

Digital IC Design Fundamentals

Course Version 2.0

Lecture Manual

Revision 1.0

© 1990-2023 Cadence Design Systems, Inc. All rights reserved.

Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. (Cadence) contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence trademarks, contact the corporate legal department at the address shown above or call 1-800-862-4522.

All other trademarks are the property of their respective holders.

Restricted Print Permission: This publication is protected by copyright and any unauthorized use of this publication may violate copyright, trademark, and other laws. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. This statement grants you permission to print one (1) hard copy of this publication subject to the following conditions:

The publication may be used solely for personal, informational, and noncommercial purposes;

The publication may not be modified in any way;

Any copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement; and

Cadence reserves the right to revoke this authorization at any time, and any such use shall be discontinued immediately upon written notice from Cadence.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. The information contained herein is the proprietary and confidential information of Cadence or its licensors, and is supplied subject to, and may be used only by Cadence customers in accordance with, a written agreement between Cadence and the customer.

Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Table of Contents

Digital IC Design Fundamentals

Module 1	About This Course	2
Module 2	Digital IC Design Flow – Overview and Challenges	8
	Subtopic: The Scale of the Challenge	84
Module 3	Digital IC Functional Design.....	117
	Subtopic: Digital IC Functional Design: Hardware Implementation	130
	Subtopic: Data Representation	132
	Subtopic: Combinatorial Building Blocks	144
	Subtopic: Clocked Building Blocks and Synchronous Design	156
	Subtopic: Arithmetic Building Blocks	178
	Subtopic: Finite State Machines	187
	Subtopic: Memory Structures	204
	Subtopic: Functional Design Challenges	219
	Subtopic: X- Propagation	225
	Subtopic: Clock Domain Crossing (CDC)	233
	Subtopic: Reset Domain Crossing (RDC)	246
	Subtopic: Clock Gating	251
	Subtopic: Low-Power Concepts	261
	Subtopic: Introduction to Low-Power Simulation	266
Module 4	SystemVerilog Fundamentals for Design	280
	Subtopic: Design Modules	283
	Subtopic: Standard SystemVerilog Types	298
	Subtopic: Making Procedural Statements	320
	Subtopic: Using Operators	338
	Subtopic: Using Blocking and Nonblocking Assignments	356
	Subtopic: User-Defined Data Types and Structures	372
	Subtopic: Packages	385
	Subtopic: Coding RTL for Synthesis	396
	Subtopic: Designing Finite State Machines	413
Lab 1	Modeling a Data Driver	
Lab 2	Modeling a Simple Multiplexor	
Lab 3	Modeling an Arithmetic Logic Unit (ALU)	
Lab 4	Modeling a Simple Register	

- Lab 5 Remodeling the ALU
- Lab 6 Modeling a Simple Counter
- Lab 7 Modeling a Sequence Controller

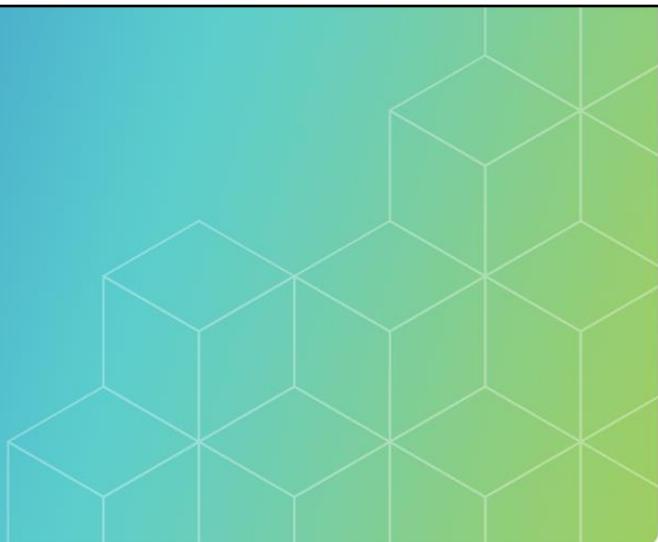
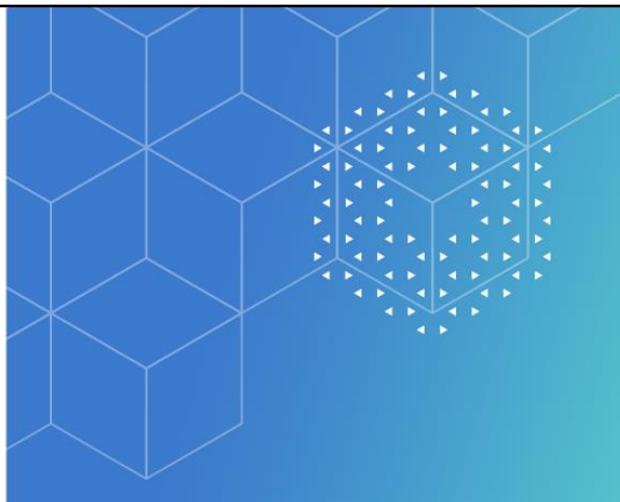
Module 5	Functional Verification.....	425
Subtopic:	Debug Techniques	430
Subtopic:	Debugging Using Interactive Debug with Xcelium: GUI and Textual/Batch Commands	438
Subtopic:	Introduction to a Modern Verification Flow	460
Subtopic:	Introduction to MDV and Planning	487
Subtopic:	Overview of Formal Analysis Use Models	514
Subtopic:	The Verification Completeness Problem	561
Module 6	Digital IC Design Methodology	583
Module 7	IC Packaging.....	721
Module 8	Next Steps.....	788

Digital IC Design Fundamentals

Version	2.0
Revision	1.0
Estimated time:	18 hours

cadence®

Welcome to the Digital IC Design Fundamentals class. This spans about 18 hours of lecture.



Module 1

About This Course

cadence®

This short About The Course module provides information on the prerequisites required for taking this course and includes an agenda.

It also includes information on the Badge Exam associated with this course and the software licenses required.

Course Prerequisites

Before taking this course, You must have experience and knowledge of the following:

- Basic Programming

You must have completed the following:

- Semiconductor 101



This page does not contain notes.

Course Objectives

In this course, you will:

- Draw a flow diagram of the entire design flow and explore the entire ASIC design flow process.
- Identify the distinction between Digital IC design, verification, and implementation.
- Recognize the different stages of front-end design and verification.
- Demonstrate the SystemVerilog HDL for design and verification.
- Recognize the different stages of design implementation.
- Create, verify and implement a system-level design with a simple architecture.
- Identify the challenges of scaling, costs, physical attributes, as well as low power and area constraints before tapeout.
- Identify the different processes in the semiconductor industry used to handle the above realistic challenges.



This page does not contain notes.

Course Agenda

Module 1

- About This Course

Module 2

- Digital IC Design Flow – Overview and Challenges
 - The Scale of the Challenge

Module 3

- Digital IC Functional Design
 - Digital IC Functional Design: Hardware Implementation
 - Data Representation
 - Combinatorial Building Blocks
 - Clocked Building Blocks and Synchronous Design
 - Arithmetic Building Blocks
 - Finite State Machines
 - Memory Structures
 - Functional Design Challenges
 - X- Propagation
 - Clock Domain Crossing (CDC)
 - Reset Domain Crossing (RDC)
 - Clock Gating
 - Low-Power Concepts
 - Introduction to Low-Power Simulation

Module 4

- SystemVerilog Fundamentals for Design
 - Design Modules
 - Standard SystemVerilog Types
 - Making Procedural Statements
 - Using Operators
 - Using Blocking and Nonblocking Assignments
 - User-Defined Data Types and Structures
 - Packages
 - Coding RTL for Synthesis
 - Designing Finite State Machines

Module 5

- Functional Verification
 - Debug Techniques
 - Debugging Using Interactive Debug with Xcelium: GUI and Textual/Batch Commands
 - Introduction to a Modern Verification Flow
 - Introduction to MDV and Planning
 - Overview of Formal Analysis Use Models
 - The Verification Completeness Problem

Module 6

- Digital IC Design Methodology

Module 7

- IC Packaging

5 © Cadence Design Systems, Inc. All rights reserved.



Please take a few minutes to go through the agenda presented in this slide of all the modules covered.

Become Cadence Certified by Earning a Digital Badge



Digital badges indicate mastery in a certain Micro-Credential and give managers and potential employers a way to validate your expertise.

Benefits of Cadence Certified Digital Badges

- Validate expertise
 - Expand career opportunities
- Professional credibility
 - Stand apart from your peers
- Your digital badge can be added to your email signature or social media platforms like LinkedIn or Facebook.



How do I register to take the exam?

- Log in to our [Learning Management System](#) to locate the exam in your transcript.

How long will it take to complete the exam?

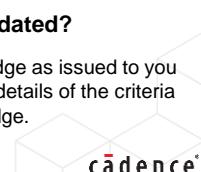
- Most exams take 45 to 90 minutes to complete. You may retake the exam multiple times to pass the exam.

How do I access and use the digital badge?

- After you pass the exam, you get a digital badge and instructions on how to place it on social media sites.

How is the digital badge validated?

- [Credly](#) validates the digital badge as issued to you by Cadence and includes the details of the criteria you completed to earn the badge.



This page does not contain notes.

Icons Used in This Class



Best Practice



Language/
Command Syntax



Concept/
Glossary



Frequently Asked
Questions/
Quiz



Error Message



Problem & Solution



Quick Reference



GUI and Command



How To

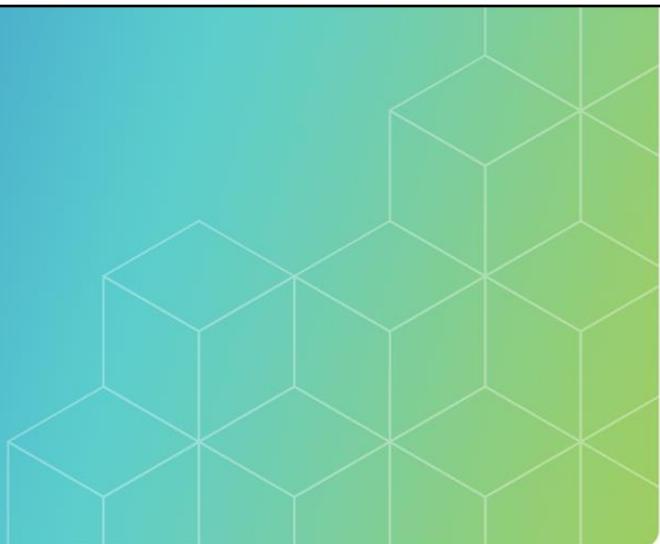
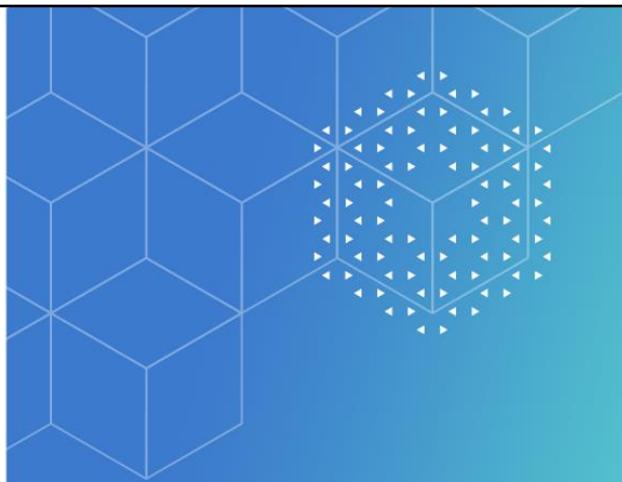


Lab List

Throughout this class, we use icons to draw your attention to certain kinds of information. Here are the icons we use, and what they mean.



This page does not contain notes.



Module 2

Digital IC Design Flow – Overview and Challenges

cadence®

This module spans 3 hours of lecture time.

