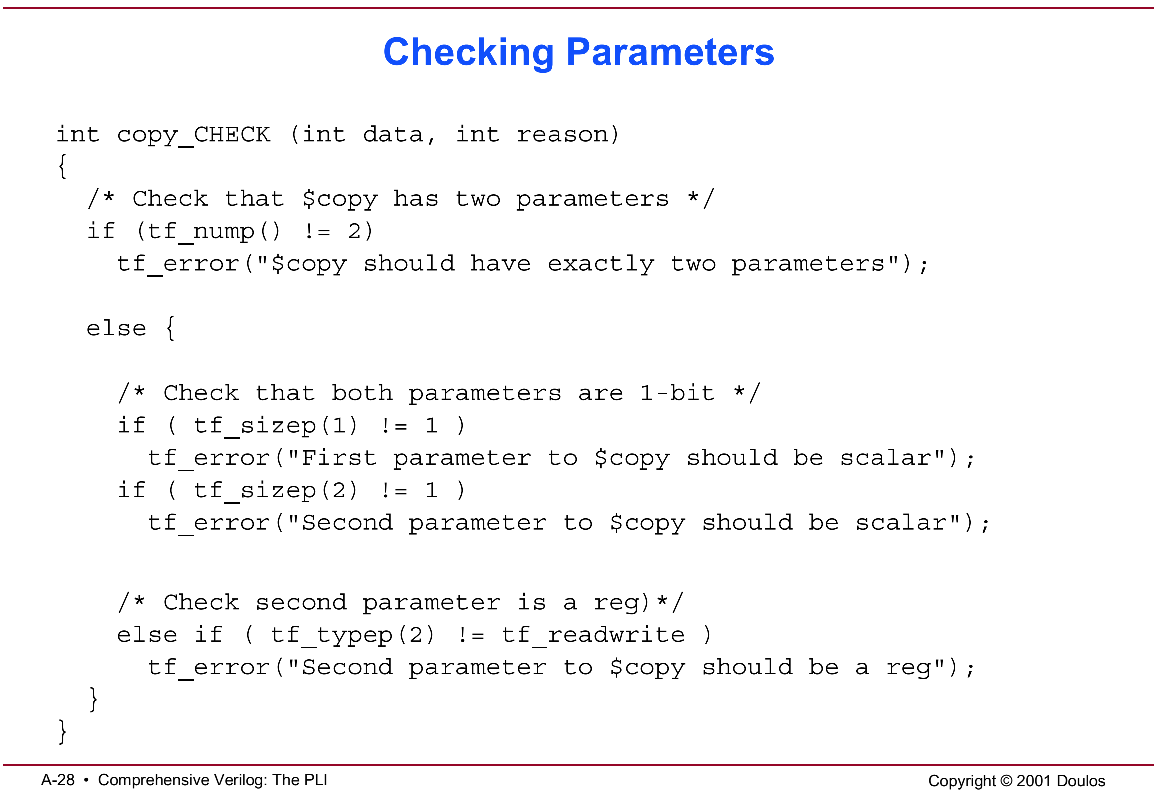
Notes:

The function copy_CHECK is defined to be the checktf routine for $copy in the veriusertfs table (Verilog-XL or ModelSim) or PLI table (VCS).

The Verilog compiler will call copy_CHECK once at compile time for each reference to $copy in the Verilog code. Its purpose is to check that the person who wrote the Verilog code has used $copy correctly - that it has two scalar parameters, the second of which must be able to be written to.

The example is continued on the following page.

Checking Parameters



Checking Parameters

```
int copy_CHECK (int data, int reason)
{
  /* Check that $copy has two parameters */
  if (tf_nump() != 2)
    tf_error("$copy should have exactly two parameters");

  else {

    /* Check that both parameters are 1-bit */
    if ( tf_sizep(1) != 1 )
      tf_error("First parameter to $copy should be scalar");
    if ( tf_sizep(2) != 1 )
      tf_error("Second parameter to $copy should be scalar");


    /* Check second parameter is a reg)*/
    else if ( tf_typep(2) != tf_readwrite )
      tf_error("Second parameter to $copy should be a reg");
  }
}
```

A-28 • Comprehensive Verilog: The PLI

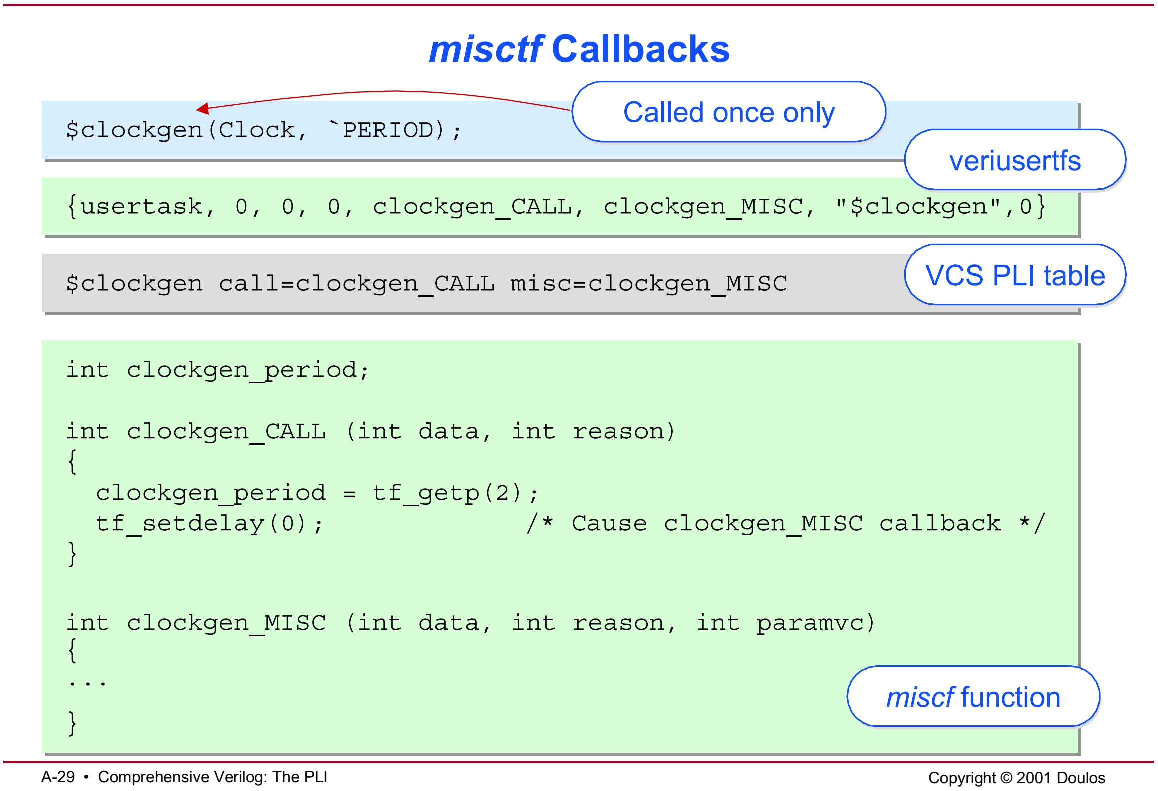
Copyright © 2001 Doulos

Notes:

copy_CHECK calls various TF routines to do its work:

- tf_nump returns the number of parameters used in the reference to $copy.
- tf_sizep(N) returns the size in bits for the Nth parameter of $copy.
- tf_typep(N) returns the type of the Nth parameter. The value tf_readwrite indicates that the parameter is a register type. The other values that can be returned are tf_nullparam, tf_string, tf_readonly, tf_readonlyreal, tf_readwritereal. These values are described in the IEEE 1364-1995 manual.

misctf Callbacks



misctf Callbacks

```
$clockgen(Clock, `PERIOD);
```

Called once only

```
{usertask, 0, 0, 0, clockgen_CALL, clockgen_MISC, "$clockgen",0}
```

veriusertfs

```
$clockgen call=clockgen_CALL misc=clockgen_MISC
```

VCS PLI table

```
int clockgen_period;

int clockgen_CALL (int data, int reason)
{
  clockgen_period = tf_getp(2);
  tf_setdelay(0);              /* Cause clockgen_MISC callback */
}

int clockgen_MISC (int data, int reason, int paramvc)
{
...
}
```

miscf function

A-29 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

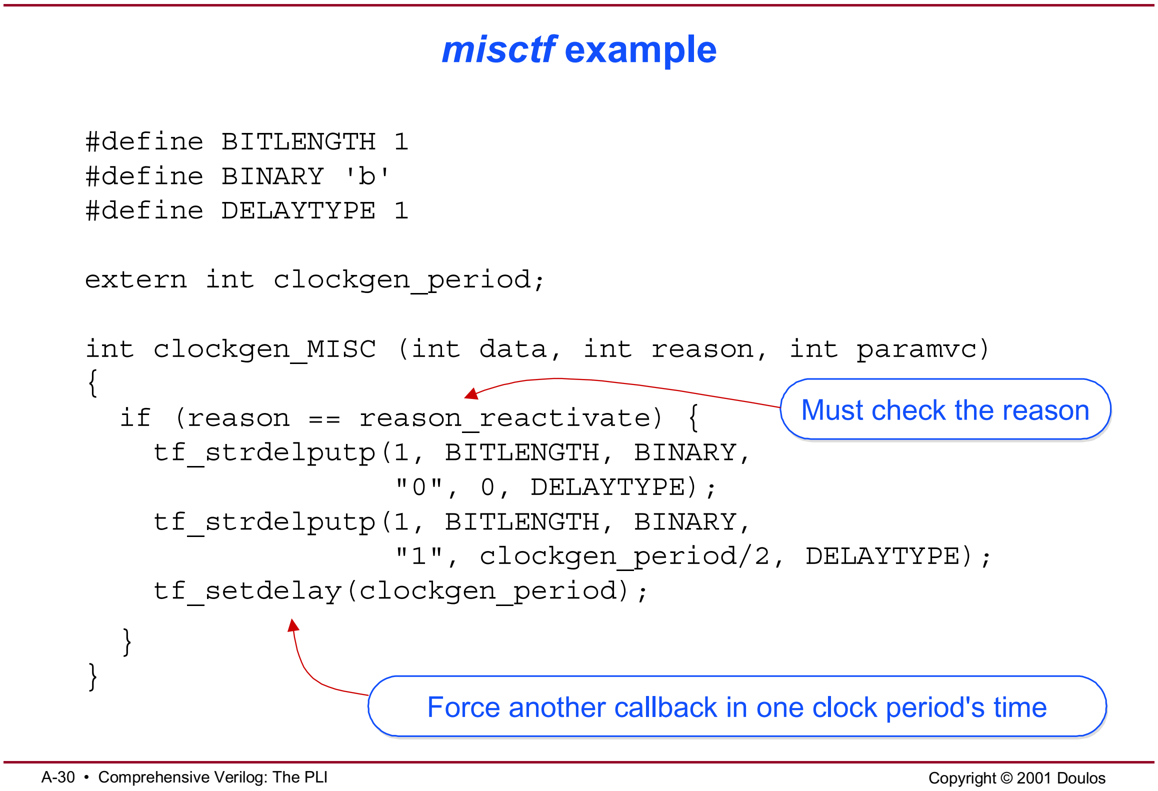
Notes:

If a misctf routine is associated with a user-defined system task or function, it may be called for a number of reasons. For example, when simulation starts or finishes or when a $stop is executed. The reason is passed as the second argument. When writing a misctf routine it is important to check the reason, and act appropriately.

By calling tf_setdelay, it is possible for a PLI application to cause the misctf routine to be called after a specified delay. The reason will be reason_reactivate. This mechanism can be used to generate test stimulus in a PLI application.

Here, the task $clockgen is called once. It retrieves the value of the clock period from the first parameter, and stores it in the global variable clockgen_period. The call to tf_setdelay will cause clockgen_MISC to be called by the simulator at the end of the current simulation time step.

misctf example



misctf example

```
#define BITLENGTH 1
#define BINARY 'b'
#define DELAYTYPE 1

extern int clockgen_period;

int clockgen_MISC (int data, int reason, int paramvc)
{
  if (reason == reason_reactivate) {
    tf_strdelputp(1, BITLENGTH, BINARY,
                  "0", 0, DELAYTYPE);
    tf_strdelputp(1, BITLENGTH, BINARY,
                  "1", clockgen_period/2, DELAYTYPE);
    tf_setdelay(clockgen_period);
  }
}
```

Must check the reason

Force another callback in one clock period's time

A-30 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

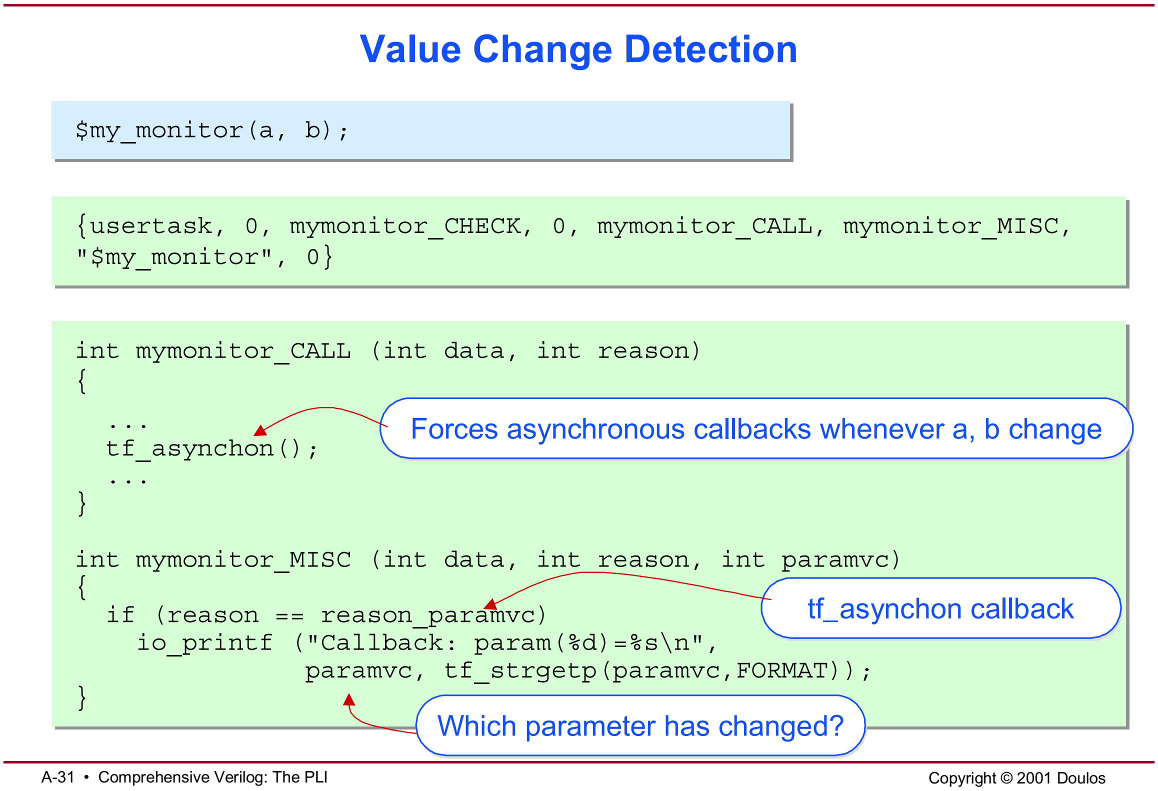
Notes:

clockgen_MISC schedules two events on the first parameter to $clockgen (i.e the Clock reg). The first sets the clock to 0 at the current time (delay of 0); the second sets the clock to 1 after half a period. The delay type argument is 1, indicating "modified transport delay". In other words, any future events on the clock are cancelled.