Module Objectives

In this module, you will:

- Draw a flow diagram of the entire design flow.
- Identify the distinction between Digital IC design, verification, and implementation.
- Recognize the different stages of front-end design and verification.
- Recognize the different stages of design implementation.
- Identify the challenges of scaling, costs, physical attributes, as well as low power and area constraints before tapeout.
- Identify the different processes in the semiconductor industry used to handle the above realistic challenges.



In this module, you will:

- Draw a flow diagram of the entire design flow.
- Identify the distinction between Digital IC design, verification, and implementation.
- Recognize the different stages of front-end design and verification.
- Recognize the different stages of design implementation.
- Identify the challenges of scaling, costs, and physical attributes, as well as low power and area constraints before tapeout.
- Identify the different processes in the semiconductor industry used to handle the above realistic challenges.

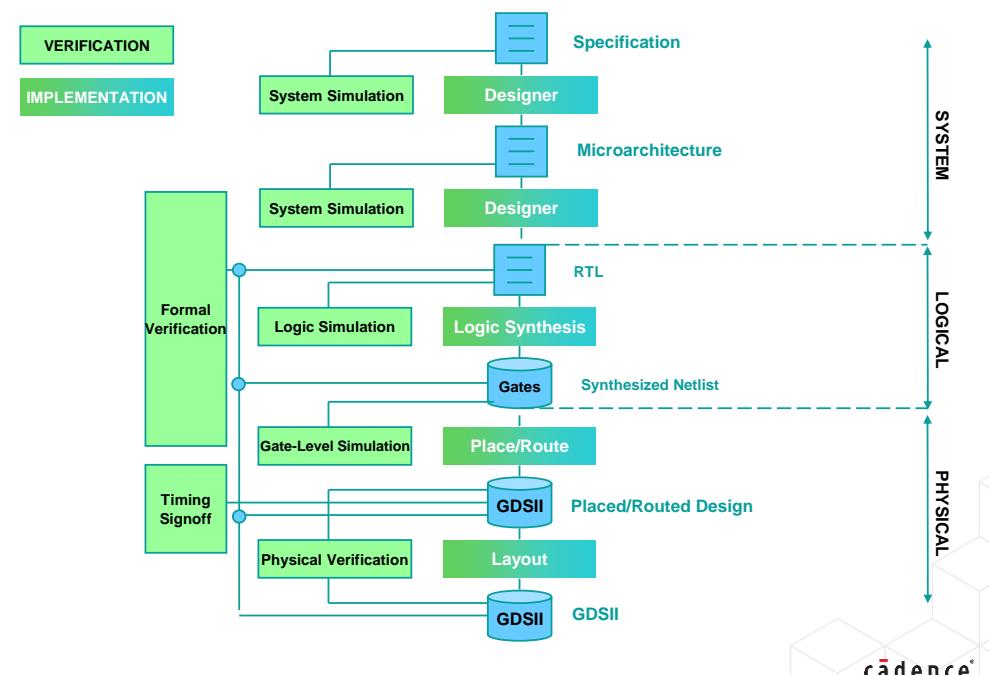
Complete Basic IC Design Flow

A design flow can be divided into three phases:

- System
- Logical
- Physical

In each phase, two main processes need to be performed:

- Implementation
- Verification



10 © Cadence Design Systems, Inc. All rights reserved.

Let us discuss briefly the three phases of design, the system, the logical, and the physical.

In the system phase, you define the requirements and develop the specifications for your design. You then develop the micro-architecture and run simulation to test your system for various parameters to deduce the expected power, area, and timing. When that phase is satisfactory, you move on to the logical phase.

In the logical phase, you develop the (Register Transfer Logic) RTL. Intellectual property (IP) reuse is common in this step. The RTL must then be simulated to verify the desired functionality.

The RTL must then be synthesized to gates from the technology of the foundry you desire to target.

The gates are transferred in the form of a netlist into the place and route tool, where you run floorplanning, placement and routing.

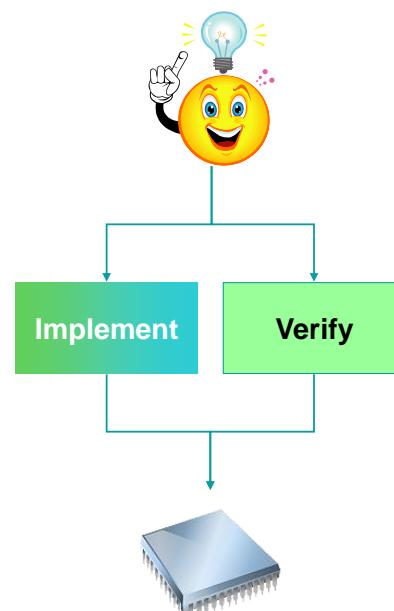
After one more round of verification at the physical level, and after meeting all your signoff requirements, you then proceed to generate a, G D S 2, so as to create a mask for the foundry.

Also remember that in each of these phases of the, system, logical and physical, we just discussed, there are 2 processes which run all the time, one is Verification and the other is Implementation.

Two Main Processes of the IC Design Flow

At the highest level, what tasks need to be performed when creating a chip from an idea?

- Implementation
 - Transform the design from idea or specification to various representations of logical and physical hardware.
- Verification
 - Ensure the functionality, timing, and integrity of the changing design through the process.



11 © Cadence Design Systems, Inc. All rights reserved.



The current day digital systems such as 5G networks, related smartphones, compute farms, cloud computing, edge computing, artificial intelligence (AI) machines, autonomous cars etc. all started with a specification.

Each system must be thoroughly verified to provide the correct functionality, but also to achieve the proper balance of power and performance.

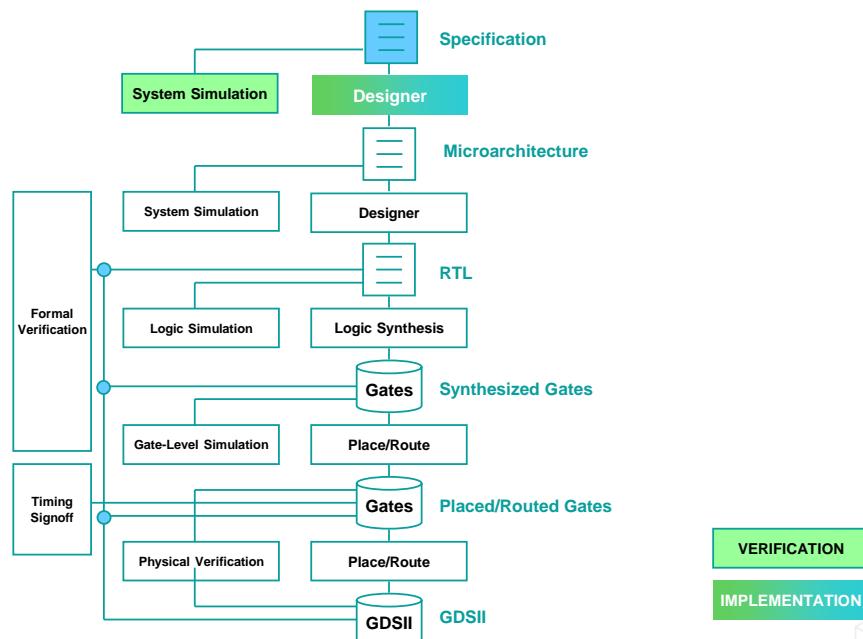
Many trade-offs are made along the way to achieve an optimal result.

So, At the highest level, what tasks need to be performed when creating a chip from an idea?

Implementation, that is, to transform the design from idea or specification to various representations of logical and physical hardware.

Verification, which is, to ensure the functionality, timing, and integrity of the changing design through the process.

It Starts with a Specification

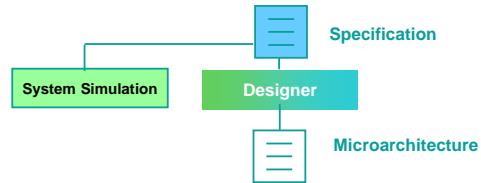


12 © Cadence Design Systems, Inc. All rights reserved.

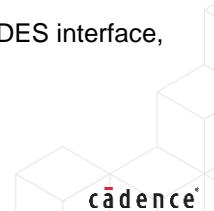
What Is a Specification?

A specification is an explicit set of requirements to be satisfied by a material, product, or service.

- Ideas begin with a specification, which can be a textual, graphical, or even a software representation.
- For chip design, the specification is the reference model the team uses to:
 - Design the overall chip.
 - Specify the intellectual property used.
 - Specify the “new logic” to be created.
 - Specify the block-level and chip-level interfaces.
 - Partition the chip into functional blocks.
 - Communicate interfaces and requirements with other teams.
 - Measure actual performance versus specified targets.
- **Example:** The specification for the latest chip called for a 250-MHz core clock with a 6G SERDES interface, able to process 1M streams of data per second at less than 10W total power.



13 © Cadence Design Systems, Inc. All rights reserved.



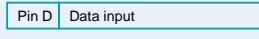
Let us understand what a Design Specification is. It is a set of explicit requirements to be satisfied by a material, a product or a service.

It can conceptualize as an idea which can be textual, graphical or even a software representation.

For chip design, the specification is the reference model the team uses to:

- Design the overall chip.
- Specify the intellectual property used.
- Specify the “new logic” to be created.
- Specify the block-level and chip-level interfaces.
- Partition the chip into functional blocks.
- Communicate interfaces and requirements with other teams.
- Measure actual performance versus specified targets.

Example: Specification

Title	Specification for XYZ Design
List of Reviewers	HW, SW, Test, Manufacturing, Customer, etc.
Modification History	1.0 – 12/2007 – Initial Revision, 2.0 – 1/2008 – Changes to ...
Table of Contents	Section 1 Overview – Section 2 Block 1 ...
Glossary	XYZ = Codename for project, etc.
Overview	The XYZ chip is a new product targeted for consumers...
Performance Targets	125 MHz, 1W Total power, 1M streams/sec
Block Diagrams	
Graphs	
Tables	
Detailed Description	Block A is input block, it receives signals from the I/O...

14 © Cadence Design Systems, Inc. All rights reserved.



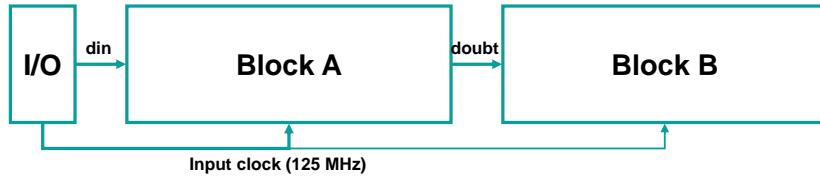
This slide shows a table containing the items that should be part of a specification document.

First is the title of the specification, in this case, the X Y Z Design, then the list of reviewers , or the stakeholders we can say, The Modification history if at all it has been modified in the past, the table of contents of the document, the Glossary, the short intro or overview of the design, The Performance target parameters like the frequency, power and speed. The design block diagrams, then graphs if any to depict the functionality, tables of values if any, and then the detailed description of the entire design.

Specification Snippet Example

Block A Interface

Block A is the input block, which receives serial data from the I/O and transfers to Block B.



Port Name	Direction	Source/Destination	Size	Description
clk	input	I/O	1 bit	Clock at 800 MHz
din	input	I/O	32 bits	Input data
dout	output	A and B	32 bits	Output data

15 © Cadence Design Systems, Inc. All rights reserved.



This slide shows the way a block is described at a higher level in the specification. An example of Block A is described here. A short intro of Block A, followed by the block diagram and the table of important values associated with the Block.

In this example, it says Block A is the input block, which receives serial data from the I/O and transfers it to Block B. The input clock frequency is 125 MHz.

High-Level Decisions

In creating or modifying the specification, high-level decisions *that affect the system and its environment* are made.

- For example, the choice between external SRAM or DRAM:
 - Control for each one is drastically different.
 - I/Os for the chip will be affected.
 - Board itself will be affected, components plus signal routing.
- Another example is the choice to run the design at 250 MHz or 500 MHz:
 - Chip-level clock input has to change.
 - Software might have to change because the performance could be 2X different.



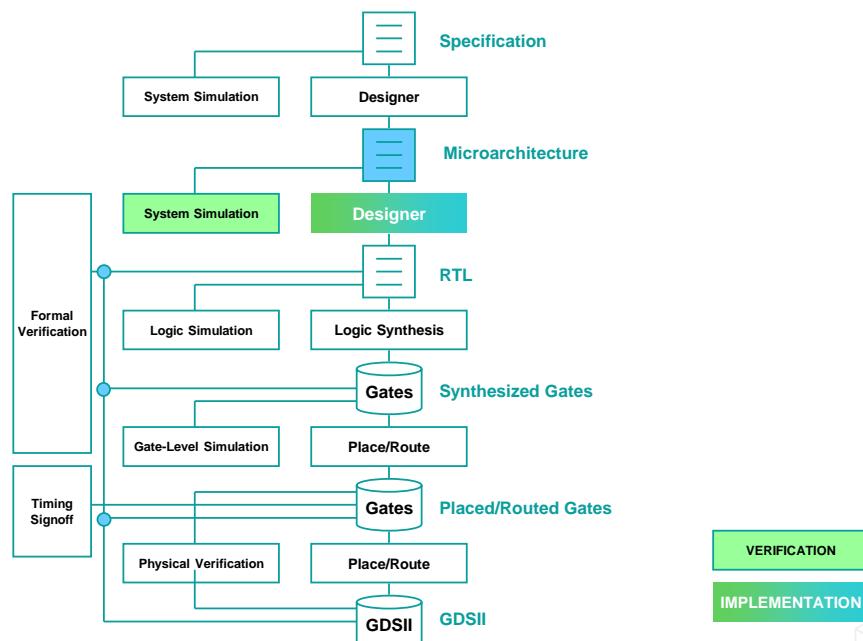
Higher-level decisions about creating a specification are made by the Architect and a team of Stakeholders together. These are the kind of decisions that affect and decide the system and its environment.

For example, the choice between external SRAM and DRAM. Choosing any of these will have a different control for each and will be drastically different from the other.

Further, the I/Os of the chip will be affected, and the board itself will be affected, as well as the components and the signal routing too.

Another example is the choice to run the design either at 250 MHz or 500 MHz. For this, the chip-level clock input has to change. Also, the software might have to change because the performance could be 2X different.

Microarchitecture



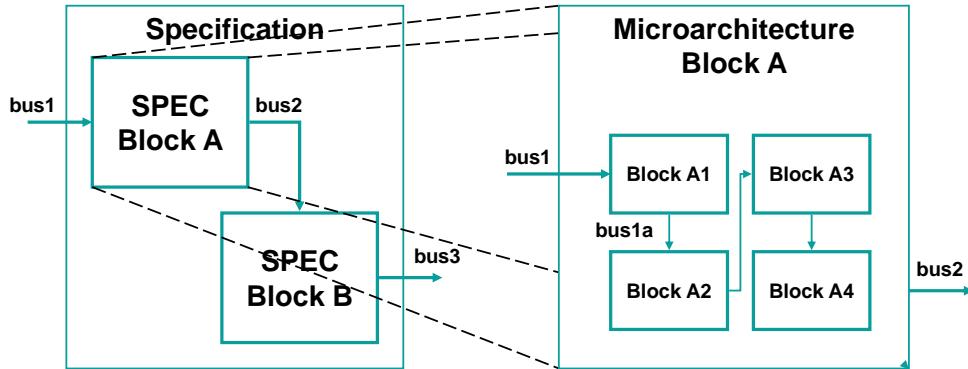
17 © Cadence Design Systems, Inc. All rights reserved.



The next step in the design flow is converting the high-level design specification to provide more details for the individual design components and blocks and defining the inter-communication amongst them. This is termed as the microarchitecture.

Example: Microarchitecture

The microarchitecture is typically based on a block in the specification.



18 © Cadence Design Systems, Inc. All rights reserved.



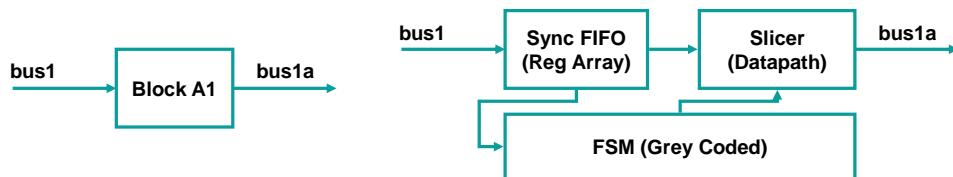
The microarchitecture is typically based on individual blocks in the specification. As seen in the diagram on this slide, the design specification defines Block A and Block B in the spec.

The microarchitecture for Block A, then defines the inner block A1, A2, A3 and A4 and the intercommunication between them. Also, it provides the protocol of Bus1 and Bus2 connecting Block A to other components or parts in the design.

Example: Microarchitecture (continued)

Section 2.1.1 Block A1

This is the input block for Block A. Its function is to process and slice the data into 16-bit segments, based on control signals from the FSM, and send it to block B2.



Port Name	Direction	Source/Destination	Size	Description
clk	input	I/O	1 bit	Clock at 125 MHz
bus1	input	I/O and Sync FIFO	32 bits	Input data
bus1a	output	Slicer	16 bits	Output data

19 © Cadence Design Systems, Inc. All rights reserved.



This slide shows the microarchitecture description snippet of Block A inside the main specification. It describes its input signals and its functionality. It explains that Block A's function is to process and slice the data into 16-bit segments, based on control signals from the F S M and send it to Block B2.

This information is depicted as a block diagram, as seen.

There is a table of information provided with values that must be used for running this block, like, the clock frequency of the buses, the signal directions and the size of the signals, etc.

Mid-Level Decisions

In creating or modifying the microarchitecture, mid-level decisions that affect the block itself are made.

- For example, the choice between internal SRAM or register array:
 - Interface to the outside environment is the same.
 - Performance of the block may vary slightly, but functionality is the same.
- The choice to use multiple datapaths versus a single datapath:
 - Area is a tradeoff versus performance.
 - As long as the performance targets are met, how the design is actually implemented is a microarchitectural decision.



As the high-level decisions are made at the Design Spec level, there are mid-level decisions that are made at the Microarchitecture level. These are the decisions for creating or modifying the microarchitecture at the block level.

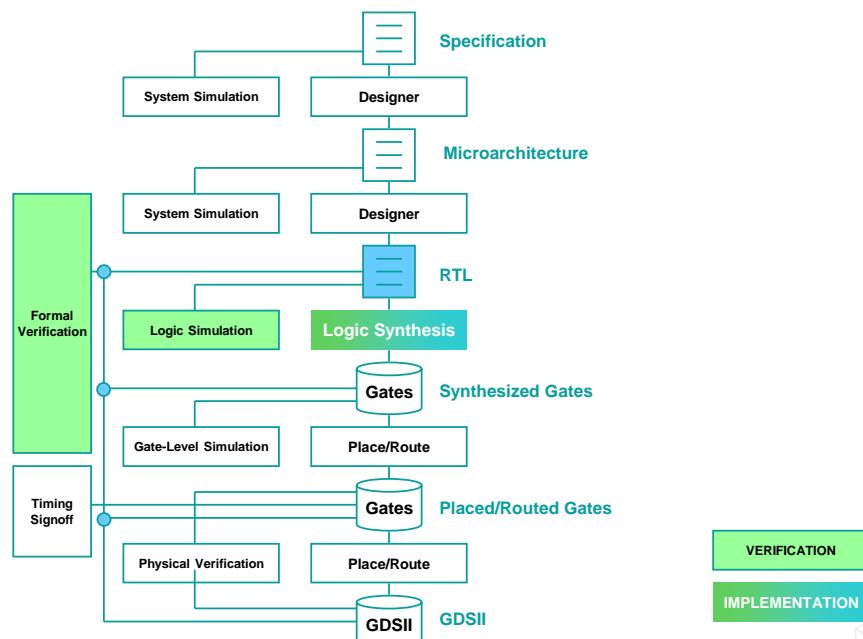
For example, to choose between SRAM and a register array. For each of these choices, the interface to the outside world might be the same, but the performance of the block may vary slightly, even though the functionality remains the same.

Another example of a decision would be to use multiple datapaths versus a single datapath.

In this case, there is a tradeoff between area and performance.

As long as the performance targets are met, how the design is actually implemented is a microarchitectural decision.

Register Transfer Level (RTL) Phase



21 © Cadence Design Systems, Inc. All rights reserved.

After the main Design Spec is written and also the microarchitecture block-level details are finetuned, then the designer converts his or her design blocks into behavioral modeling or register transfer level modeling through the high-level design language or HDL such as Verilog and SystemVerilog. And high-level verification language or HVL, such as Verilog and SystemVerilog, are used for creating the verification testbench and its related infrastructure.

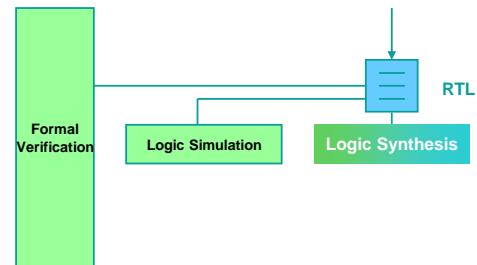
The design code created is “tested” using the testbench written. This phase is called the logic simulation.

What Is RTL?

A way of describing the operation of a digital circuit where the behavior is defined in terms of the flow of signals between registers and the operations performed.

For chip design, the RTL is the reference model the designer uses to:

- Design the block for final implementation.
- Instantiate and connect intellectual property.
- Code the “new logic.”
- Create the block-level interfaces.
- Partition the block into sub-blocks.
- Verify the interfaces to other blocks.
- Run simulations to measure actual performance versus specified targets.
- The translation of a system specification to RTL is a difficult and time-consuming task.



22 © Cadence Design Systems, Inc. All rights reserved.



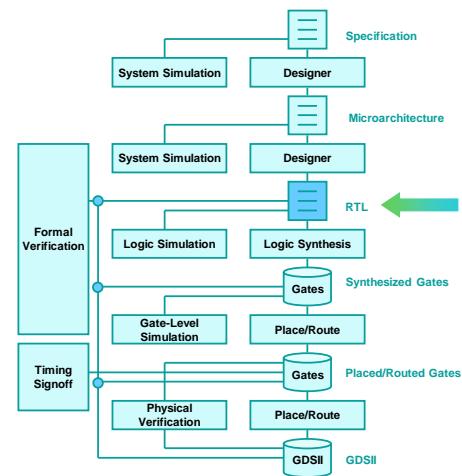
RTL stands for Register Transfer Level. It is a way of describing the operation of a digital circuit where the behavior is defined in terms of the flow of signals between registers and the operations performed.

For chip design, the RTL is the reference model the designer uses to:

- Design the block for final implementation.
- Instantiate and connect intellectual property.
- Code the “new logic.”
- Create the block-level interfaces.
- Partition the block into sub-blocks.
- Verify the interfaces to other blocks.
- Run simulations to measure actual performance versus specified targets.

Importance of RTL in the Flow

- RTL is the representation that bridges the specification and microarchitecture to the final implementation.
- Most RTL is manually created by skilled and experienced design engineers, using the specification and microarchitecture.
- The quality of the RTL directly affects the end product:
 - Time
 - Effort
 - Money



23 © Cadence Design Systems, Inc. All rights reserved.



RTL, as described in the previous slides, is the format in which the design blocks are coded. It is the representation that bridges the specification and microarchitecture to the final implementation.