Having scheduled these two events, clockgen_MISC then causes itself to be called again after a delay corresponding to the clock period.

Note that clockgen_MISC only performs these actions if the reason for it being called is reason_reactivate (i.e. by a previous call to tf_setdelay). It is very important to check the reason in a misctf application.

Value Change Detection



Value Change Detection

```
$my_monitor(a, b);
```

```
{usertask, 0, mymonitor_CHECK, 0, mymonitor_CALL, mymonitor_MISC,
"$my_monitor", 0}
```

```
int mymonitor_CALL (int data, int reason)
{
  ...
  tf_asynchon();
  ...
}

int mymonitor_MISC (int data, int reason, int paramvc)
{
  if (reason == reason_paramvc)
    io_printf ("Callback: param(%d)=%s\n",
               paramvc, tf_strgetp(paramvc,FORMAT));
}
```

Forces asynchronous callbacks whenever a, b change

tf_asynchon callback

Which parameter has changed?

A-31 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

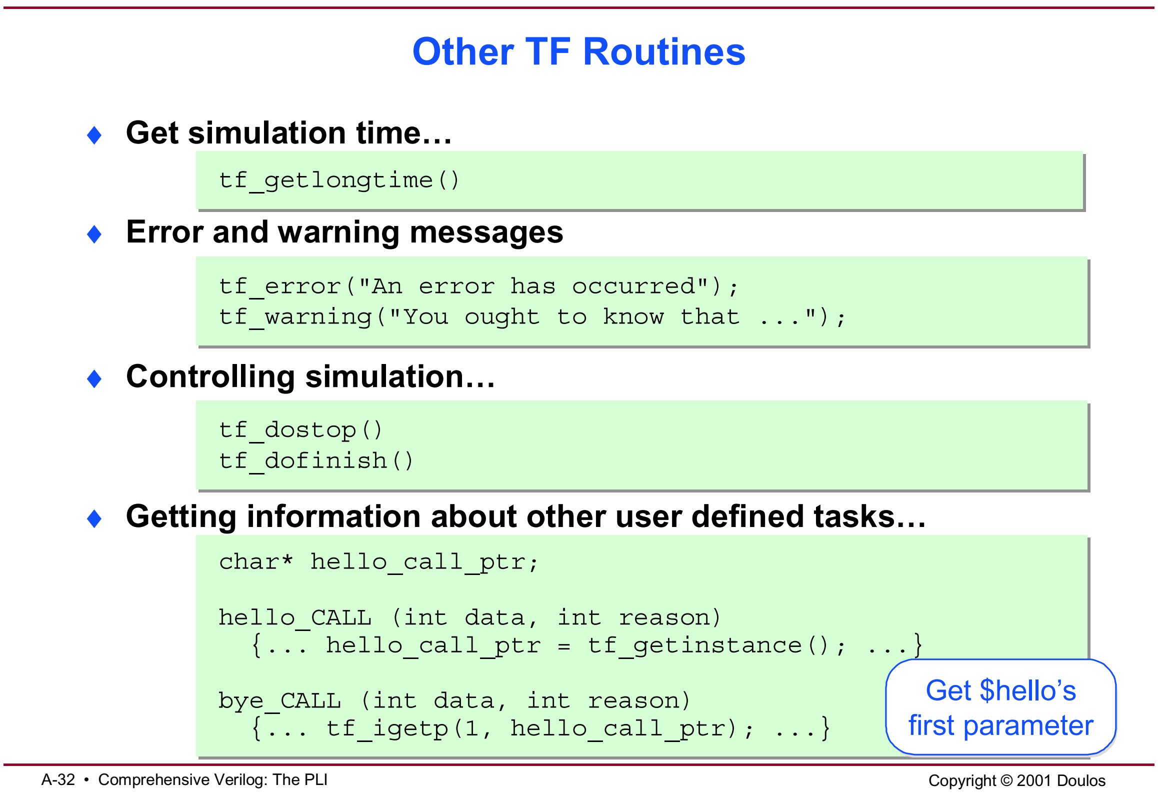
Notes:

This example shows that a user-defined system task can be made to be called asynchronously, like the built-in system task $monitor.

The C function $my_monitor is called only once, and it calls tf_asynchon. This causes the misctf function mymonitor_MISC to be called whenever the value of a or b changes. (a and b are the parameters in the call to $my_monitor.)

When mymonitor _MISC is called in this way, the reason argument will be the pre-defined value reason_paramvc, indicating a parameter has changed value, and the paramvc argument will indicate which of a or b has changed.

Other TF Routines



Other TF Routines

- **Get simulation time…**

```
tf_getlongtime()
```

- **Error and warning messages**

```
tf_error("An error has occurred");
tf_warning("You ought to know that ...");
```

- **Controlling simulation…**

```
tf_dostop()
tf_dofinish()
```

- **Getting information about other user defined tasks…**

```
char* hello_call_ptr;

hello_CALL (int data, int reason)
  {... hello_call_ptr = tf_getinstance(); ...}

bye_CALL (int data, int reason)
  {... tf_igetp(1, hello_call_ptr); ...}
```

Get $hello's first parameter

A-32 • Comprehensive Verilog: The PLI

Notes:

Here are some further examples of TF routines.

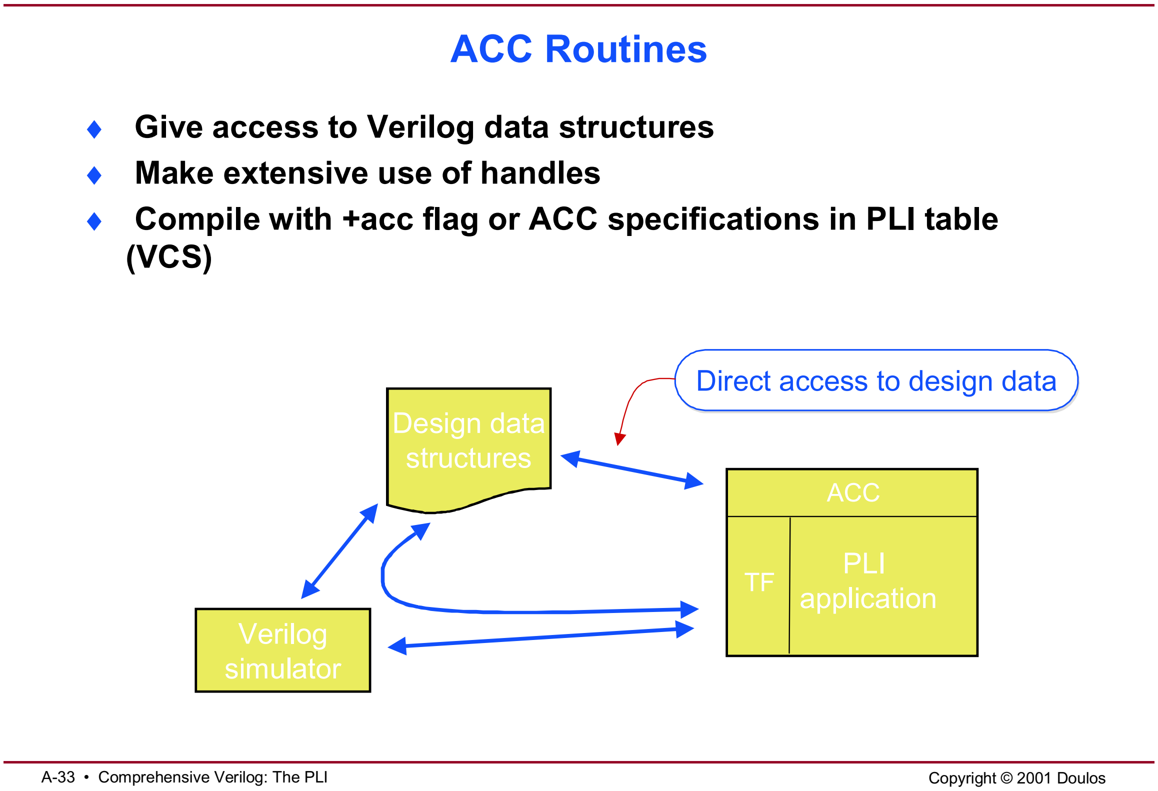
tf_warning and tf_error report warnings and errors in the same format as the simulator usually reports warnings and errors.

tf_dostop and tf_dofinish have the same effect as the Verilog built in system tasks $stop and $finish.

Many TF routines come in pairs. For example tf_getp and tf_igetp. The latter form enables one PLI application to use the parameters passed to another PLI application. For example, $bye could find out the values of $hello's parameters, even though they have different calltf functions.

Full details of all the TF routines can be found in the IEEE 1364 LRM. They may also be described in the documentation for the Verilog simulator you are using.

ACC Routines



ACC Routines

- Give access to Verilog data structures
- Make extensive use of handles
- Compile with +acc flag or ACC specifications in PLI table (VCS)

Direct access to design data

Design data structures

ACC

TF

PLI application

Verilog simulator

A-33 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

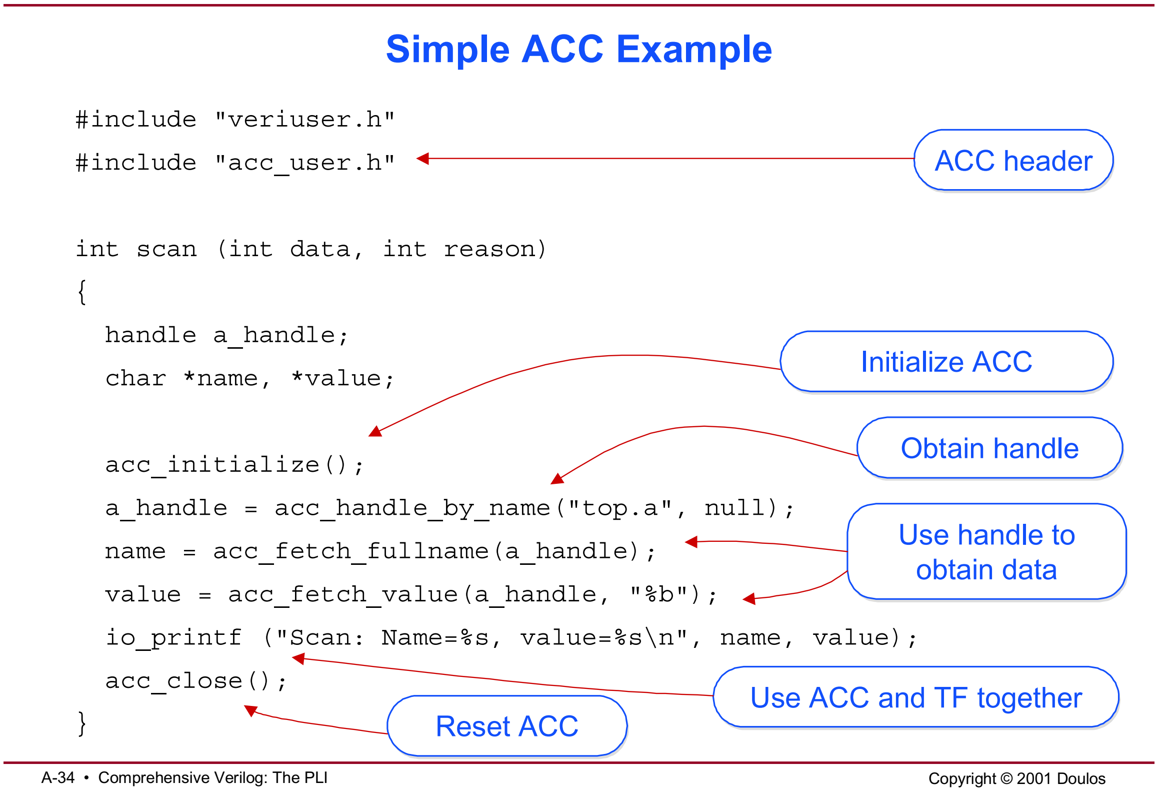
The ACC routines are used to access the data structures of a compiled Verilog hierarchy during simulation. The ACC routines are able to do this directly. With the TF routines, values have to be passed to the PLI application using user-defined system task parameters.

The ACC routines make extensive use of handles. Once the handle of an object such as a net or a module is known, information about that object can be obtained.

Some simulators require special command line parameters or configuration options to enable the ACC routines to be used. For example, Synopsys VCS requires the +acc command line option or ACC specifications in the PLI table.

You will need to refer to the documentation for the simulator you are using for details.

Simple ACC Example



Simple ACC Example

```
#include "veriuser.h"
#include "acc_user.h"

int scan (int data, int reason)
{
  handle a_handle;
  char *name, *value;

  acc_initialize();
  a_handle = acc_handle_by_name("top.a", null);
  name = acc_fetch_fullname(a_handle);
  value = acc_fetch_value(a_handle, "%b");
  io_printf ("Scan: Name=%s, value=%s\n", name, value);
  acc_close();
}
```

ACC header
Initialize ACC
Obtain handle
Use handle to obtain data
Use ACC and TF together
Reset ACC

A-34 • Comprehensive Verilog: The PLI

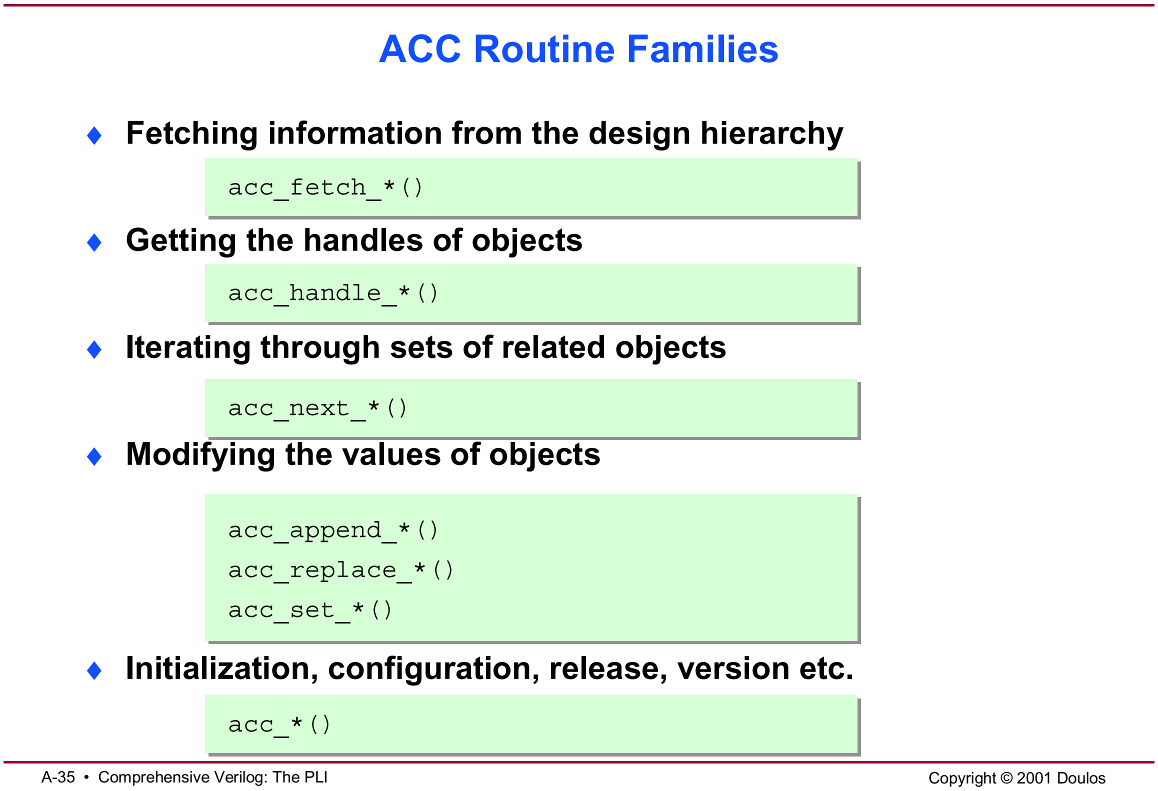
Copyright © 2001 Doulos

Notes:

The example above shows a simple ACC PLI application. Each application should first call acc_initialize. Before finishing, it should call acc_close() to free memory etc.