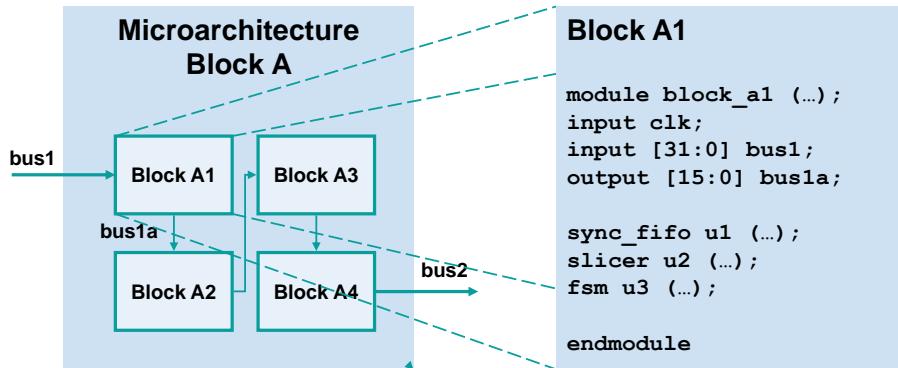
Most RTL is manually created by skilled and experienced design engineers, using the specification and microarchitecture.

The quality of the RTL directly affects the end product in terms of:

- Time, effort and money.
- A good design coding will prove to be a great cause for achieving all these 3 factors.

Example: RTL

The RTL is typically based on a block in the microarchitecture.



24 © Cadence Design Systems, Inc. All rights reserved.



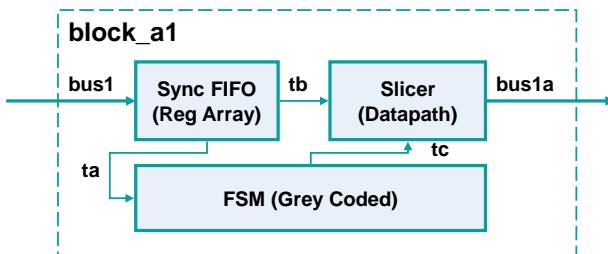
The RTL is typically based on a block in the microarchitecture.

Each block is coded separately and integrated using the designated HDL as decided by the project.

Also, the bus protocols and bus interfaces are also defined for intercommunication between the blocks.

As you can see in this slide, Block A1 is coded as a module, and its structure defined within it.

Example: RTL (continued)



```
// File : block_a1.v
module block_a1 (...);
  input clk;
  input [31:0] bus1;
  output [15:0] bus1a;
  sync_fifo u1 (...);
  slicer u2 (...);
  fsm u3 (...);
endmodule
```

```
// File : sync_fifo.v
module sync_fifo (clk, bus1, ta, tb);
...
endmodule

// File : slicer.v
module slicer (clk, tb, tc, bus1a);
...
endmodule

// File : fsm.v
module fsm (clk, ta, tc);
...
endmodule
```

25 © Cadence Design Systems, Inc. All rights reserved.



Further details for writing a Design Block is described in this slide.

Block A1 consists of 3 blocks, as you can see in the block diagram.

The Sync FIFO Register Array, the Slicer, which is the datapath, and the Finite State machine or FSM, which uses the grey coding style for coding the logic.

These 3 separate sub-blocks are created in 3 separate files, and the logic is defined in modules.

Then, the overall 4th file, with the Block A1 module, is created with the instantiation of the 3 sub-blocks within it.

This is how the block-level microarchitecture details are coded.

Low-Level Decisions

In creating or modifying the RTL, low-level decisions *that affect the implementation of the block itself* are made.

- For example, the choice to use a particular coding style:
 - Designer has previous knowledge of an optimal style for implementation or verification.
 - Designer is more comfortable with a particular style.
- The choice to add pipeline stages versus forcing more logic into a single cycle:
 - Cycle time is traded off for sequential area.
 - As long as the performance targets are met, how the design is actually implemented is a microarchitectural decision.
 - It is possible that the latency of the top-level block is “flexible.”



We already talked about high-level and mid-level decisions. As we go into the details of each block, we have to make more detailed low-level decisions.

These decisions will affect the implementation of the blocks.

Let's look at some examples.

The choice to use a particular coding style would be:

- Designer has previous knowledge of an optimal style for implementation or verification.
- Designer is more comfortable with a particular style.

The choice to add pipeline stages versus forcing more logic into a single cycle is based on factors like:

- Cycle time is traded off for sequential area.
- As long as the performance targets are met, how the design is actually implemented is a microarchitectural decision.
- It is possible that the latency of the top-level block is “flexible.”

Test Your Understanding

Specification/Microarchitecture/RTL

- Who creates the ___?
- What information is required to come up with the details of the ___?
- What other decisions would be considered ___?
- How would we validate these decisions?

	Specification	Microarchitecture	RTL
Who?	CEO, CTO, Marketing, Chip Lead, etc.	Chip Lead, Block-Level Designer	Block-Level Designer
What information?	Customers, Market Data, Competitive Data, etc.	Performance Targets, Block I/O, etc.	Performance Targets, Block I/O, etc.
Other decisions?	Overall architecture, Use of specific IP	Block Partitioning, Memory size and Quantity	Instantiate vs. Infer, Re-use code vs. create
Validate?	System Simulation	System Simulation	RTL Simulation

27 © Cadence Design Systems, Inc. All rights reserved.



There are more details to be captured once we start coding the design blocks.

From the Main Design specification to the architecture and then to the RTL level, we need to capture details as shown in the table.

At the main specification level, who is writing the specification? What information goes into this? What decisions need to be made at this level, and what kind of validation is required at this phase in the design cycle?

Then at the microarchitecture level, is the chip lead or the block-level designer who would write the specification? What information is covered in the blocks, like the performance targets, Block I/Os, etc.? What kind of decisions need to be made here, like block partitioning, memory size and quality, etc? And then what kind of validation is needed to test all these at this microarchitecture level, like the system-level simulation?

At the RTL level, who will code the design block? Is it the block-level designer? What information goes into the coding as in the block I/O description? Decisions like whether to instantiate or use inference, whether to re-use code, or create from scratch, need to be made at this level.

And the validation at this level would be the RTL simulation.

Design Challenges

Design

- Size and complexity

Multimillion gate designs, dividing up the design and delegating the work, and interfacing between the divided blocks.

- Power

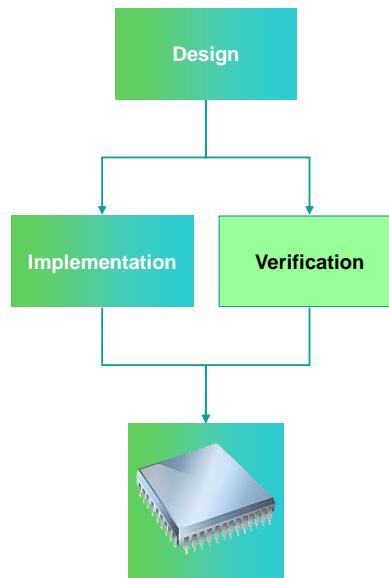
Power requirements have to be considered in the design process.

- Performance

High clock frequencies mean the designers must consider the technology information during the design process.

- IP

Choosing the right IP from the right source for the right application and technology.



28 © Cadence Design Systems, Inc. All rights reserved.

So, what are the Design Challenges one has to face?

- Size and complexity

Multimillion gate designs, dividing up the design and delegating the work, and interfacing between the divided blocks.

- Power

Power requirements have to be considered in the design process.

- Performance

High clock frequencies mean the designers must consider the technology information during the design process.

- IP

Choosing the right IP from the right source for the right application and technology.

Flow Example

Let's take a simple example through the flow phases we have seen so far.

We will cover each step and highlight the following:

- Definition and step in the overall flow
- Inputs and outputs
- Formats
- Example per step



Now let us take an example and go through the different phases that we discussed just now and then go on to the next phase in the design cycle, which is Implementation.

In the next few slides, we shall cover an example that shall explain very clearly the different phases we covered till now, namely, Design Specification, defining microarchitecture, coding RTL, and then running simulation.

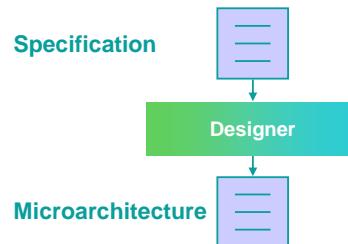
So, we shall essentially be covering and highlighting the following concepts:

- Definition and step in the overall flow
- Inputs and outputs
- Formats
- Example per step

Specification and Microarchitecture: Input and Output, Format

Specification

- Input: Requirements from Marketing, CEO (Chief Executive Officer), CTO (Chief Technology Officer), etc.
- Output: Document or model in text/graphics or software (C++, SystemC, SystemVerilog, etc.) format.



Microarchitecture

- Input: Specification + requirement from the designer.
- Output: Typically, a document in text/graphics could be software as well.



This slide shows the first 2 phases, the specification creation, and the microarchitecture creation.

Specification needs requirements from the Marketing, the CEO, or the Chief Executive Officer, as well as the CTO, or the Chief Technology Officer, and others too as required.

Here the input is the requirements as mentioned by all these stakeholders, and the output will be a document or model in terms of text, graphics or software, for example, in C++, SystemC, SystemVerilog, etc.

For the microarchitecture phase, the input will be the specification document created in the first phase we just mentioned, as well as the requirement from the designer. Then the output will typically be a document in text and/or graphics, or even the software itself.

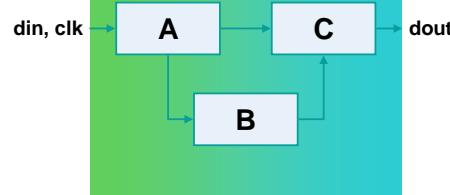
Example: Specification

Let's assume we have a specification, microarchitecture, and RTL.

We are designing a chip called "EX" with:

- Three main partitions "A," "B," and "C."
- Memories in each partition.
- Perimeter I/O.
- 250-MHz clock.
- 10W total power.
- Die size not to exceed $10 \times 10 \text{ mm}^2$ due to custom package requirements.

EX (Block Diagram)



31 © Cadence Design Systems, Inc. All rights reserved.



Let's look at an example now.

Let's assume we have a specification, microarchitecture, and RTL.

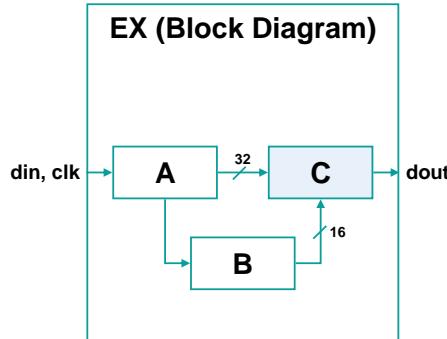
We are designing a chip called "EX" with:

- Three main partitions "A," "B," and "C."
- Memories in each partition.
- Perimeter I/O.
- 250-MHz clock.
- 10W total power.
- Die size not to exceed $10 \times 10 \text{ mm}^2$ due to custom package requirements.

Example: Microarchitecture

For Block C

- 32-bit data bus interface to Block A.
- 16-bit control interface from Block B.
- Use 64 Mb of SRAM.
- Duplicate datapath elements in a parallel implementation.
- Limit of five clock cycles from data input processed to data output.



32 © Cadence Design Systems, Inc. All rights reserved.



Now at the microarchitecture level, out of the 3 sub-blocks in the design, let's take the example of Block C.

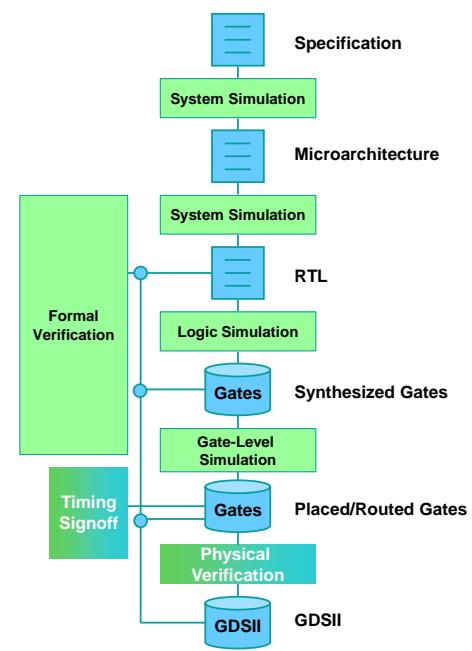
Block C has the following Micro-architecture details:

- *It has a 32-bit data bus interface to Block A.*
- *A 16-bit control interface from Block B.*
- *It uses 64 Mb of SRAM.*
- *It duplicates datapath elements in a parallel implementation.*
- *And has a Limit of five clock cycles from data input processed to data output.*

Verification

Various resources necessary to translate the representations through the verification flow.