The application gets the handle corresponding to the name top.a and uses it to obtain the full hierarchical name and the value of that object. These are then printed.

ACC Routine Families



ACC Routine Families

- **Fetching information from the design hierarchy**

```
acc_fetch_*()
```

- **Getting the handles of objects**

```
acc_handle_*()
```

- **Iterating through sets of related objects**

```
acc_next_*()
```

- **Modifying the values of objects**

```
acc_append_*()
acc_replace_*()
acc_set_*()
```

- **Initialization, configuration, release, version etc.**

```
acc_*()
```

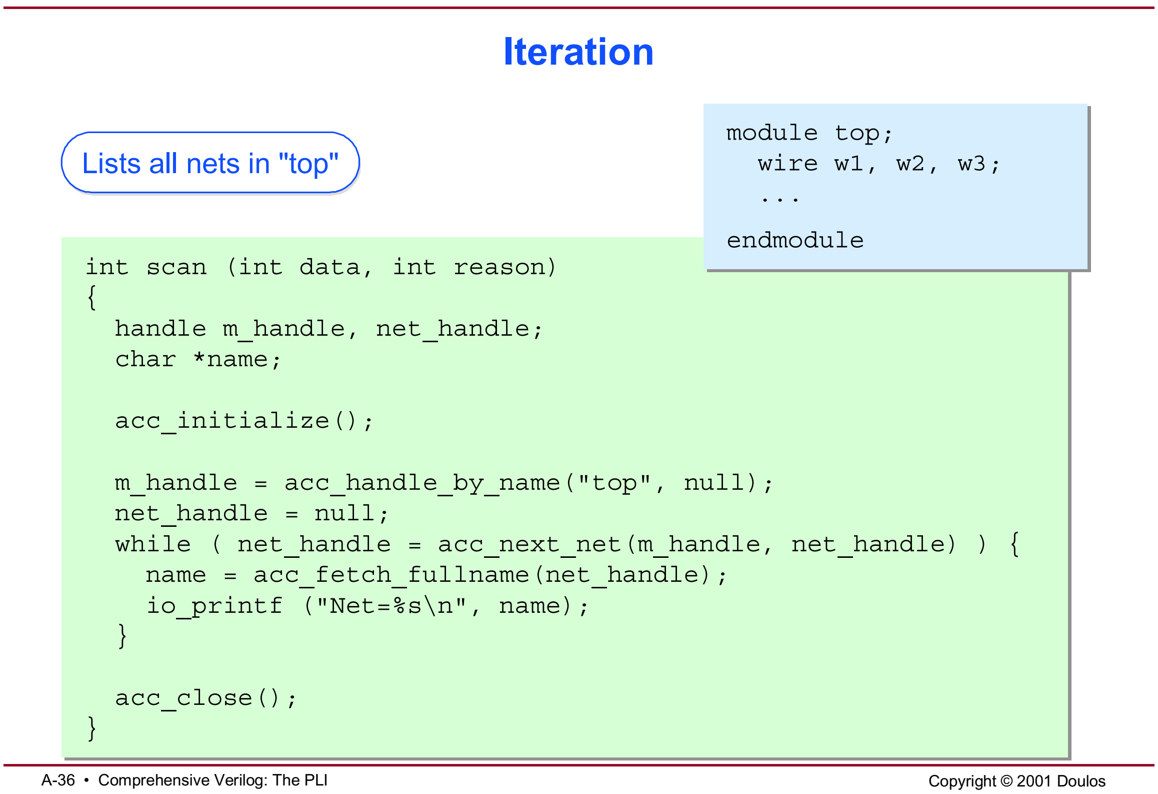
A-35 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

There are various families of ACC routines. Full details of all the ACC routines can be found in the IEEE 1364 LRM. They may also be described in the documentation for the Verilog simulator you are using.

Iteration



Iteration

Lists all nets in "top"

```
module top;
  wire w1, w2, w3;
  ...
endmodule
```

```
int scan (int data, int reason)
{
  handle m_handle, net_handle;
  char *name;

  acc_initialize();

  m_handle = acc_handle_by_name("top", null);
  net_handle = null;
  while ( net_handle = acc_next_net(m_handle, net_handle) ) {
    name = acc_fetch_fullname(net_handle);
    io_printf ("Net=%s\n", name);
  }

  acc_close();
}
```

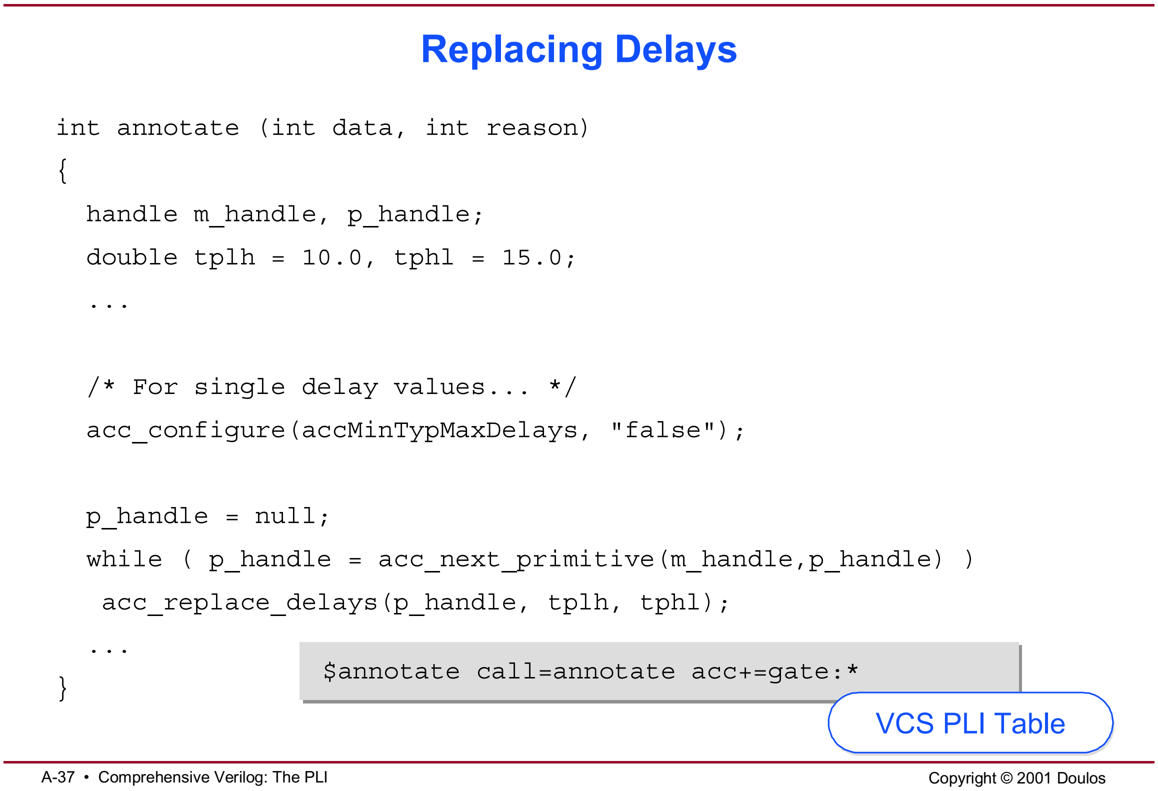
A-36 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

Iteration is common when using ACC routines. This example shows how handles are used to iterate through a number of similar objects. The function scan prints the names of all the nets in the module top.

Replacing Delays



Replacing Delays

```
int annotate (int data, int reason)
{
  handle m_handle, p_handle;
  double tplh = 10.0, tphl = 15.0;
  ...

  /* For single delay values... */
  acc_configure(accMinTypMaxDelays, "false");

  p_handle = null;
  while ( p_handle = acc_next_primitive(m_handle,p_handle) )
   acc_replace_delays(p_handle, tplh, tphl);
  ...
}
```

```
$annotate call=annotate acc+=gate:*
```

VCS PLI Table

A-37 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

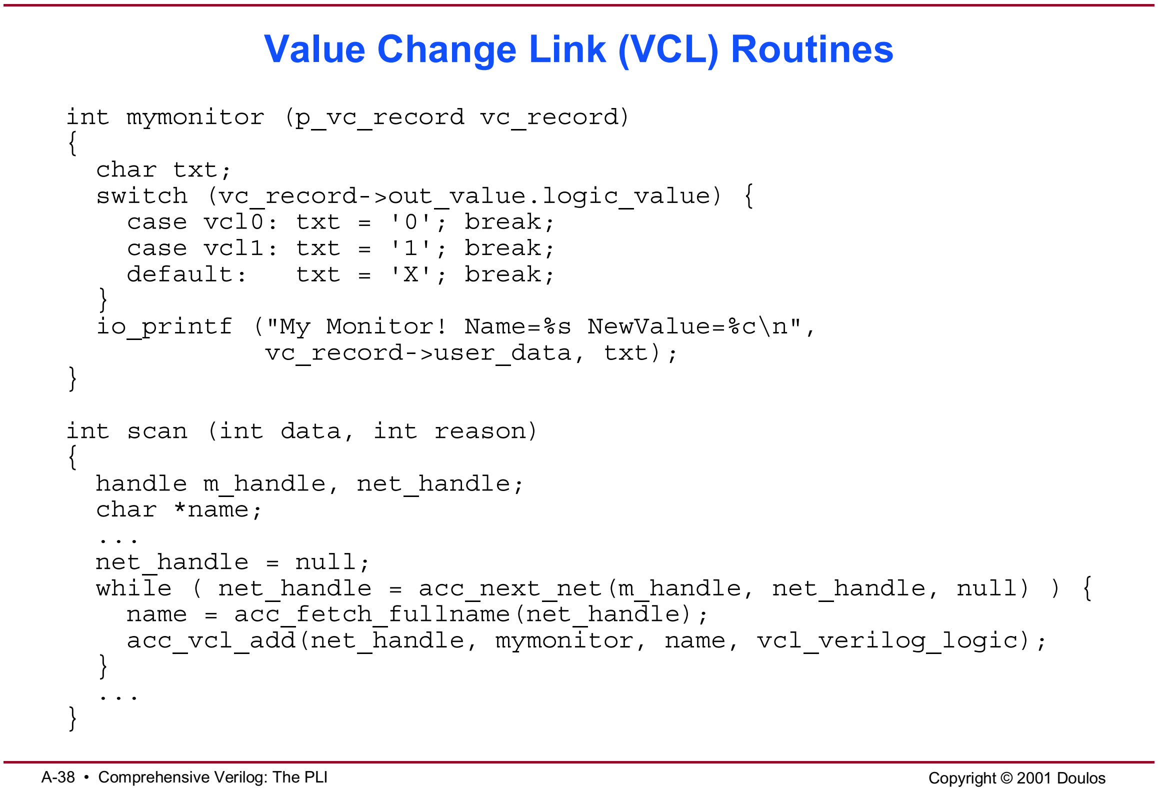
Notes:

This example shows that the PLI can be used not only to find information about the data structures, but to make changes there.

By default, all built-in and user-defined primitives gave a delay of 0. This application changes the delays in a module to a rise delay of 10 and a fall delay of 15.

(For Synopsys VCS, the PLI table entry acc+=gate:* enables back-annotation of gate delays for all modules in the design.)

Value Change Link (VCL) Routines



Value Change Link (VCL) Routines

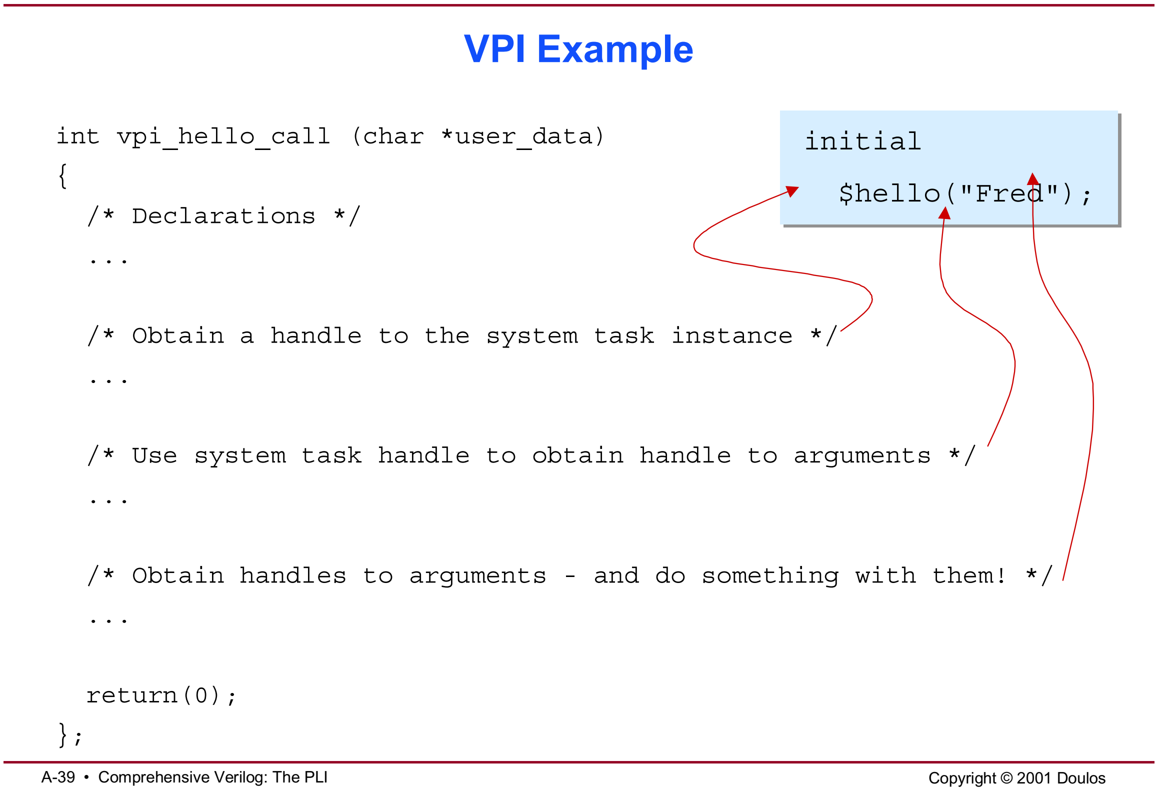
```
int mymonitor (p_vc_record vc_record)
{
  char txt;
  switch (vc_record->out_value.logic_value) {
    case vcl0: txt = '0'; break;
    case vcl1: txt = '1'; break;
    default:   txt = 'X'; break;
  }
  io_printf ("My Monitor! Name=%s NewValue=%c\n",
             vc_record->user_data, txt);
}

int scan (int data, int reason)
{
  handle m_handle, net_handle;
  char *name;
  ...
  net_handle = null;
  while ( net_handle = acc_next_net(m_handle, net_handle, null) ) {
    name = acc_fetch_fullname(net_handle);
    acc_vcl_add(net_handle, mymonitor, name, vcl_verilog_logic);
  }
  ...
}
```

Notes:

The Value Change Link routines are part of the ACC routines. They allow a PLI application to monitor simulation value changes of selected objects.