System Simulation	Simulation at the behavioral or system level.
Logic Simulation	Simulation at the RTL level.
Gate-Level Simulation	Simulation of the discrete logic.
Physical Verification	Design rule checking, and layout versus schematic checking.
Formal Verification	Logical equivalence checking.
Timing Signoff	Static timing analysis.



33 © Cadence Design Systems, Inc. All rights reserved.



This slide shows the different kinds of testing or verification done at different phases of the design flow. We have the necessary resources in place at each of the phases accordingly.

For the simulation at the behavioral or system level, we have System Level Simulation as the process to go to the next phase in the verification flow.

At the RTL level, we have logic simulation.

We need gate-level simulation to verify the discrete logic.

For design rule checking and layout versus schematic checking, we have physical verification.

We have formal verification for logical equivalence checking

And we need to have a timing signoff for Static Timing Analysis.

Verification Challenges

Verification

- Size and complexity

Larger and more complex designs require sophisticated verification strategies.

- Testbenches

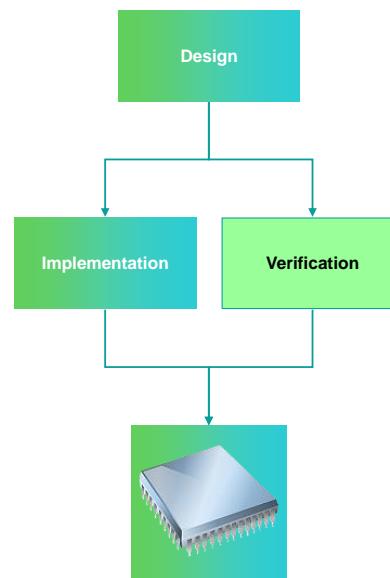
Designers need to create environments to fully test the behavior of their system.

- Corner cases

Designers need to understand the complete behavior of their system, including all of the exception cases, so they can test for them.

- Design for test

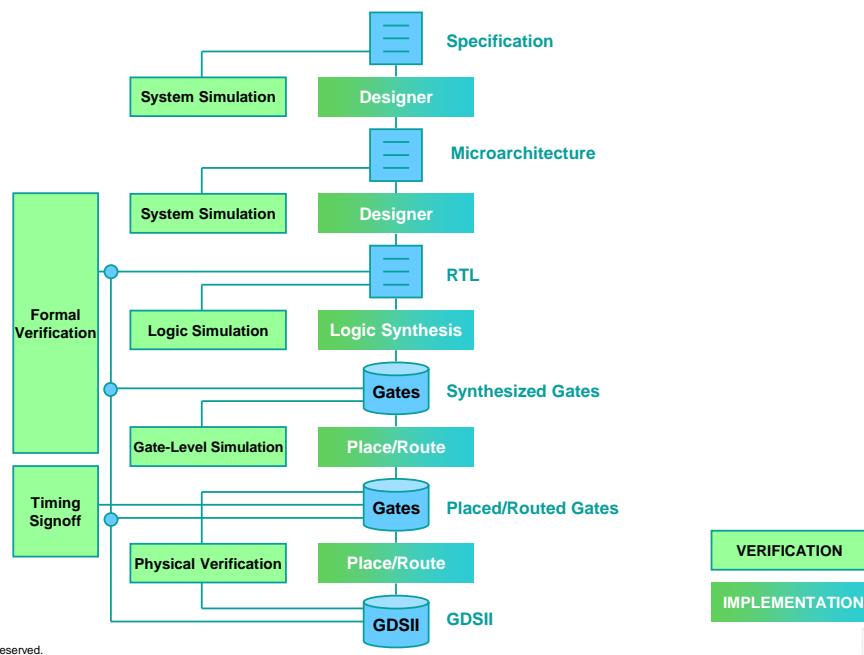
Designers must consider the strategies for testing the final product.



Just like the design challenges we listed out, we have verification challenges too!

- In regards to size and complexity, larger and more complex designs require sophisticated verification strategies.
- Designers need to create environments, which is the complete infrastructure as in testbenches and related components, to fully test the behavior of their system.
- Designers need to understand the complete behavior of their system, including all of the exceptions and corner cases, so that they can test for them.
- Designers must consider the strategies for testing the final product, which constitutes the Design for the test phase.

Implementation and Verification



35 © Cadence Design Systems, Inc. All rights reserved.



Apart from verification, the other process which runs in parallel throughout the design flow at different phases is the implementation process.

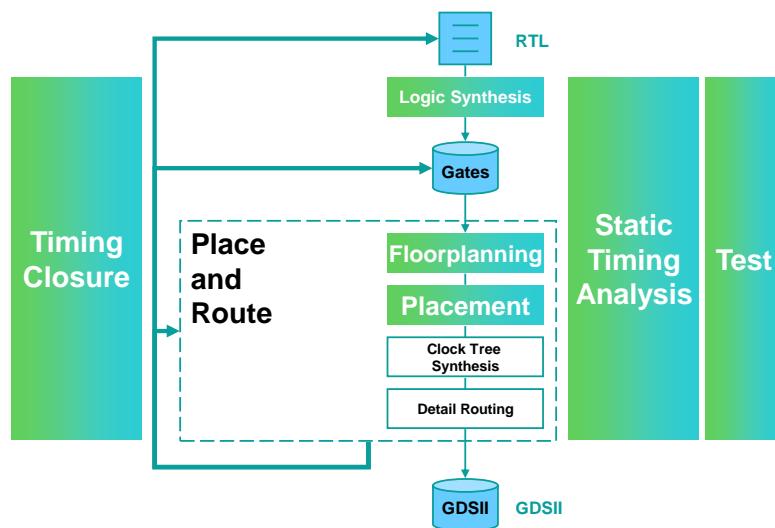
Each phase of the verification will have a corresponding implementation phase, as seen in the flow diagram on this slide.

We have logic synthesis after the Logic simulation process successfully passes.

We have place and route along with gate-level sims, which are run on the verification side.

GDSII is the final output after place and route post-physical verification.

Implementation Flow Overview



36 © Cadence Design Systems, Inc. All rights reserved.

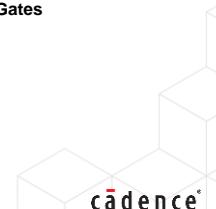
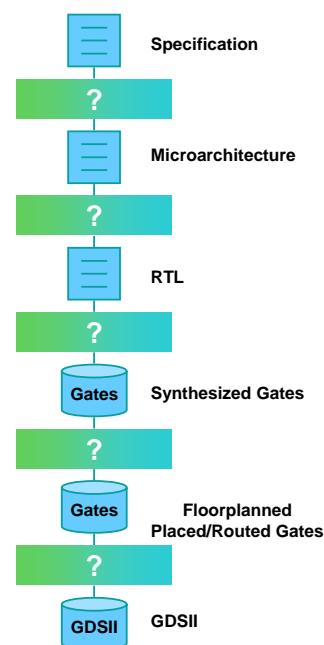


This diagram shows an overview of implementation starting with logic synthesis, which is the process of generating a gate-level netlist from an RTL. The synthesis step is followed by place-and-route. Place-and-route includes floorplanning, placement, clock tree synthesis, and detail routing. Optimizations will be run for setup and hold to meet timing if the timing is not met at the placement, clock tree synthesis, and routing stages. The static timing analysis tool will analyze the timing for violations at each stage, and a report of the violations will be generated. Test structures are added during synthesis to detect manufacturing defects after the parts have been fabricated.

Basic Flow: Implementation

Various representations of logical and physical hardware:

- **Specification:** Textual, graphical, or software representation of the overall behavior and goals of a design.
- **Microarchitecture:** Textual, graphical, or software representation of the initial implementation of the specification.
- **RTL:** Software representation of the design to be implemented.
- **Synthesized gates:** Logical representation of the design after mapping to discrete gates.
- **Placed/routed gates:** Logical and physical representation of the design after place/route.
- **GDSII:** Physical representation of the design just before tapeout.



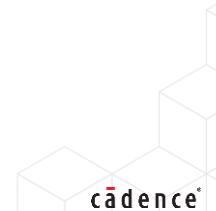
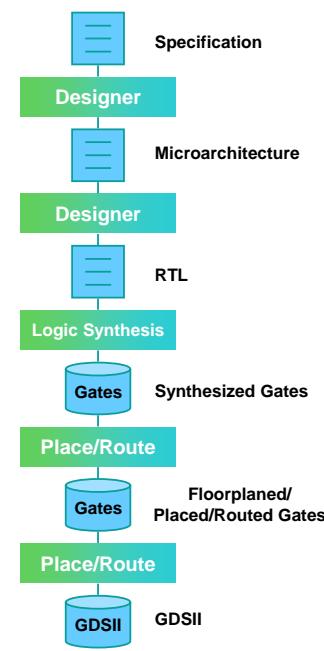
To represent the logical and physical design, create a specification. A specification is the textual, graphical, or software representation of the overall behavior and goals of a design. Also, define the microarchitecture, which is a textual, graphical, or software representation of the initial implementation of the specification. Create an RTL, which is a software representation of the design to be implemented. The RTL will be synthesized into gates which is the logical representation of the design after synthesis maps the netlist to discrete gates. The synthesized gates will be placed and routed into a physical design by the place-and-route tool. After the design is placed and routed, it will be output to a GDS2 file which is the final stage of the implementation process.

Basic Flow: Implementation

(continued)

Various resources necessary to translate the representations through the implementation flow:

- **Designer:** The architects of the chip or the block level design engineers.
- **Logic synthesis:** Process of translating RTL into discrete logic.
- **Place/route:** Process of physical design of the discrete logic.



38 © Cadence Design Systems, Inc. All rights reserved.

There are several different tasks in implementing a design, and therefore, various types of expertise are necessary to translate the representations through the implementation flow. The design engineers are the architects of the chip, or the block, which will be integrated into a top-level design. There are engineers responsible for logic synthesis, which is the process of translating the RTL into discrete logic.

Engineers responsible for place-and-route create a physical design of the discrete logic.

Implementation Challenges

Implementation

- Size and complexity

The larger and more complex the design, the longer it takes to complete.

- Power

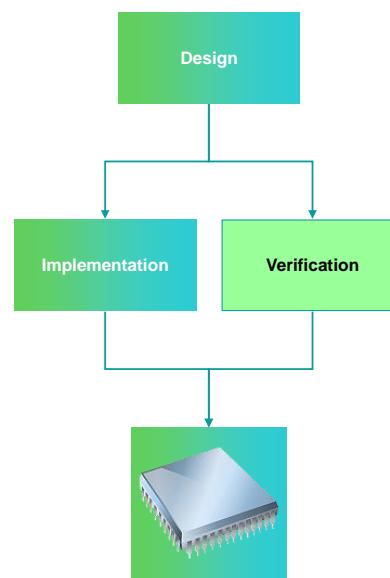
Power must be addressed at many levels during implementation, with very sophisticated strategies to handle the power requirements.

- Timing and design closure

With advanced process technologies, it becomes harder to converge on timing and design rule checks.

- Yield

Affecting the bottom line directly, yield issues must be addressed in the implementation.



39 © Cadence Design Systems, Inc. All rights reserved.



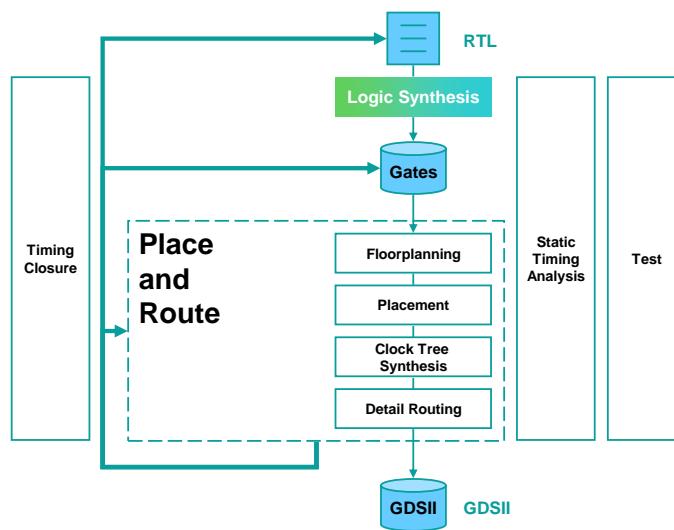
There are several challenges in implementation. The size and complexity of the design are important considerations. The larger and more complex the design, the longer it takes to implement.

If power consumption is an important consideration for the design, power must be addressed at the various stages of implementation, with sophisticated design strategies to handle the power requirements.

Converging timing and fixing design rule violations at advanced process technologies is a constant challenge through implementation.

Yield is measured as the number of working chips or dies on a wafer. To make sure that the parts that are manufactured provide acceptable yields, strategies to maximize yields must be included during implementation.

Logic Synthesis



40 © Cadence Design Systems, Inc. All rights reserved.



Logic synthesis is the first step in implementation.

What Is Logic Synthesis?

The process of parsing, translating, and optimizing RTL code into discrete logic gates.

Example: To determine the feasibility of the design, we need to synthesize the RTL code into gates and measure timing, power, and area.

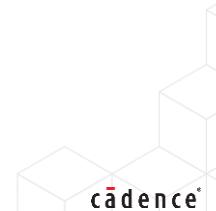
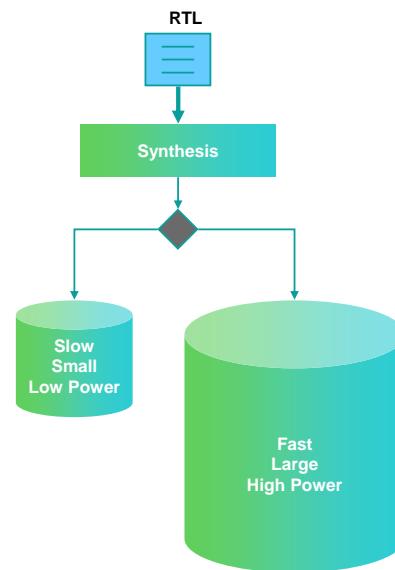


Logic synthesis is where a specification of a design that was written in RTL is converted into a gate-level netlist by the logic synthesis tool. An example of its usage in a sentence is to determine the feasibility of the design; we need to synthesize the RTL into gates and report the timing, power, and area.

How Do We Measure the Quality of the RTL?

By synthesizing to our target technology...

- Timing
 - We can measure the number of logic levels in the design.
 - We can “time” the design against its requirements to see if it meets timing.
- Area
 - We can measure the number of gate units used in the design.
 - We can compare the size of the design against its requirements to see if it meets its area goals.
- Power
 - We can measure the dynamic and leakage power used in the design.
 - We can compare the power numbers against its requirements to see if it meets its power goals.



42 © Cadence Design Systems, Inc. All rights reserved.

Logic synthesis generates a gate-level netlist from the RTL and maps it to a target technology.

A few parameters are used to measure the quality of the RTL. They include timing, area, and power.

To measure the quality of the generated netlist, start with the timing. First, you can measure the number of logic levels in the design. Secondly, you can time the design against its constraints to see if it meets timing.

To measure the area, you measure the number of gate units used in the design. And then, compare the size of the design against its requirements to see if it meets its area goals.

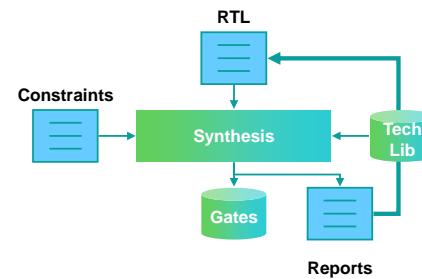
For power, you measure the dynamic and leakage power used in the design. And then, compare the power numbers against its requirements to see if it meets its power goals.

When Do We Change the RTL?

Based on the constraints and the synthesis reports, we will know...

- The timing of the design and which specific paths it violates.
- The overall area of the design.
- The overall power of the design.

If we violate timing, area, or power by a large margin, we *may* have to modify the RTL accordingly.



43 © Cadence Design Systems, Inc. All rights reserved.



Sometimes during synthesis, when you check the quality of results, you might need to change the RTL.

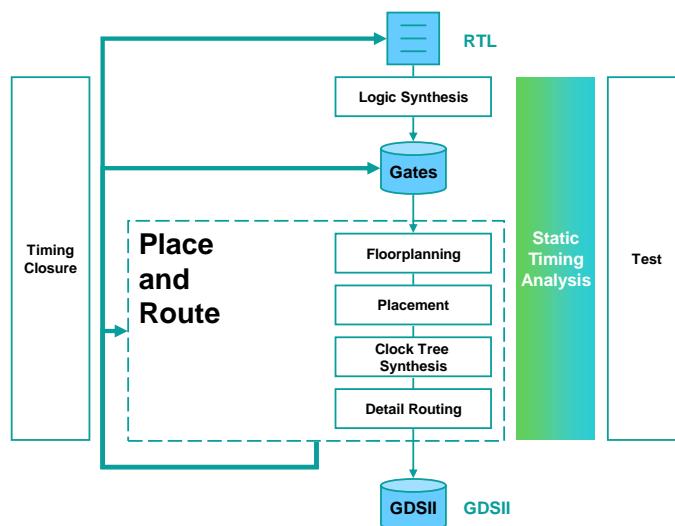
But this depends upon the final results and specifications of the design.

At the end of the synthesis process, you generate various reports and analyze the results based on input constraints and original specifications.

You check for the timing violations, overall area, and power of the design.

If there are some real issues and violations which are not resolvable by modifying the synthesis constraints, then in such scenarios, you might need to modify the RTL.

Static Timing Analysis



44 © Cadence Design Systems, Inc. All rights reserved.



It is important to verify that you are meeting the timing specification during the different stages of the design flow.

As seen here, static timing analysis is used across the board at various steps.

After each design stage, you must constantly run static timing analysis to validate whether your design is meeting your specification, which was previously defined in the form of timing constraints.

What Is Static Timing Analysis or STA?

A method of computing the expected timing of a digital circuit without using SPICE (transistor level) simulation.

- Static timing analysis (STA) is the preferred method for timing signoff.
- After logic synthesis, STA is used to determine if the timing goals of the design are met.
- If the timing goals are not met, then the RTL may have to be modified.
- During the physical design (for example, floorplanning), STA is again used to determine if the timing goals of the design are met.
- If the timing goals are not met, then the RTL may have to be modified.

Example: To determine the timing of the design, we ran static timing analysis after synthesis and saw several paths violating their setup time requirements.



Unlike SPICE analysis, which analyzes dynamic waveforms of the signals traveling through the design, static timing analysis uses static tabulated values of timing checks and derived values from timing constraints such as clock periods to compute the required times. Then, it calculates the delay of the path via tabulated values from a timing library and computes arrival times. Finally, static timing analysis determines if a path meets the required time or violates the required time.

If a goal is not met, sometimes, the design is modified using specialized optimization techniques in synthesis. But, if all other workarounds fail, then you might have to go back to modify your RTL.

Example

- Assume the following RTL code:

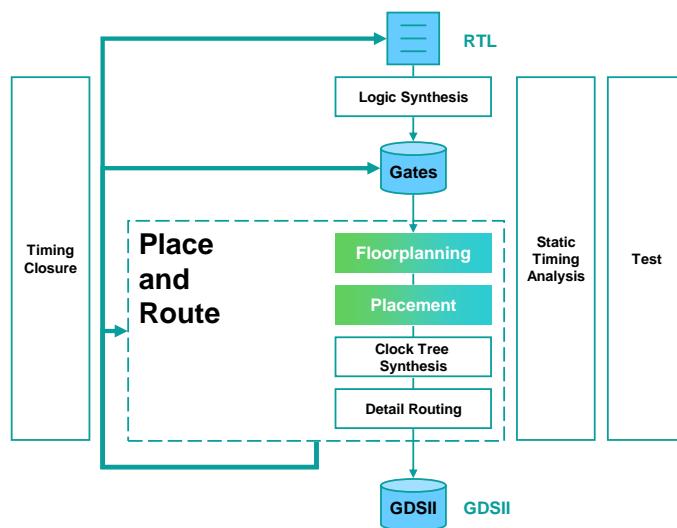
```
always @ (posedge clk) begin  
    z <= a + b;  
end
```

- Depending on the constraints, the netlist could be synthesized to:
 - A small, but slow ripple-carry-adder
 - A larger sized, but faster carry-look-ahead adder
- STA is used *during synthesis* to determine which adder meets the constraints of the design.



In the example, an adder is used to compute the sum of A and B. The implementation selection of the adder is determined by synthesis by validating the timing of the design through the path of the adder and many other paths. If the adder falls on a critical path, a large but faster adder may be implemented; otherwise, a smaller adder may be picked for implementation. Static timing analysis is used to both select the correct implementation and to validate whether the implementation achieves the desired results.

Floorplanning and Placement



47 © Cadence Design Systems, Inc. All rights reserved.



At the top level of the design, floorplanning is the process of laying out the physical partitions of a design to determine the size and connectivity of each partition relative to the chip.

Floorplanning is also where you define the size of the design, place the blocks in the design and plan the power for the design. Placement is the step after floorplanning, where the standard cells, which are discrete components, are placed in the design in specific locations.

What Are Floorplanning and Placement?

Floorplanning is the process of laying out the physical partitions of a design to determine the size and connectivity of each partition relative to the chip.

Floorplanning has a significant effect on the constraints of a design and, therefore, has a significant effect on the RTL.

- The size and the aspect ratio of the block may dictate changes to the RTL.
- RAM sizes, aspect ratios, timing, and power characteristics may dictate changes to the RTL.
- Pin placement may cause the input and output delays to the block to change significantly, requiring attention at the RTL level.

Placement is the process of finding specific locations for each discrete component of an integrated circuit.

Placement has a significant effect on the accurate implementation of the design; therefore, it has a significant effect on the RTL.

- The critical path may be very different between synthesis and placement, so the designer needs to be careful which path in the RTL to fix.
- Because placement (and CTS) can add a significant amount of logic for placement optimization, buffering, etc., the designer needs an accurate area/power estimate to determine if the area/power goals will be met.



Floorplanning has a significant effect on the constraints of a design and, therefore, has a significant effect on the RTL. If the standard cells that were derived from the RTL cannot be placed in the selected size and aspect ratio of the die, this may dictate changes to the RTL.

Additionally, RAM sizes, aspect ratios, timing, and power characteristics may dictate changes to the RTL.

The pin placement may cause the input and output delays to the block to change significantly, requiring attention at the RTL level.

In addition to floorplanning, the placement has a significant effect on the implementation of the design, which in turn may have a significant effect on the RTL.

The critical path may be very different between synthesis and placement, so the designer needs to be careful about which path in the RTL to fix.

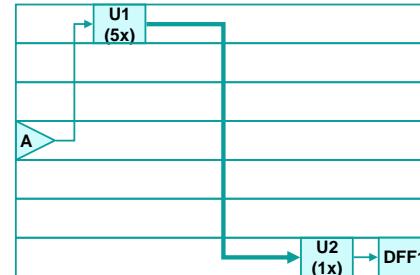
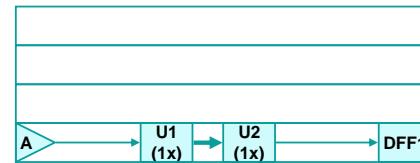
Because placement and clock tree synthesis can add a significant amount of logic for placement optimization and buffering, the designer needs an accurate area and power estimate to determine if the area and power goals will be met.