This example shows how to write your own version of $monitor. The scan function calls acc_vcl_add for all the nets in the module, and this causes the C function mymonitor to be called whenever the value of one of these nets changes during simulation.

VPI Example



VPI Example

```
int vpi_hello_call (char *user_data)
{
  /* Declarations */
  ...

  /* Obtain a handle to the system task instance */
  ...

  /* Use system task handle to obtain handle to arguments */
  ...

  /* Obtain handles to arguments - and do something with them! */
  ...

  return(0);
};
```

```
initial
  $hello("Fred");
```

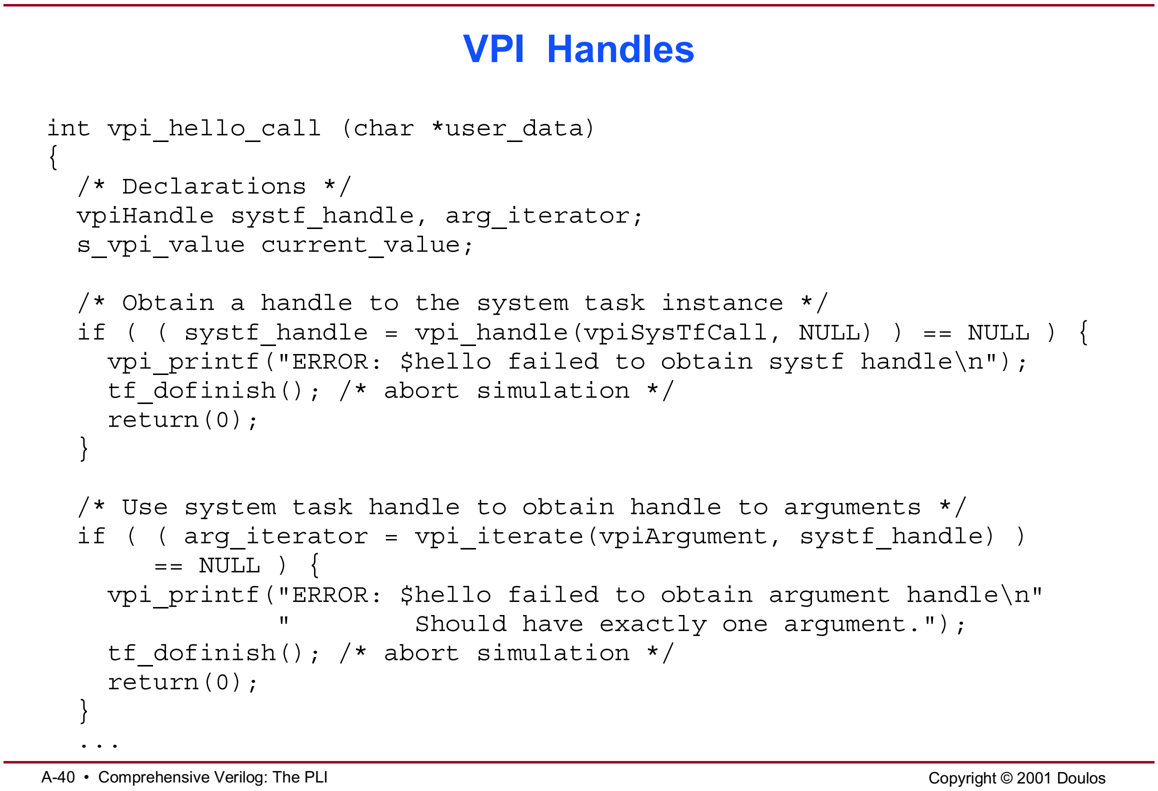
A-39 • Comprehensive Verilog: The PLI

Notes:

In this and the following slides, a simple VPI application is presented. This example simply prints some text, including a string which has been passed as an argument to the user-defined system task $hello.

The general outline of any VPI application is the same as is outlined here. First, the handle to the specific user-defined system task call is obtained. This handle is then used to obtain the handle to the user-defined system task call's arguments. Now the argument values can be retrieved and the actioned.

VPI Handles



VPI Handles

```
int vpi_hello_call (char *user_data)
{
  /* Declarations */
  vpiHandle systf_handle, arg_iterator;
  s_vpi_value current_value;

  /* Obtain a handle to the system task instance */
  if ( ( systf_handle = vpi_handle(vpiSysTfCall, NULL) ) == NULL ) {
    vpi_printf("ERROR: $hello failed to obtain systf handle\n");
    tf_dofinish(); /* abort simulation */
    return(0);
  }

  /* Use system task handle to obtain handle to arguments */
  if ( ( arg_iterator = vpi_iterate(vpiArgument, systf_handle) )
       == NULL ) {
    vpi_printf("ERROR: $hello failed to obtain argument handle\n"
               "        Should have exactly one argument.");
    tf_dofinish(); /* abort simulation */
    return(0);
  }
  ...
```

A-40 • Comprehensive Verilog: The PLI

Notes:

The VPI relies heavily on the use of handles. The function vpi_handle returns a value of type vpiHandle. The value is NULL if there is an error.

The following call gets the handle to the specific invocation of $hello that caused the application to be executed:

```
systf_handle = vpi_handle(vpiSysTfCall, NULL)
```

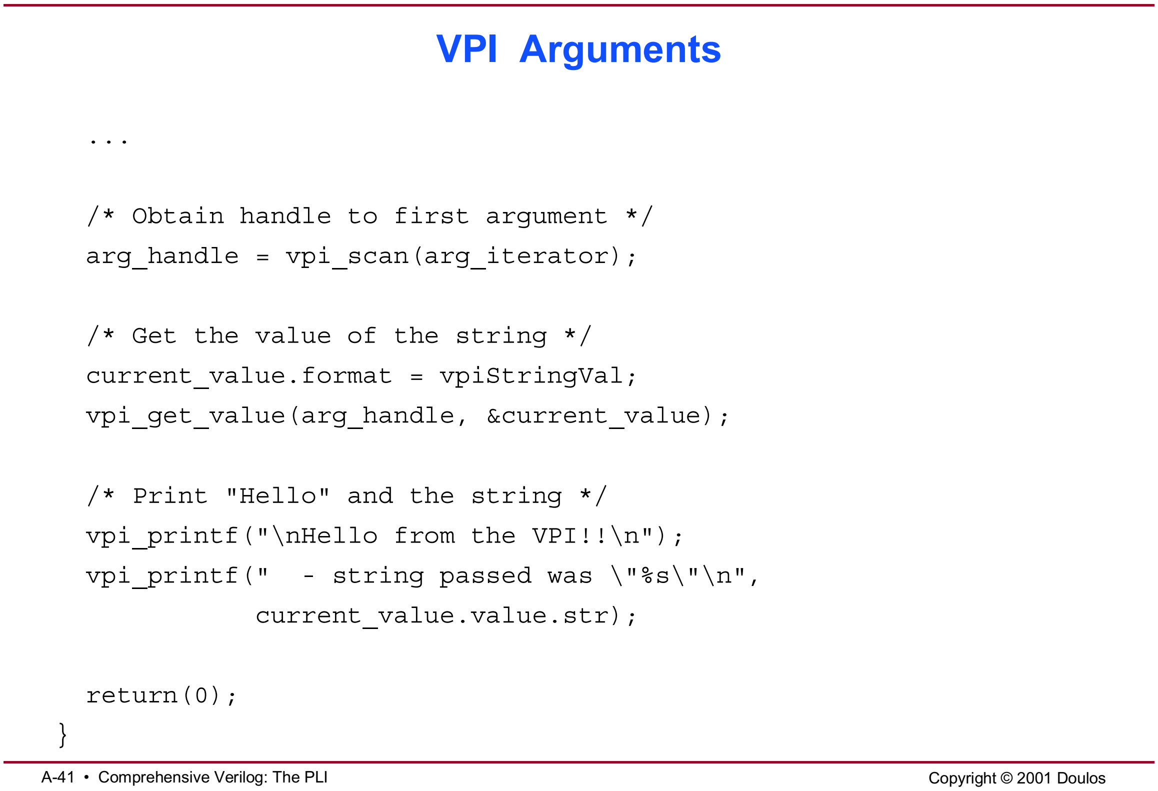
Then the handle obtained is used to obtain the handle to the arguments:

```
arg_iterator = vpi_iterate(vpiArgument, systf_handle)
```

This would return NULL if there were no arguments:

```
$hello;
```

VPI Arguments



VPI Arguments

```
    ...

    /* Obtain handle to first argument */
    arg_handle = vpi_scan(arg_iterator);

    /* Get the value of the string */
    current_value.format = vpiStringVal;
    vpi_get_value(arg_handle, &current_value);

    /* Print "Hello" and the string */
    vpi_printf("\nHello from the VPI!!\n");
    vpi_printf("  - string passed was \"%s\"\n",
               current_value.value.str);

    return(0);
  }
```

A-41 • Comprehensive Verilog: The PLI

Copyright © 2001 Doulos

Notes:

Having obtained the handle to the arguments, you can iterate through them using vpi_scan. In this example, there is only one argument, so vpi_scan is called just once to obtain the handle for the one and only argument.

To get the value of the argument (which is a Verilog string) in the format of a null-terminated C string, the function vpi_get_value is called. This requires two arguments: the argument handle and a pointer to a struct.

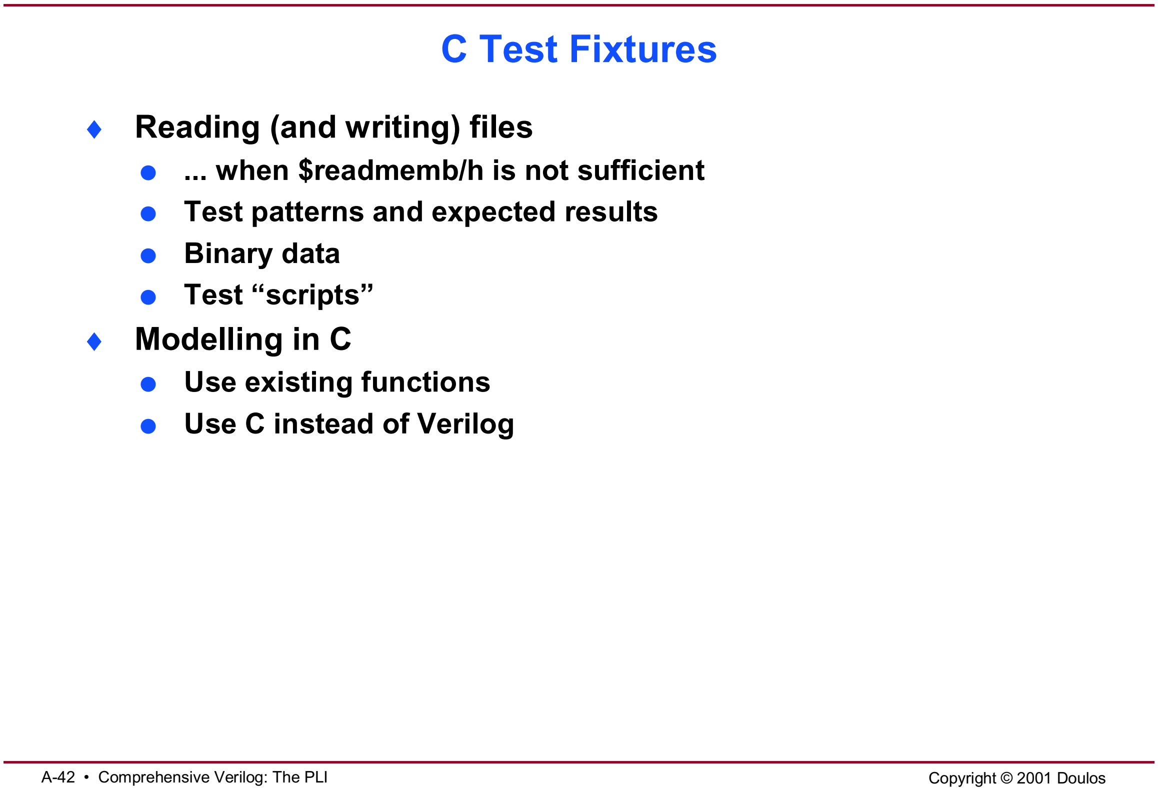
```
typedef struct t_vpi_value {
  int format;
  union {
    char *str;
    int scalar;
```

```
      double real;
      ...
    } value;
  } s_vpi_value, *p_vpi_value;
```

The format field indicates the format in which to retrieve the argument's value. The value field, which is a union, will contain the value in the specified format.

Finally, vpi_printf is similar to the stand C printf function, except that it writes to the Verilog simulator's console and/or log file.

C Test Fixtures



C Test Fixtures

- Reading (and writing) files
  - ... when $readmemb/h is not sufficient
  - Test patterns and expected results
  - Binary data
  - Test "scripts"
- Modelling in C
  - Use existing functions
  - Use C instead of Verilog

A-42 • Comprehensive Verilog: The PLI

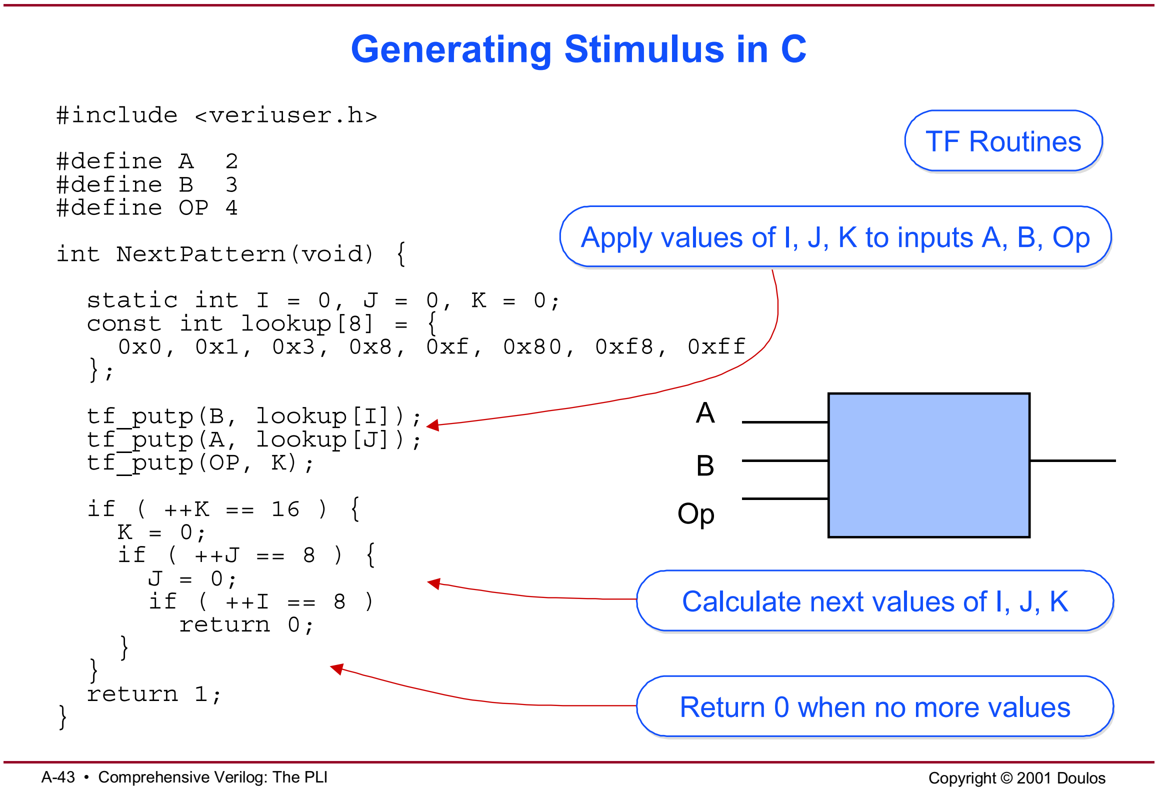
Notes:

In the final part of this PLI introduction we are going to see some techniques that can be used for writing tests in C.

As we have seen, one limitation of the Verilog language is the lack of built in system tasks to read text files. One of the most useful applications of the Verilog PLI is to be able to use the full text processing capabilities of C in a Verilog test fixture. This enables us to read test patterns and expected results from files, and also to read test scripts. It would be much harder to do this without using the PLI.

As well as being able to read (and write) text and binary data files, the PLI enables you to use existing C functions - or write new ones - that model the system in which your design resides. Again, this is often much easier than translating such models into Verilog. It also makes it possible for system engineers, who may have a working knowledge of C, but not of Verilog, to create tests.

Generating Stimulus in C



Generating Stimulus in C

```
#include <veriuser.h>

#define A  2
#define B  3
#define OP 4

int NextPattern(void) {

  static int I = 0, J = 0, K = 0;
  const int lookup[8] = {
    0x0, 0x1, 0x3, 0x8, 0xf, 0x80, 0xf8, 0xff
  };

  tf_putp(B, lookup[I]);
  tf_putp(A, lookup[J]);
  tf_putp(OP, K);

  if ( ++K == 16 ) {
    K = 0;
    if ( ++J == 8 ) {
      J = 0;
      if ( ++I == 8 )
        return 0;
    }
  }
  return 1;
}
```

TF Routines

Apply values of I, J, K to inputs A, B, Op

Calculate next values of I, J, K

Return 0 when no more values

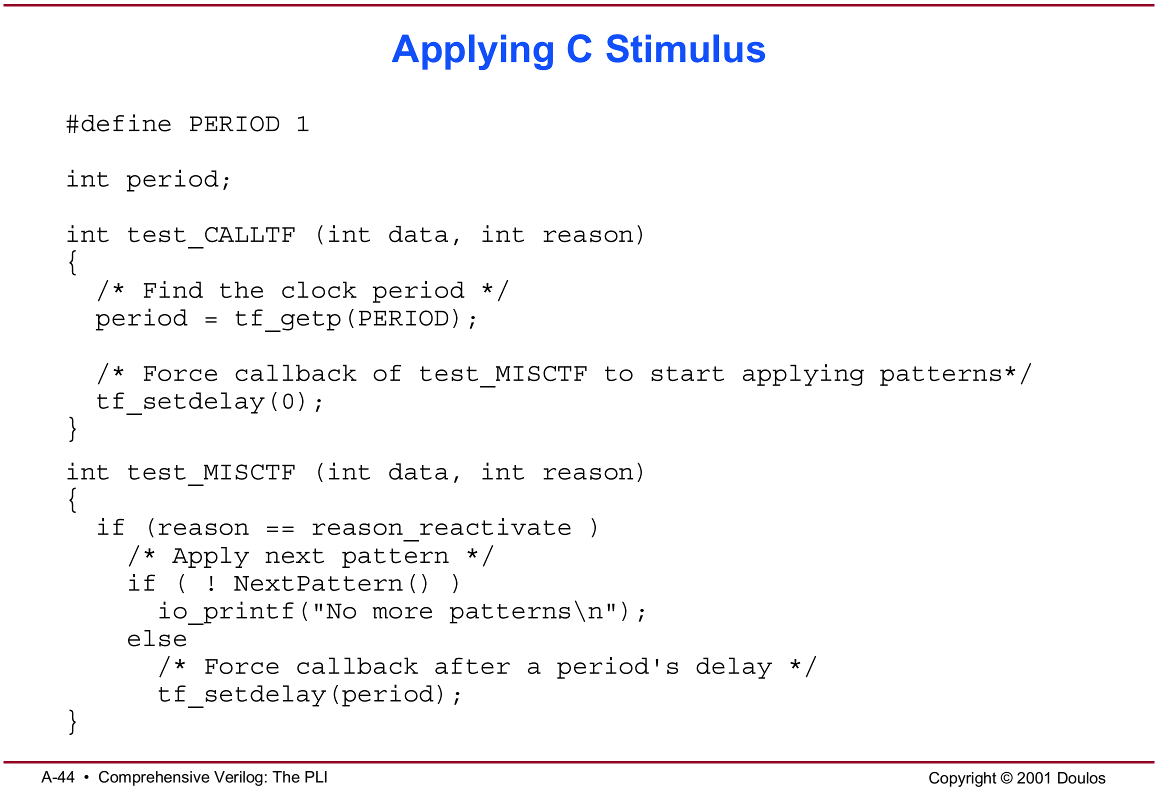
A-43 • Comprehensive Verilog: The PLI

Notes:

Here is a C function that generates the next test pattern in a sequence when it is called. The pattern is applied to the three inputs using tf_putp.

The function returns 1 if a pattern was applied successfully, and 0 when there are no more patterns to be generated.

Applying C Stimulus



Applying C Stimulus

```
#define PERIOD 1

int period;

int test_CALLTF (int data, int reason)
{
  /* Find the clock period */
  period = tf_getp(PERIOD);

  /* Force callback of test_MISCTF to start applying patterns*/
  tf_setdelay(0);
}
int test_MISCTF (int data, int reason)
{
  if (reason == reason_reactivate )
    /* Apply next pattern */
    if ( ! NextPattern() )
      io_printf("No more patterns\n");
    else
      /* Force callback after a period's delay */
      tf_setdelay(period);
}
```

A-44 • Comprehensive Verilog: The PLI

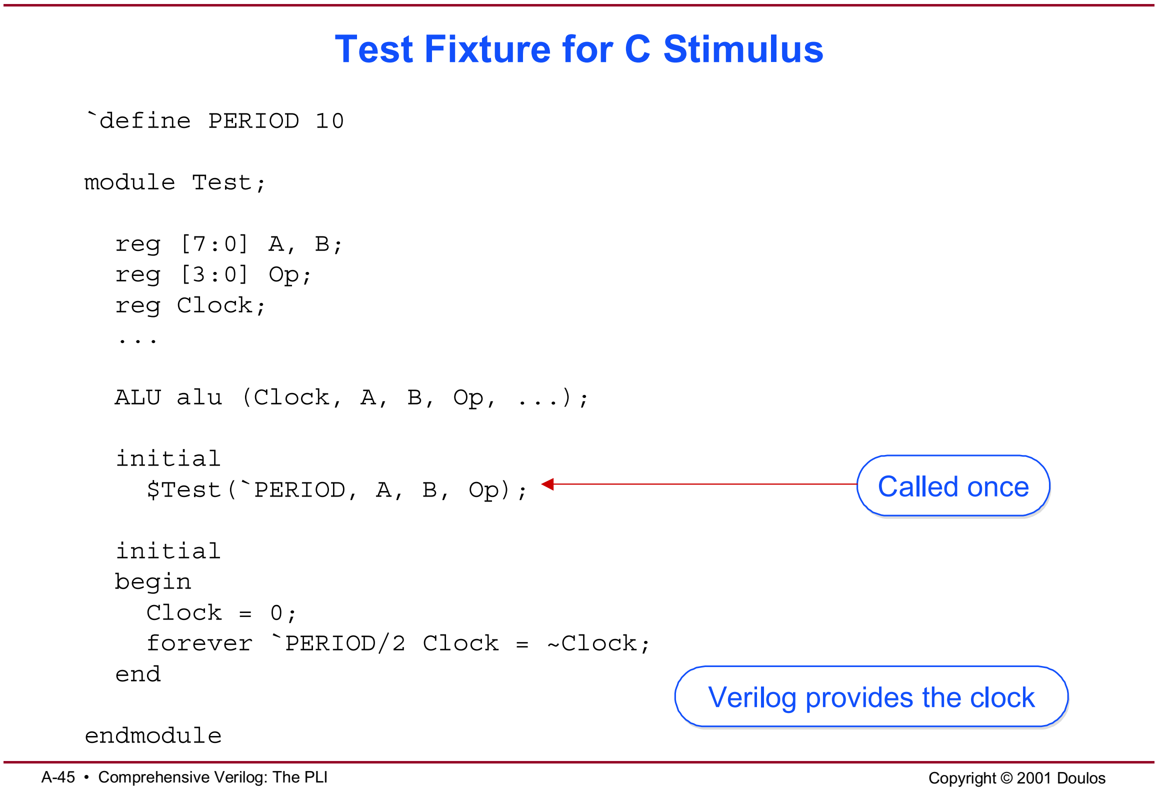
Copyright © 2001 Doulos

Notes:

The test patterns are applied by calling NextPattern repeatedly. This is done by the misctf routine, which also schedules a callback of itself one period after applying the pattern. This continues until the NextPattern function signals that there are no more patterns to apply.

The first pattern is applied by the calltf routine, test_CALLTF, scheduling a callback of the misctf routine, test_MISCTF.

Test Fixture for C Stimulus



Test Fixture for C Stimulus

```
`define PERIOD 10

module Test;

  reg [7:0] A, B;
  reg [3:0] Op;
  reg Clock;
  ...

  ALU alu (Clock, A, B, Op, ...);

  initial
    $Test(`PERIOD, A, B, Op);

  initial
  begin
    Clock = 0;
    forever `PERIOD/2 Clock = ~Clock;
  end

endmodule
```

Called once

Verilog provides the clock

A-45 • Comprehensive Verilog: The PLI

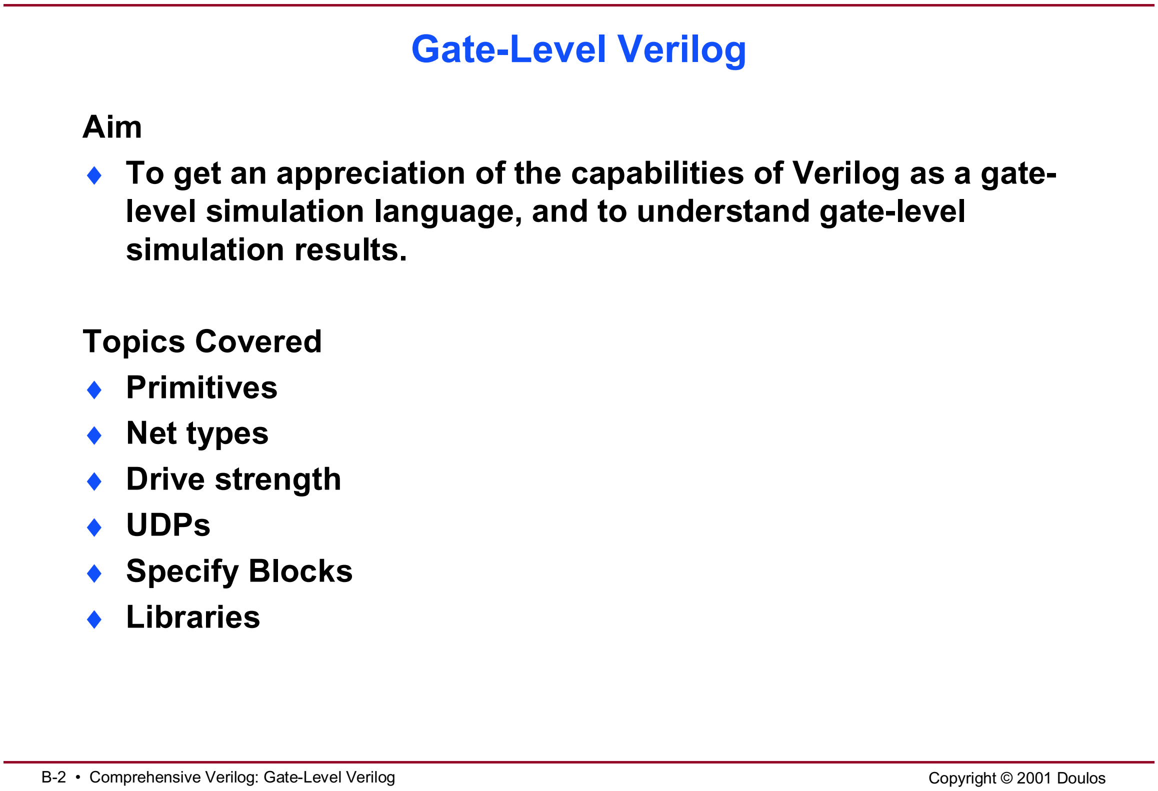
Notes:

Here is the Verilog test fixture that launches the PLI application that applies the patterns. The user-defined system task $test is called just once and this call kicks off the repeated calls to NextPattern, as has been described.

In this example, the clock is generated in the test fixture, although it too could be generated in C.

Appendix B
Gate-Level Verilog

Gate-Level Verilog



Gate-Level Verilog

Aim

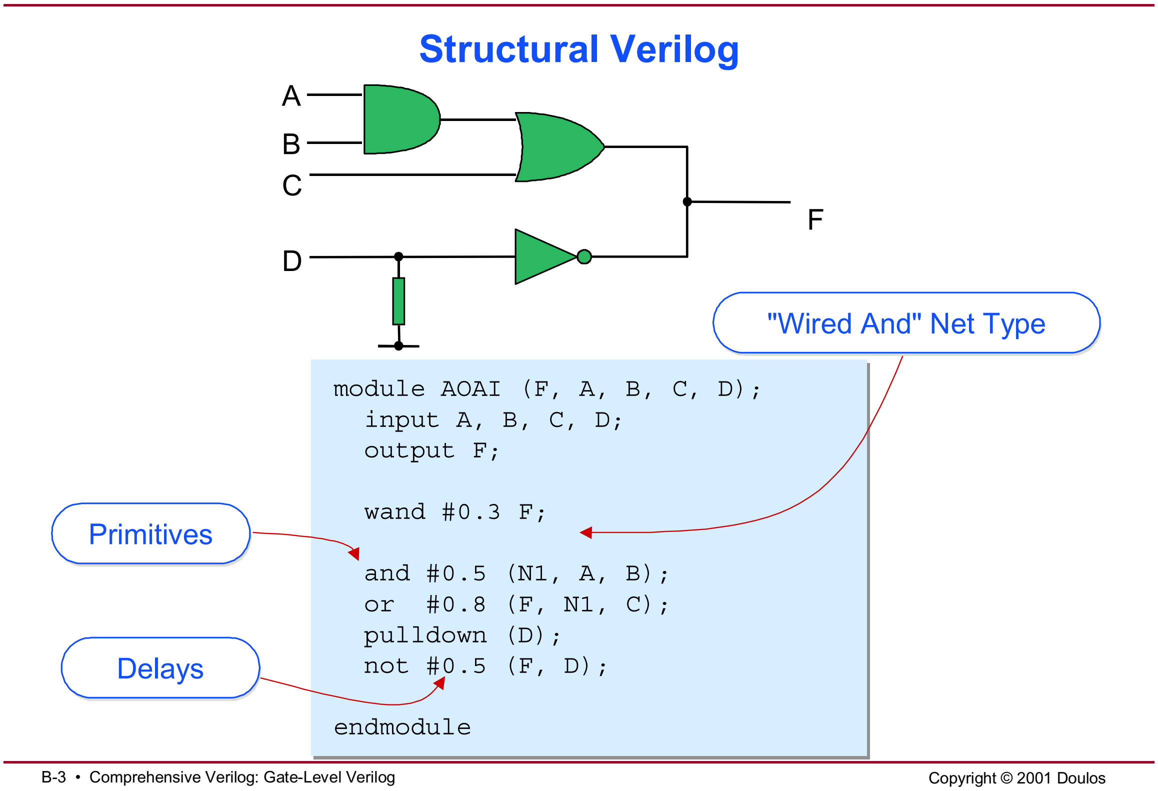
- To get an appreciation of the capabilities of Verilog as a gate-level simulation language, and to understand gate-level simulation results.