Example: Placement

In short, placement gives a more accurate picture of the timing and area versus logic synthesis.

Consider the following example:

- Gates U1 and U2 and part of the critical path.
- In logic synthesis, U1 and U2 are “placed” close together, have minimal power and size, and the delay estimated for them is very small.
- In placement, U1 and U2 are placed very far apart, U1 is upsized significantly, so the delay through U1 and through the connection is vastly different.



49 © Cadence Design Systems, Inc. All rights reserved.



The placement gives a more accurate picture of the timing and area requirements as compared to the estimates at logic synthesis.

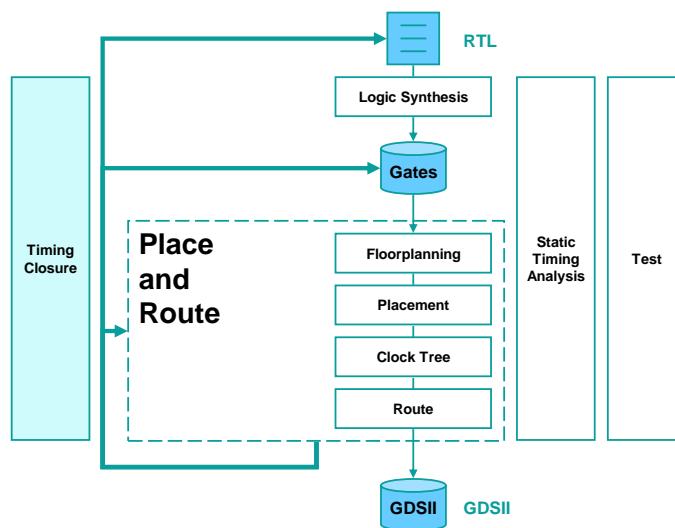
Consider the following example.

Gates U1 and U2 are a part of the critical path.

In logic synthesis, U1 and U2 are timed as being close together, have minimal power and size, and the delay estimated for them is very small.

However, during placement, U1 and U2 are placed very far apart. As a result, U1 is upsized significantly to meet the drive requirements of the larger net delay so that the timing will not be violated.

Timing Closure



50 © Cadence Design Systems, Inc. All rights reserved.



After the design is placed and routed, the timing might still not be met for various reasons. Making sure that your design meets the desired specifications is part of the timing closure process.

What Is Timing Closure?

The process of iterating through synthesis, physical design, and verification to converge on the timing of a digital design.

- One of the most challenging aspects of chip design is achieving timing closure.
- Example: To achieve timing closure, the design team went through 20 iterations over a one-month period.



The initial placement of the originally synthesized netlist from the RTL design will be based on an initial design, and further modifications will be necessary.

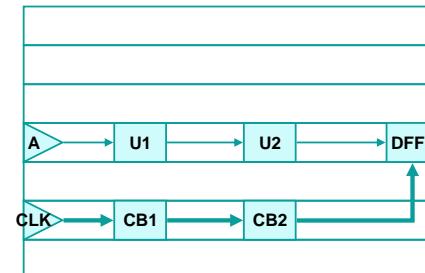
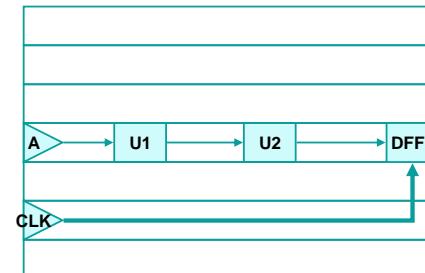
For example, the design will require power planning, test insertion, clock tree synthesis, etc.

These changes affect the timing of the design. To converge the timing of the design, several more modifications to the design must be made, including critical region re-synthesis, redoing some floorplanning, moving the placement of the cells, re-routing certain parts of the modified design, etc.

All these aspects are considered part of timing closure. Timing closure requires multiple loops or iterations.

How Does Timing Closure Affect the Design?

- CTS adds the actual clock buffers, whereas the clocks were “ideal” beforehand.
- Actual clock skew and latency need to be considered for timing.

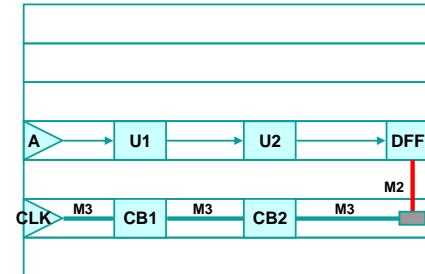
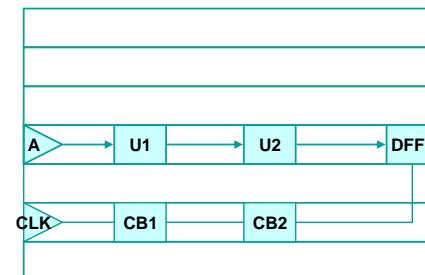


After the clock is routed to the sequential elements in the design, the timing of the design might be far off from what you might have expected. To fix timing, the stage called timing closure will need to implement optimizations to converge or meet timing. Depending on how you implemented the clock tree, it might or might not meet timing in the first round.

How Does Timing Closure Affect the Design? (continued)

Routing adds the actual wire connections, whereas the routing interconnects were estimated beforehand.

- Actual delays with crosstalk can affect timing.
- Congestion can affect routability, which in turn can affect the floorplan, which in turn can affect the overall design constraints greatly.

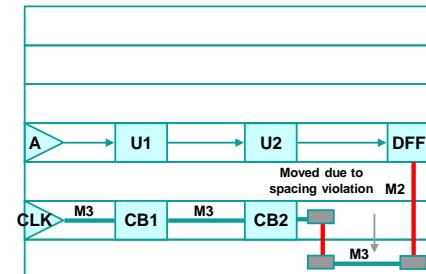
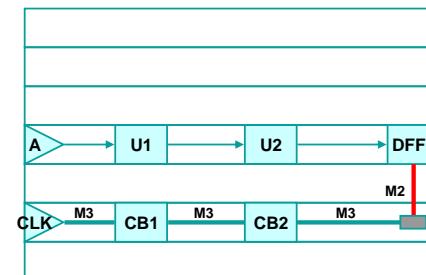


During routing, actual wire connections are added, while the routing interconnects were estimated until this point. Taking into account actual delays with crosstalk can affect timing. Congestion of standard cells in a particular area can affect routability, which can affect the floorplan, and also affect the overall design constraints. In the second diagram, you see that most nets have been routed with metal three, and one net has been routed with Metal 2, which requires a via to connect between the two layers. These actual routes and their associated resistance and capacitances may result in a very different timing report than the reports generated in previous steps, where routes were estimated and not actual.

How Does Timing Closure Affect the Design? (continued)

Physical verification adds yet another dimension since fixing a design rule may break a timing path.

- A spacing violation causes wires to be moved to avoid the violation.
- The wires moving changes the timing and could cause a timing violation.

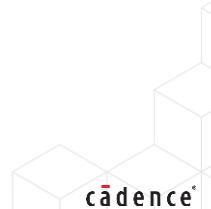


Physical verification is the process of checking for design rule violations which include open wires or shorted wires. This step adds yet another dimension since fixing a design rule may break a timing path. A spacing violation may result in wires being moved to avoid the violation. The wires that move could change the timing to cause a timing violation.

How Does Timing Closure Affect the RTL? (continued)

CTS, routing, and physical verification can cause changes that affect timing and, therefore, could influence the RTL.

- Changes to the netlist itself or to the placement, CTS, or routing can fix the timing violation without having to modify the RTL.
- In some cases, modifying the RTL is the only choice.
- Modifying the RTL is usually the last resort because it has a very significant impact on the schedule.



Clock tree synthesis (CTS), routing, and physical verification can cause changes that affect timing and influence the RTL.

Changes to the netlist or to the placement, clock tree synthesis, or routing can fix the timing violation without having to modify the RTL.

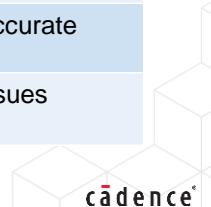
In some cases, modifying the RTL is the only choice assuming all other strategies have been tried to no avail.

Modifying the RTL is usually the last resort because it has a very significant impact on the schedule.

Implementation on RTL

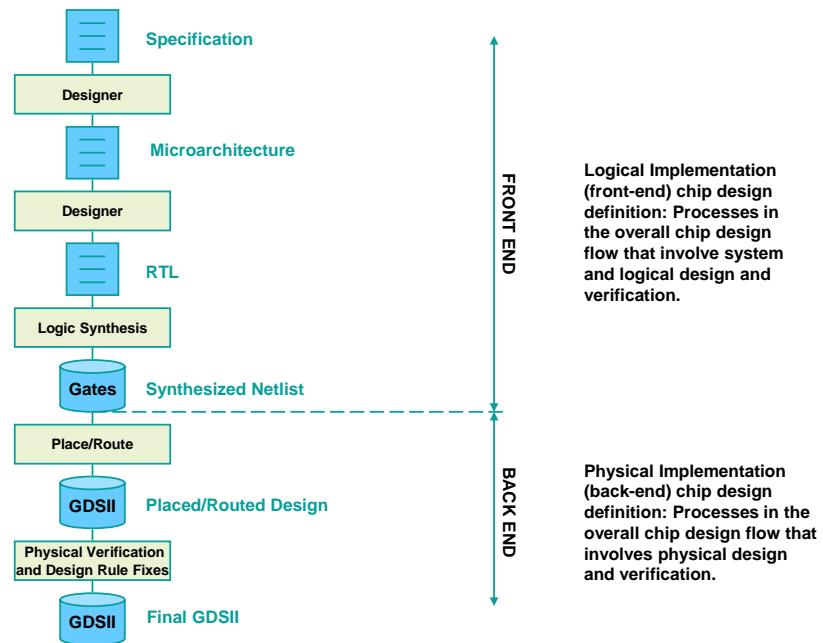
	Phase in Implementation	Comments
Logic Synthesis	Early	Timing, area, and power are estimated
Static Timing Analysis	Throughout	Timing is measured throughout the flow
DFT	Throughout	Timing, area, power, DFT rules are affected
BIST	Throughout	Timing, area, and power are affected
JTAG	Throughout	Design is affected at the chip level
Floorplanning	Early/Middle	Design is constrained more accurately here
Placement	Middle	Timing, area, and power are more accurate
Timing Closure	Late	Timing can be affected by various issues

56 © Cadence Design Systems, Inc. All rights reserved.



This table shows the implementation steps, the phases where the steps are used, and how they affect the design in terms of power, timing, and area.

Implementation Flow (continued)



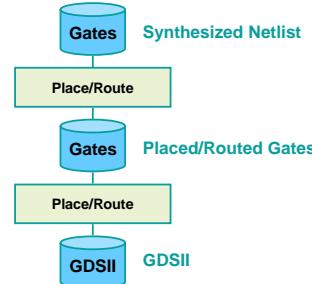
Logical Implementation (front-end) chip design definition: Processes in the overall chip design flow that involve system and logical design and verification.

Physical Implementation (back-end) chip design definition: Processes in the overall chip design flow that involves physical design and verification.