Topics Covered

- Primitives
- Net types
- Drive strength
- UDPs
- Specify Blocks
- Libraries

Notes:

Structural Verilog



Structural Verilog

"Wired And" Net Type

Primitives

Delays

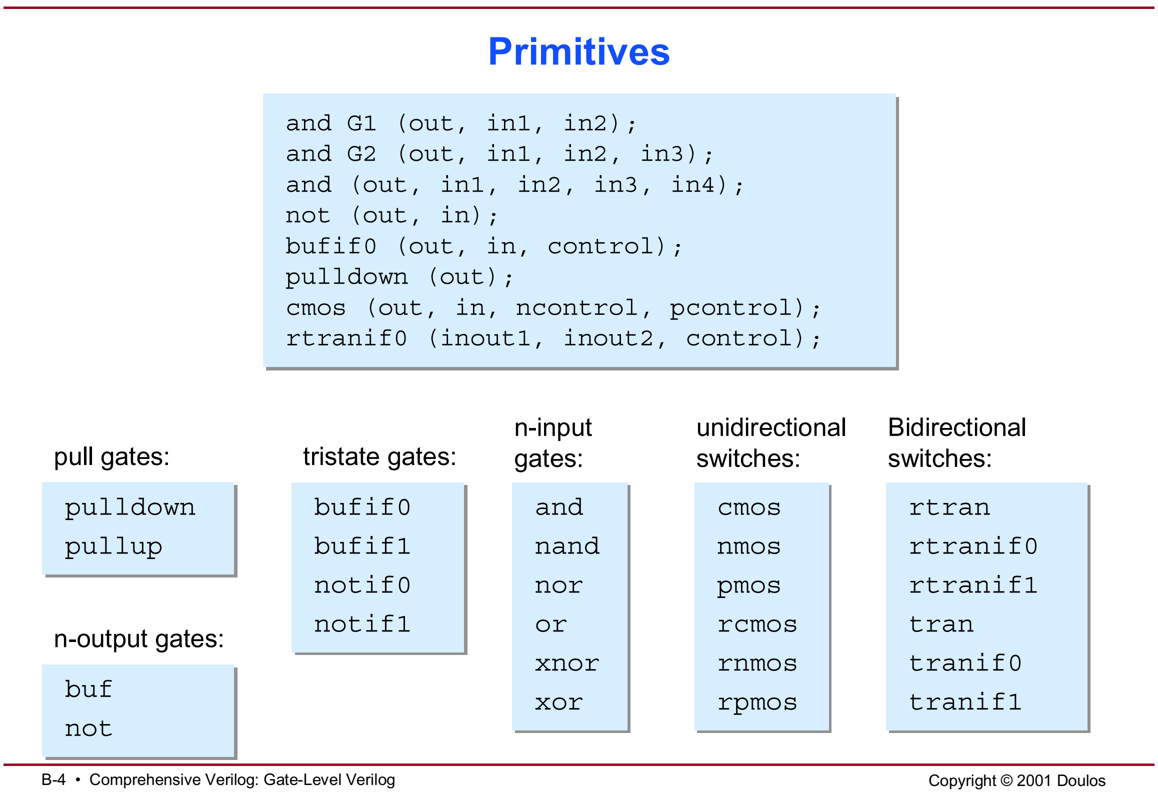
```
module AOAI (F, A, B, C, D);
  input A, B, C, D;
  output F;

  wand #0.3 F;

  and #0.5 (N1, A, B);
  or  #0.8 (F, N1, C);
  pulldown (D);
  not #0.5 (F, D);

endmodule
```

B-3 • Comprehensive Verilog: Gate-Level Verilog

Notes:

A structural description defines an electronic circuit as a set of lower level blocks connected together. These lower level blocks include Verilog's built-in primitive gates and user-defined primitives. The structural parts of the Verilog language support the description of many kinds of technology specific detail such as delays, wired functions, timing constraints, pullups, and charge storage.

Primitives



Primitives

```
and G1 (out, in1, in2);
and G2 (out, in1, in2, in3);
and (out, in1, in2, in3, in4);
not (out, in);
bufif0 (out, in, control);
pulldown (out);
cmos (out, in, ncontrol, pcontrol);
rtranif0 (inout1, inout2, control);
```

pull gates:

```
pulldown
pullup
```

n-output gates:

```
buf
not
```

tristate gates:

```
bufif0
bufif1
notif0
notif1
```

n-input gates:

```
and
nand
nor
or
xnor
xor
```

unidirectional switches:

```
cmos
nmos
pmos
rcmos
rnmos
rpmos
```

Bidirectional switches:

```
rtran
rtranif0
rtranif1
tran
tranif0
tranif1
```

B-4 • Comprehensive Verilog: Gate-Level Verilog

Notes:

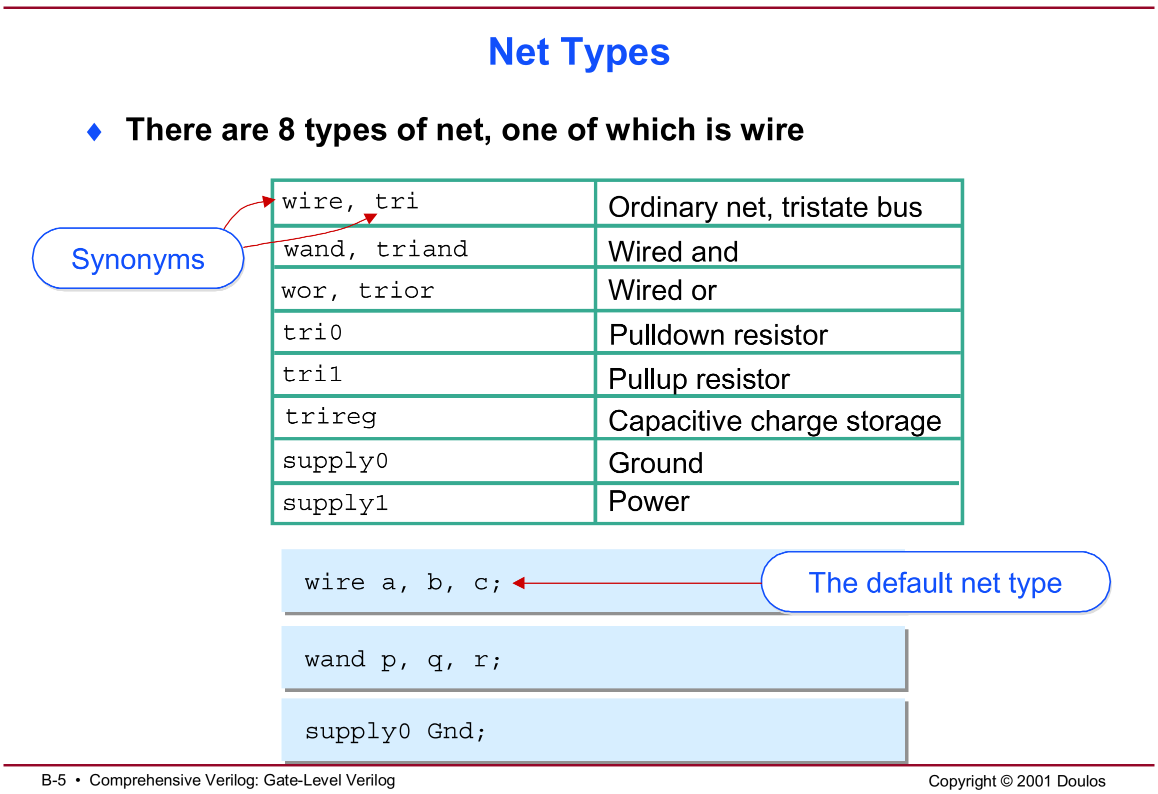
Verilog possesses a set of built in primitives representing basic logic gates, unidirectional and bidirectional switches.

Primitives are instanced just like modules, with a few minor differences of syntax:

1. The primitive names themselves are reserved words in Verilog.
2. Primitive instances may have delays and strengths. This syntax for delays is the same as the syntax for parameter overrides for module instances.
3. Unlike a module instance, an instance name is optional in a primitive instance.

4. The order of the primitives' ports is fixed: outputs always come first, followed by data inputs, and finally control inputs (if any). For the n-input gates, the primitive has exactly one output and can have any number of input ports, as shown for the and primitive opposite. For the n-output gates, the primitive has exactly one input, but any number of outputs (all of which have the same value! This feature has little practical application.)

Net Types



Net Types

- There are 8 types of net, one of which is wire

Synonyms

`wire, tri`	Ordinary net, tristate bus
`wand, triand`	Wired and
`wor, trior`	Wired or
`tri0`	Pulldown resistor
`tri1`	Pullup resistor
`trireg`	Capacitive charge storage
`supply0`	Ground
`supply1`	Power

```
wire a, b, c;
```

The default net type

```
wand p, q, r;
```

```
supply0 Gnd;
```

B-5 • Comprehensive Verilog: Gate-Level Verilog

Notes:

Nets are one of the main three data types in Verilog. (The others are registers and Parameters.) Nets represent electrical connections between points in a circuit.

So far we have used wires to represent electrical connections. In Verilog, a wire is a type of net, but there are also seven other net types, each modeling a different electrical phenomenon. Wire is the default net type (undefined names used in a connection list default to being one bit wires), and in all probability, 99% of the nets in your Verilog descriptions will be wires. However, the remaining types of net are important for modeling certain cases.

wire, tri

The wire (synonymous with tri) models either a point-to-point electrical connection, or a tristate bus with multiple drivers. In the event of a bus conflict, the value of the wire will become 'bx.

wand, wor

The wand (synonym triand) and wor (synonym trior) behave like AND and OR gates respectively when there exists more than one driver on the net.

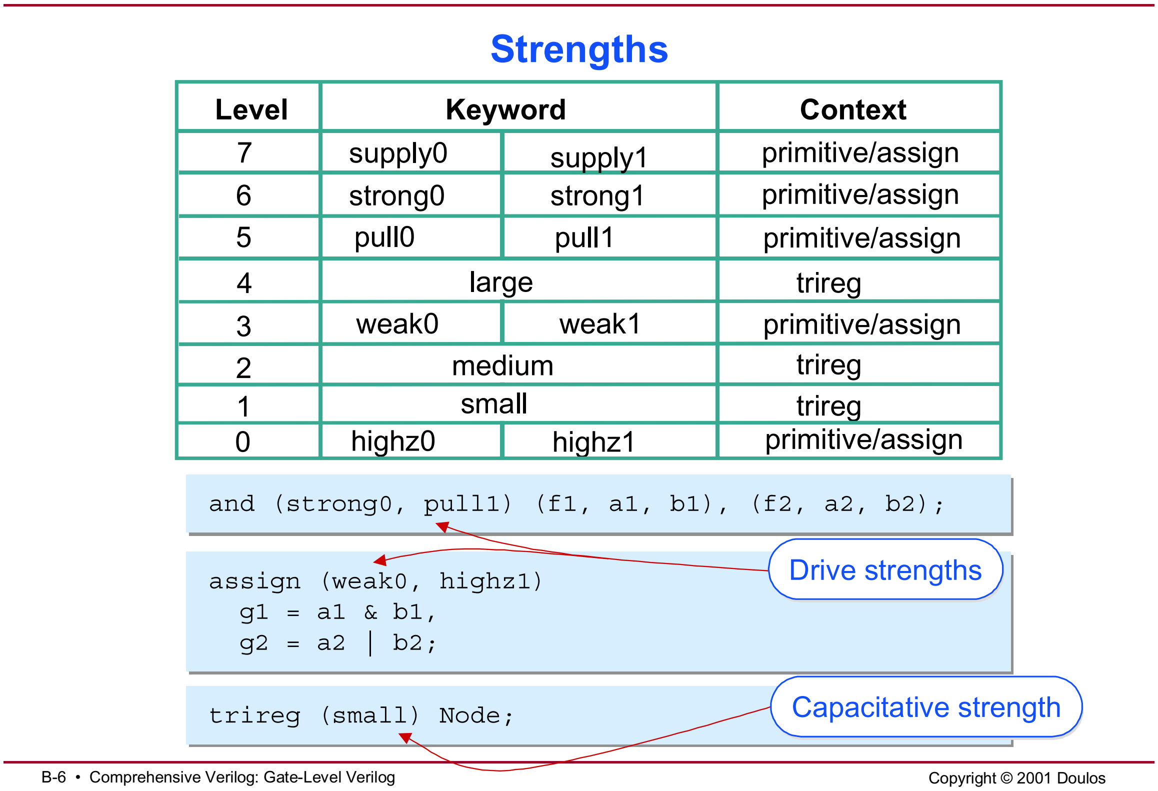
tri0, tri1

The tri0 and tri1 nets behave like wires with a pulldown or pullup resistor attached, i.e. when these nets are not being driven, the net floats to 0 or 1 respectively. In the same situation, a wire would have the value z.

The trireg net models capacitive charge storage and charge decay. When a trireg net is not being driven, it retains its previous value.

The supply0 and supply1 nets represent ground and power rails respectively.

Strengths



Strengths

Level	Keyword		Context
7	supply0	supply1	primitive/assign
6	strong0	strong1	primitive/assign
5	pull0	pull1	primitive/assign
4	large		trireg
3	weak0	weak1	primitive/assign
2	medium		trireg
1	small		trireg
0	highz0	highz1	primitive/assign

```
and (strong0, pull1) (f1, a1, b1), (f2, a2, b2);
```

```
assign (weak0, highz1)
  g1 = a1 & b1,
  g2 = a2 | b2;
```

Drive strengths

```
trireg (small) Node;
```

Capacitative strength

B-6 • Comprehensive Verilog: Gate-Level Verilog

Copyright © 2001 Doulos

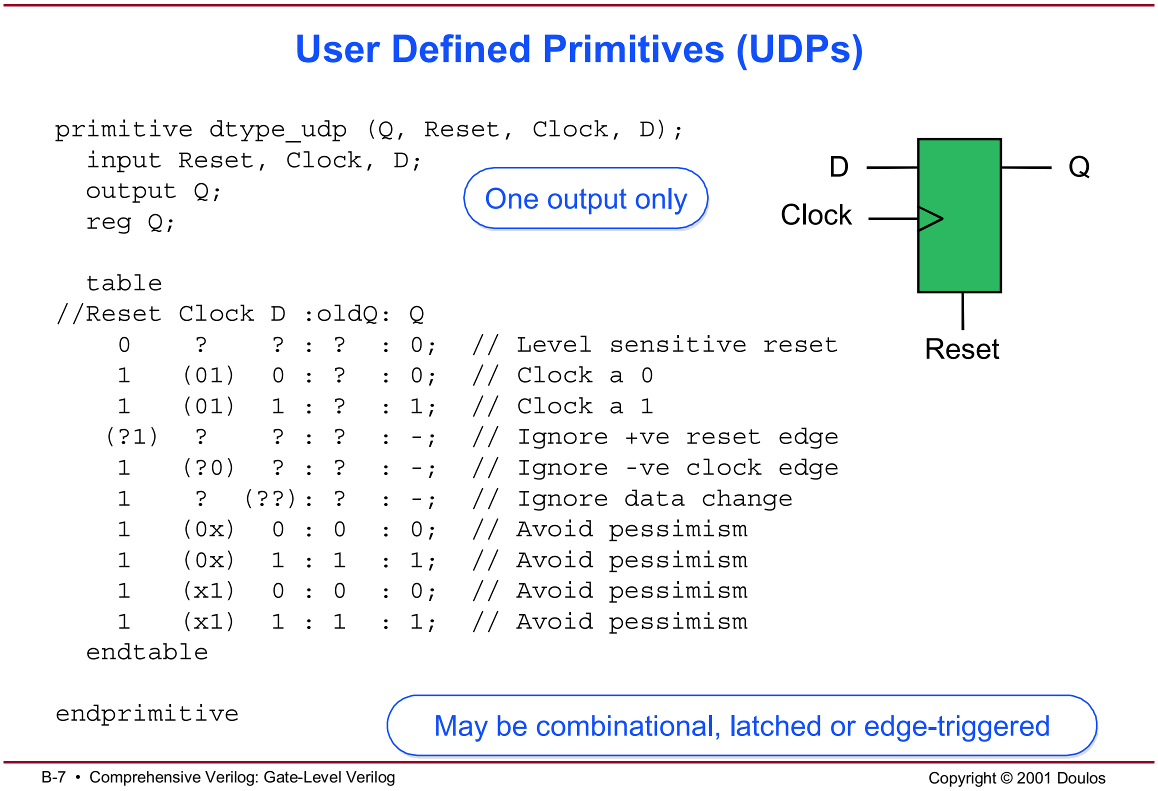
Notes:

Nets are driven either by being connected to the output of a primitive or by being assigned in a continuous assignment. Both primitive and continuous assignments can have a drive strength associated with them which is measured on a scale from 0 to 7, as shown in the table opposite. When there are multiple drivers on a net with conflicting values, the highest strength wins.

- The default drive strength, as used in 99% of all Verilog nets, is strong.
- The supply strength is used for power and ground rails, e.g. net types supply0 and supply1.

- The pull strength is used for pullup and pulldown resistors, e.g. the tri0 and tri1 nets, and the pullup and pulldown primitives.
- The weak strength is a general purpose strength weaker than pull.
- The small, medium and large strengths are exclusively for the trireg net, where they indicate the relative size of the capacitance.
- The highz strength represents high impedance.

User Defined Primitives (UDPs)



User Defined Primitives (UDPs)

```
primitive dtype_udp (Q, Reset, Clock, D);
  input Reset, Clock, D;
  output Q;
  reg Q;

  table
//Reset Clock D :oldQ: Q
    0    ?    ? : ?  : 0;  // Level sensitive reset
    1   (01)  0 : ?  : 0;  // Clock a 0
    1   (01)  1 : ?  : 1;  // Clock a 1
   (?1)  ?    ? : ?  : -;  // Ignore +ve reset edge
    1   (?0)  ? : ?  : -;  // Ignore -ve clock edge
    1    ?  (??): ?  : -;  // Ignore data change
    1   (0x)  0 : 0  : 0;  // Avoid pessimism
    1   (0x)  1 : 1  : 1;  // Avoid pessimism
    1   (x1)  0 : 0  : 0;  // Avoid pessimism
    1   (x1)  1 : 1  : 1;  // Avoid pessimism
  endtable

endprimitive
```

One output only

May be combinational, latched or edge-triggered

Notes:

User defined primitives are the usual way to model library cells consisting of small combinational functions, latches or flipflops. The only cell modeling tasks for which they are unsuitable are arithmetic functions (e.g. adders), memories, and large macrocells or megacells (e.g. UARTS, processor cores, etc).

A UDP effectively extends the set of primitives built into Verilog. It is defined by means of a state table. A UDP is limited to a single output, and practical memory limitations restrict the number of inputs to about 10. The state table cannot include the Z state; a Z arriving on an input of a UDP is treated as an X.

The output can be either combinational or sequential. Entries in the state table can be either level sensitive or edge sensitive. For sequential UDPs, the output must be defined as a reg, and each table entry includes the previous value of the output (as well as the new value of the output).

Here are some of the conventions used to define state tables:

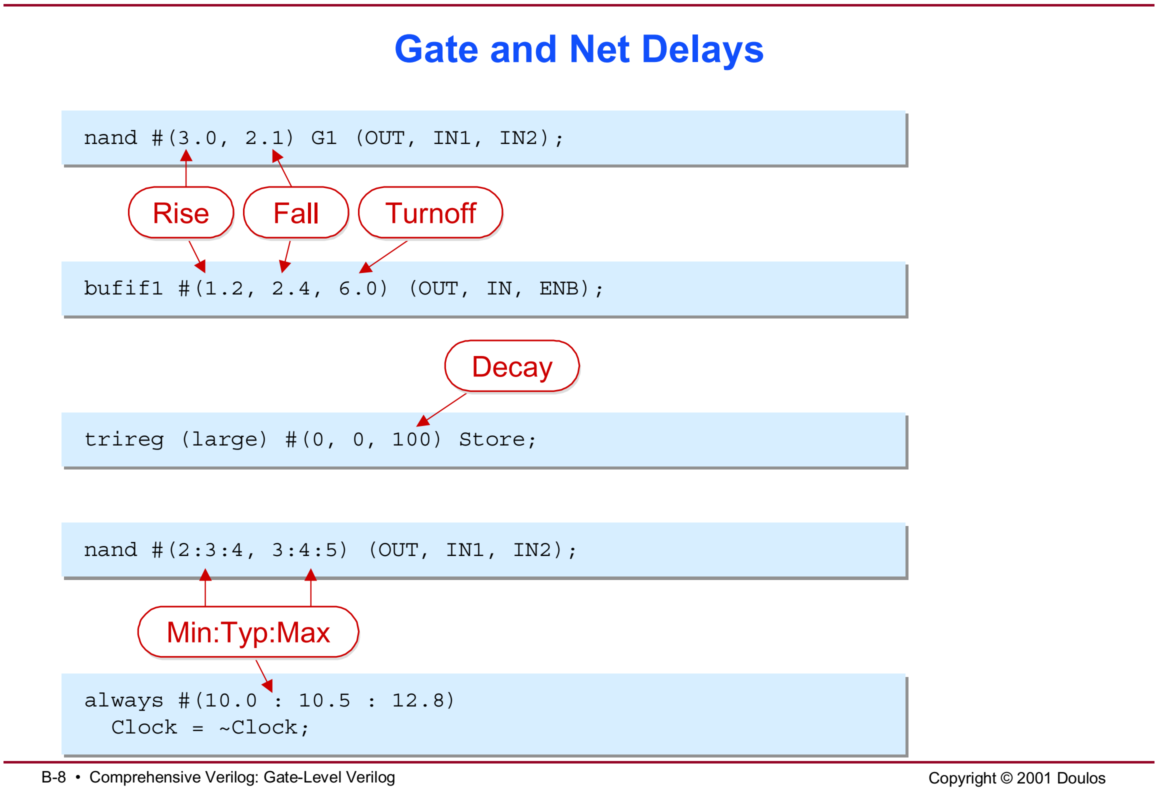
- 0 Logic0
- 1 Logic1
- x Equivalent to the Verilog unknown value 'bx
- ? Matches 0, 1 or x, not allowed in the output field
- - No change, only allowed in the output field
- (01) A transition from 0 to 1, not allowed in the output field
- (??) A transition from any value to any other value.

For any input conditions not included in the state table, the output will be set to 1'bx.

Level sensitive table entries (i.e. rows not including a transition) take precedence over edge sensitive entries (i.e. rows that do include a transition.)

There is one important rule to understand when writing UDPs; if you specify the output for one input transition, you must specify the output value for every possible input transition. Otherwise, the output will be set to 1'bx unexpectedly!

Gate and Net Delays



Gate and Net Delays

```
nand #(3.0, 2.1) G1 (OUT, IN1, IN2);
```

```
bufif1 #(1.2, 2.4, 6.0) (OUT, IN, ENB);
```

```
trireg (large) #(0, 0, 100) Store;
```

```
nand #(2:3:4, 3:4:5) (OUT, IN1, IN2);
```

```
always #(10.0 : 10.5 : 12.8)
  Clock = ~Clock;
```

B-8 • Comprehensive Verilog: Gate-Level Verilog

Notes:

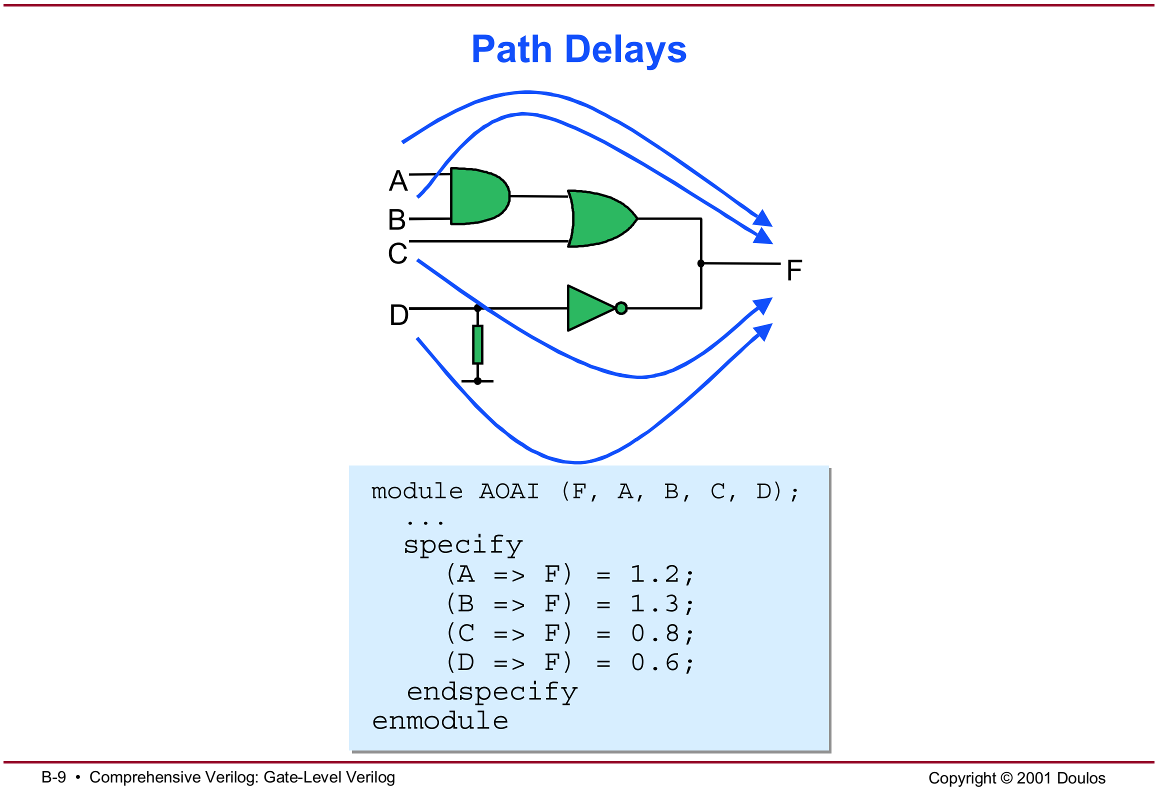
Each Verilog primitive, continuous assignment and net can be assigned a delay. Each Verilog event is delayed by the given amount as it passes through the primitive or along the net. Gate and assign delays model propagation delays; net delays model track delays.

Separate rise and fall delays can be specified. A rise delay simply means the propagation delay (through the primitive or whatever) of an event where the output is changing to 'b1. A rise delay is a propagation delay for a change to 1, not the switching time of the output or the time for the output to cross some voltage threshold.

For nets that can go high impedance, a turnoff or charge decay delay can be specified.

Minimum, typical and maximum values can be given wherever you can write a delay, including procedural delays (i.e. delays within initial and always blocks).

Path Delays



Path Delays

```
module AOAI (F, A, B, C, D);
   ...
  specify
    (A => F) = 1.2;
    (B => F) = 1.3;
    (C => F) = 0.8;
    (D => F) = 0.6;
  endspecify
enmodule
```

B-9 • Comprehensive Verilog: Gate-Level Verilog

Copyright © 2001 Doulos

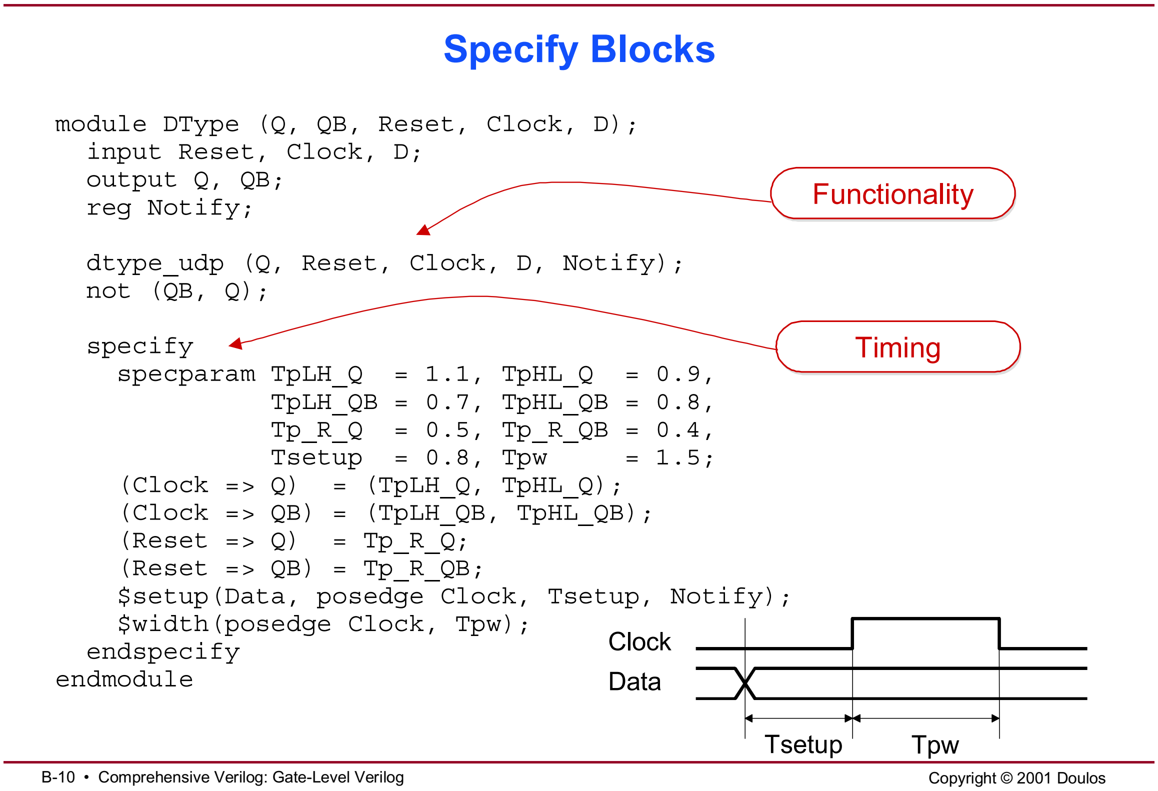
Notes:

Verilog has two methods of specifying delays: distributed delays, and path delays.

With distributed delays, delays are assigned to individual primitives within a network. The total delay from a module input to a module output is the sum of the delays of the individual primitives on the path. For an example, see module AOAI at the beginning of this section.

With path delays, delays are defined directly between module inputs and module outputs, bypassing the network in between. This method is more flexible, since two paths through the same primitives can have different delays, e.g. the paths from A to F and from B to F opposite.

Specify Blocks



Specify Blocks

```
module DType (Q, QB, Reset, Clock, D);
  input Reset, Clock, D;
  output Q, QB;
  reg Notify;

  dtype_udp (Q, Reset, Clock, D, Notify);
  not (QB, Q);

  specify
    specparam TpLH_Q  = 1.1, TpHL_Q  = 0.9,
              TpLH_QB = 0.7, TpHL_QB = 0.8,
              Tp_R_Q  = 0.5, Tp_R_QB = 0.4,
              Tsetup  = 0.8, Tpw     = 1.5;
    (Clock => Q)  = (TpLH_Q, TpHL_Q);
    (Clock => QB) = (TpLH_QB, TpHL_QB);
    (Reset => Q)  = Tp_R_Q;
    (Reset => QB) = Tp_R_QB;
    $setup(Data, posedge Clock, Tsetup, Notify);
    $width(posedge Clock, Tpw);
  endspecify
endmodule
```

B-10 • Comprehensive Verilog: Gate-Level Verilog

Notes:

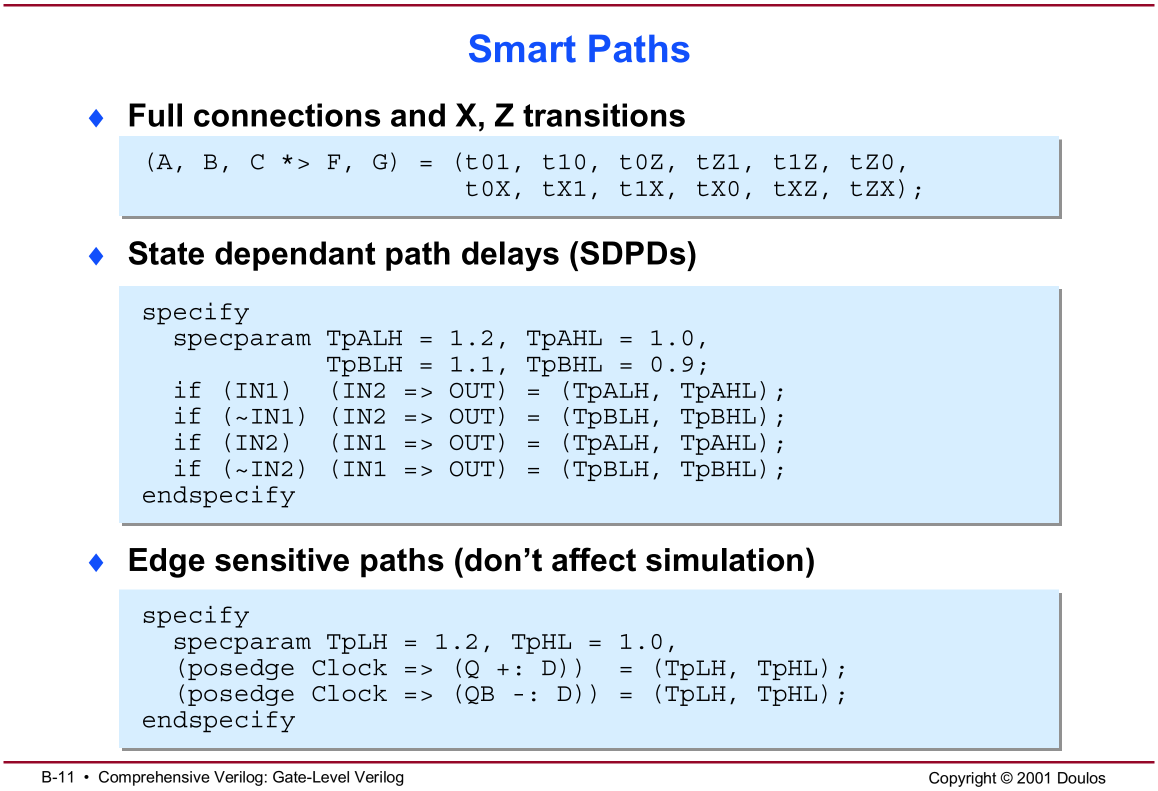
Specify blocks are the most accurate and powerful way to specify timing in Verilog. They allow the specification of path delays, timing constraints, independent delays for X and Z transitions, state dependent paths and pulse rejection.

The philosophy behind specify blocks is to separate the description of timing from the description of functionality. This makes is easier to code up timing information directly from the information given on data sheets. It also means that tools for timing verification and synthesis can make use of the Verilog modules.

Specparams allow named constants to be defined inside a specify block, which makes the code easier to read. The values of specparams can be overwritten during back-annotation.

Timing checks are performed using system tasks that can only be called within a specify block, e.g. $setup and $width check the setup time and minimum pulse width respectively.

Smart Paths



Smart Paths

- Full connections and X, Z transitions

```
(A, B, C *> F, G) = (t01, t10, t0Z, tZ1, t1Z, tZ0,
                     t0X, tX1, t1X, tX0, tXZ, tZX);
```

- State dependant path delays (SDPDs)

```
specify
  specparam TpALH = 1.2, TpAHL = 1.0,
            TpBLH = 1.1, TpBHL = 0.9;
  if (IN1)  (IN2 => OUT) = (TpALH, TpAHL);
  if (~IN1) (IN2 => OUT) = (TpBLH, TpBHL);
  if (IN2)  (IN1 => OUT) = (TpALH, TpAHL);
  if (~IN2) (IN1 => OUT) = (TpBLH, TpBHL);
endspecify
```

- Edge sensitive paths (don’t affect simulation)

```
specify
  specparam TpLH = 1.2, TpHL = 1.0,
  (posedge Clock => (Q +: D))  = (TpLH, TpHL);
  (posedge Clock => (QB -: D)) = (TpLH, TpHL);
endspecify
```

B-11 • Comprehensive Verilog: Gate-Level Verilog

Notes:

Specify blocks support some very powerful and sophisticated methods of specifying timing.

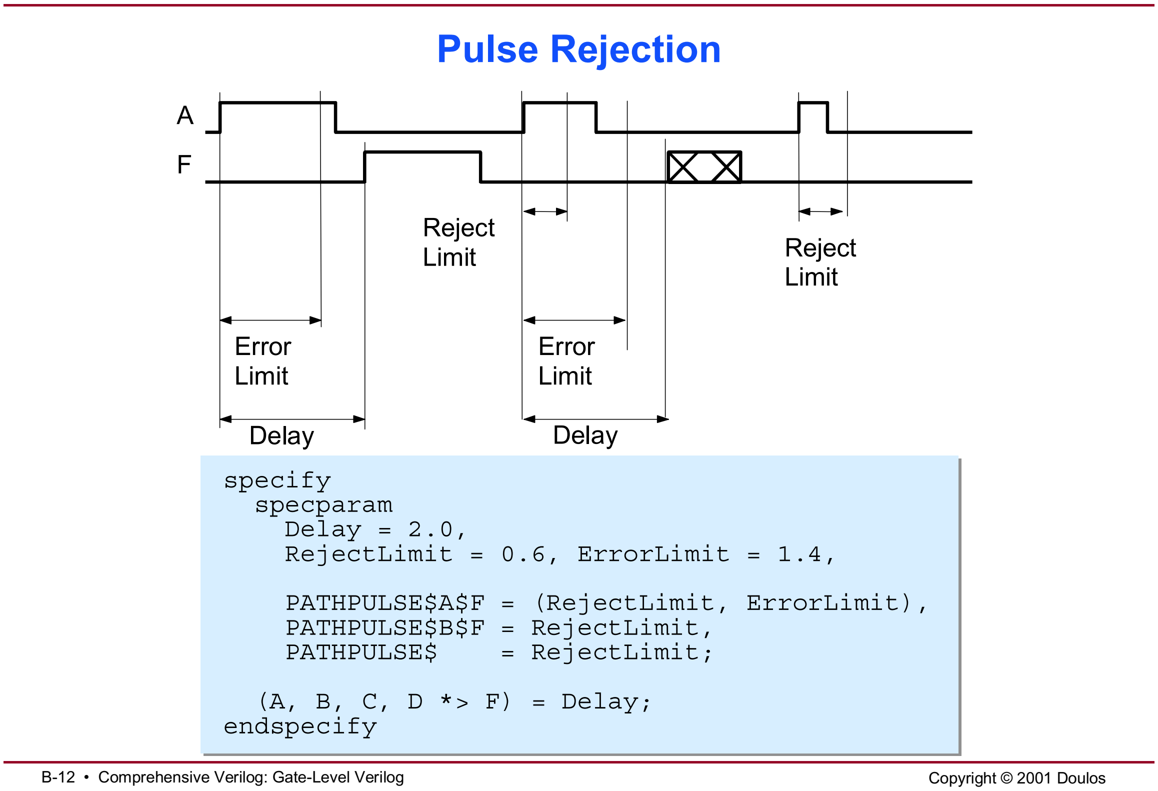
Independent delays can be specified for all possible transitions between the values 0, 1, Z and X. This can be a list of 2 values (transitions to 0 and 1), 3 values (transitions to 0, 1 and Z), 6 values (transitions between 0, 1 and Z) or 12 values (transitions between 0, 1, Z and X). An example of this latter case is given opposite. It is important to use specparams to give meaningful names to the values!

The *> symbol in the path specification indicates a full connection, i.e. every possible path from the inputs on the left to the outputs on the right (3x2=6 paths in this example). The alternative is the symbol => which represents a parallel connection i.e. the input and output must both be vectors of the same length, and the number of paths equals the number of bits in each vector.

State dependent path delays allow a path to be made conditional on the state of another input. Thus, different delays can be specified according to the value on another input.

Edge sensitive path delays include functional as well as timing information. The edge sensitive path delays shown opposite indicate that there are delays from the rising edge of Clock to Q and to QB; also, the value of Q is derived from the value of D with no inversion, and the value of QB is derived from D with inversion. Edge sensitive path information has no affect on simulation, but the information can be used for Timing Verification.

Pulse Rejection



Pulse Rejection

```
specify
  specparam
    Delay = 2.0,
    RejectLimit = 0.6, ErrorLimit = 1.4,

    PATHPULSE$A$F = (RejectLimit, ErrorLimit),
    PATHPULSE$B$F = RejectLimit,
    PATHPULSE$    = RejectLimit;

  (A, B, C, D *> F) = Delay;
endspecify
```

B-12 • Comprehensive Verilog: Gate-Level Verilog

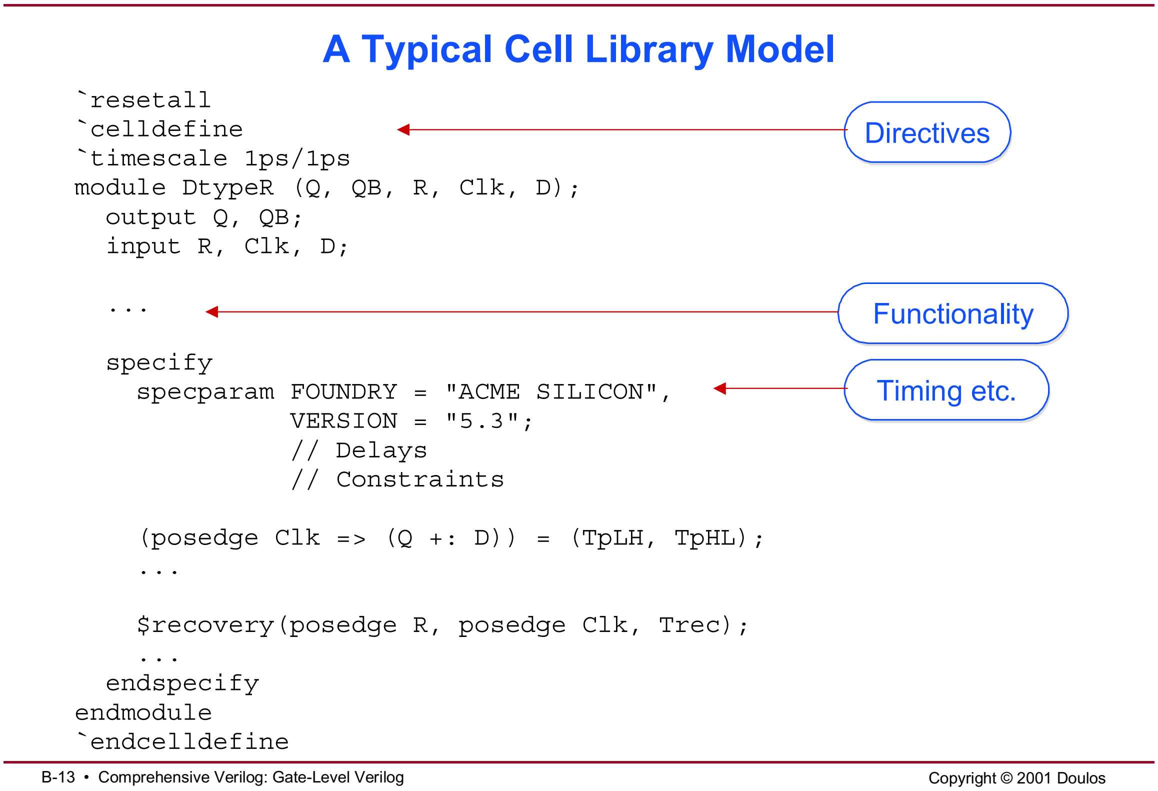
Notes:

Delays on Verilog primitives and specify blocks are inertial, i.e. pulses less than the delay get filtered out. This pulse rejection is programmable, and can be altered via the specify block.

To change the pulse rejection period, you must use a specparam with the conventionally defined name PATHPULSE$. The value of the specparam can be either a single reject limit, or a list of two values, a reject limit and an error limit. Pulses shorter than the reject limit are filtered out. Pulses longer than the reject limit but shorter than the error limit are transmitted as X pulses. Pulses longer than the error limit are transmitted as solid pulses, even if they are shorter than the path delay.

Pulse rejection can be defined for individual paths by identifying the path within the name of the specparam. E.g. the specparam PATHPULSE$A$F defines pulse rejection for the path from input port A to output port F.

A Typical Cell Library Model



A Typical Cell Library Model

```
`resetall
`celldefine
`timescale 1ps/1ps
module DtypeR (Q, QB, R, Clk, D);
  output Q, QB;
  input R, Clk, D;

  ...

  specify
    specparam FOUNDRY = "ACME SILICON",
              VERSION = "5.3";
              // Delays
              // Constraints

    (posedge Clk => (Q +: D)) = (TpLH, TpHL);
    ...

    $recovery(posedge R, posedge Clk, Trec);
    ...
  endspecify
endmodule
`endcelldefine
```

Directives

Functionality

Timing etc.

B-13 • Comprehensive Verilog: Gate-Level Verilog

Copyright © 2001 Doulos

Notes:

Here is the skeleton code of a typical Verilog cell library model.

Notice that the module is enclosed in the compiler directives `celldefine...`endcelldefine. These tell the PLI that the module is a cell for back annotation purposes. The `resetall directive cancels any previous directives so that the compiler is in a known state.

Internally, the module is divided into two parts - the functionality and the timing. The timing is divided into three parts - parameters, path delays and timing checks. It is common to include specparams giving general information about the model such as the version number.

Part Number: 069287