Back End or Physical Design

The terms “physical design” or “back end” or “place/route” encompass many process steps, such as:

- Floorplanning
- Placement
- Clock Tree Synthesis (CTS)
- Route
- Extraction
- Delay Calculation
- Static Timing Analysis (STA)
- Signal Integrity
- Design Optimization
- Physical Synthesis
- Design Verification
- Mask Prep

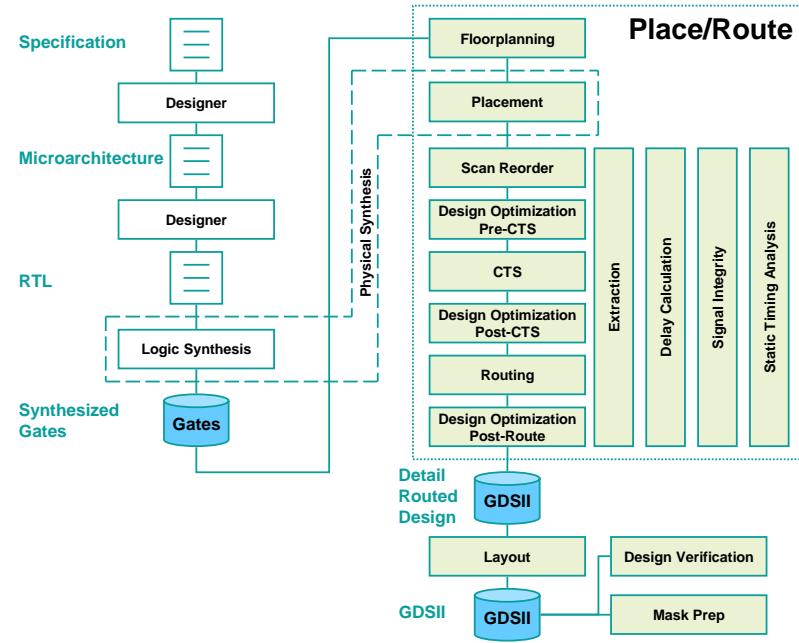


58 © Cadence Design Systems, Inc. All rights reserved.



These are the main steps in backend or physical design. They include floorplanning, placement, clock tree synthesis (CTS), routing, extraction of the resistances and capacitance in the design, delay calculation to compute the timing, static timing analysis, which compares the existing timing to the constraints and generates a report, signal integrity analysis, design optimization, physical synthesis, design verification, and finally, mask prep.

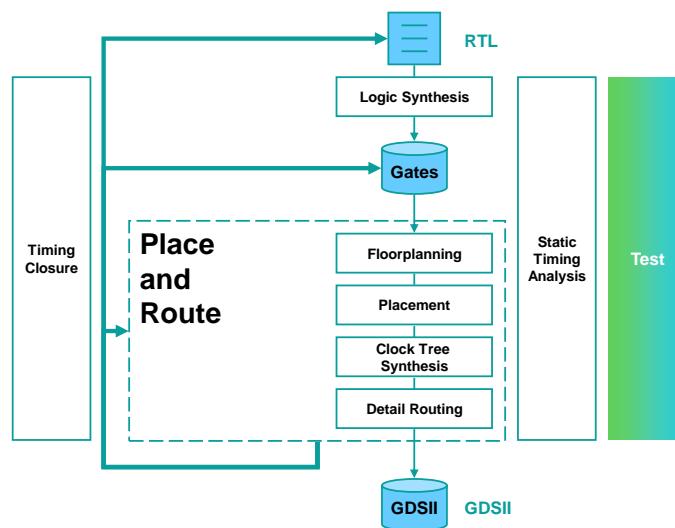
Back-End Implementation Flow



59 © Cadence Design Systems, Inc. All rights reserved.

This is the main physical implementation flow we'll cover during the class and the rest of the course. We'll briefly introduce each topic, understand their meaning and importance in the overall flow, define the inputs and outputs, and see a brief example. During the course, we'll dive into much greater detail on each topic.

Testing Phase in Implementation



60 © Cadence Design Systems, Inc. All rights reserved.



A part of the implementation is to add test structures to the design so that during manufacture, defects in the part can be detected.

What Is Design for Test (DFT)?

Design techniques to add testability features in integrated circuits to make it easier to apply manufacturing tests.

How does DFT affect the RTL?

- Timing

The additional MUX in front of the RTL requires the designer to take this extra level of logic into account when writing the RTL.

- Area and power

The additional MUX also increases the overall area and power of the design.

- DFT rules

Registers inside of the design must be observable and controllable, so the RTL code may need modification to make sure the DFT rules are met.



Let's understand design for test or DFT.

The test process refers to testing the ASIC for manufacturing defects on automatic test equipment called ATE. It is the process of analyzing the logic on the chip to detect logic faults.

Before inserting test points in the design, you should plan and find out the DFT requirements of your design to implement the same.

So, do you need to consider DFT at the RTL level? Yes.

When you insert DFT logic, you need to account for a few things in advance at the RTL level.

One is the timing; as there is an additional mux in front of the RTL, it requires the designer to take this extra level of logic into account when writing the RTL.

The presence of the additional mux also increases the overall area and power of the design.

When you do DFT insertion, you also need to ensure that the registers inside the design must be observable and controllable. And so, the RTL code will need modifications to make sure the DFT rules are met.

What Is Built-in Self-Test (BIST)?

Extra logic in a design to verify all or a portion of the internal functionality.

How does BIST affect the RTL?

- Area and power

The addition of the MBIST and LBIST structures should be considered when calculating the overall area and power goals for the design.

- Timing

Timing is affected by the additional logic, so this should be considered when writing the RTL.



Built-in self-test, or BIST, is inserted into a design to generate self-test patterns. The intent is to verify the internal functionality.

These structures can be added for memory or logic to improve overall testing efficiency.

Adding BIST logic affects the RTL as you need to account for the area, power, and timing due as a result of this additional logic.

What Is Joint Test Action Group (JTAG)?

Extra logic in a design to verify all or a portion of the internal functionality.

How does JTAG affect the RTL?

- Not very much. Since JTAG is a very mature standard and many chips require it, the TC and I/Os are often taken into account at the chip level already.
- Chip architects must budget for JTAG at the chip and board level.



A JTAG macro contains test logic to access and control DFT features built into the design.

JTAG, or Joint Test Action Group, is an IEEE 1149.1 standard for the controller of test access ports used for testing designs with boundary scans.

It is an extra logic in a design to verify all or a portion of the internal functionality.

The addition of the JTAG logic doesn't affect the RTL too much.

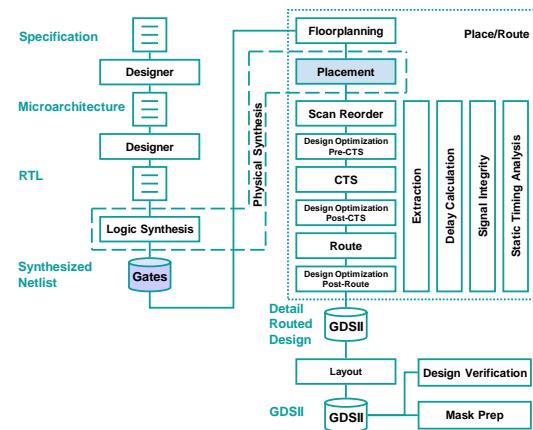
The JTAG is a very mature standard logic used by many chips. The Tap Controller and inputs and outputs are often already taken into account at the chip level.

So, the chip architects must budget for JTAG at the chip and board level.

What Do We Need Physical Synthesis?

The placement is the process of placing the standard cells in a floorplanned design.

- **Example:** After the chip was floorplanned, we performed placement and discovered the floorplan was too small to fit all of the cells and macros in the design.
- **Question:** How can we avoid this problem?



64 © Cadence Design Systems, Inc. All rights reserved.



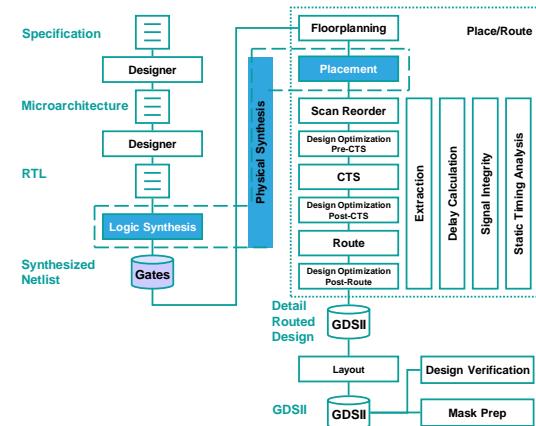
Placement is the process of placing the standard cells in specific locations in the physical design after the design was floor-planned. An example of how you might hear the word placement used in context is shown here. After the chip was floor-planned, we ran placement and discovered that the floorplan was too small to fit all of the cells and macros in the design.

A way to avoid this problem might be to change the RTL or to change the aspect ratio or size of the floorplan.

What Is Physical Synthesis?

The combination of logical synthesis and placement.

Example: To meet timing, we ran physical synthesis which, in addition to upsizing and downsizing components, also ran logic restructuring.



65 © Cadence Design Systems, Inc. All rights reserved.



Physical synthesis is the combination of logical synthesis and placement.

Traditional synthesis uses estimated models for the wires, which do not provide accurate wire delay information, especially for designs where a significant portion of the delays are contributed by the wires. Consequently, you can see relatively big differences in performance, area, and power between the logic and physical designs.

As a result, it gets difficult to close timing in the backend. To take care of the gaps for better timing closure, you have floorplan and placement information that can be used during logic synthesis to provide better results than correlate better with the backend result. You can think of floorplanning as a bridge to close the implementation gap between logic synthesis and physical synthesis during place and route. Therefore, to meet timing, you run physical synthesis along with other optimizations like upsizing, downsizing, and logic restructuring.

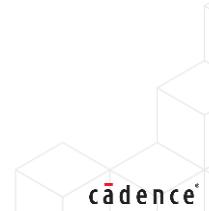
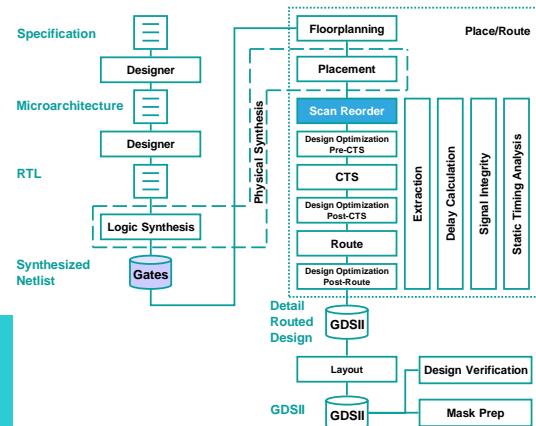
What Is Scan Reorder?

Process of re-connecting the scan chains in a design to optimize for routing, timing, etc.

Example: Since logic synthesis arbitrarily connects the scan chain, we need to perform a scan reorder after placement so that the scan chain routing will be optimal.

What is a scan chain?

A scan chain is the connection of the flip-flops in a design, such that test patterns can be scanned in and results scanned out during automated testing.

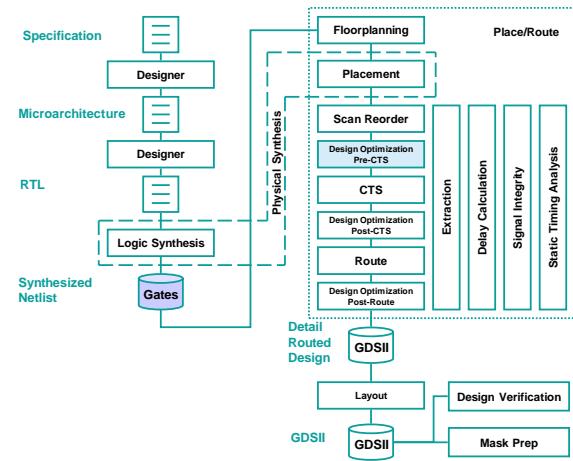


Scan cells are inserted into a netlist for testability during or after placement. The scan components are connected in a chain such that test patterns can be shifted in and out or scanned in and scanned out. Scan reordering is the process of reordering the connections between scan components to save routing resources. The example describes how you might hear about scan reordering in the context of optimal routing.

What Is Design Optimization?

Process of using automated algorithms to improve the quality of a digital design.

Example: After initial placement, we run a pass of pre-CTS design optimization to fix timing violations that may show up now that the design is placed, and we have delays based on the estimated interconnect.



67 © Cadence Design Systems, Inc. All rights reserved.

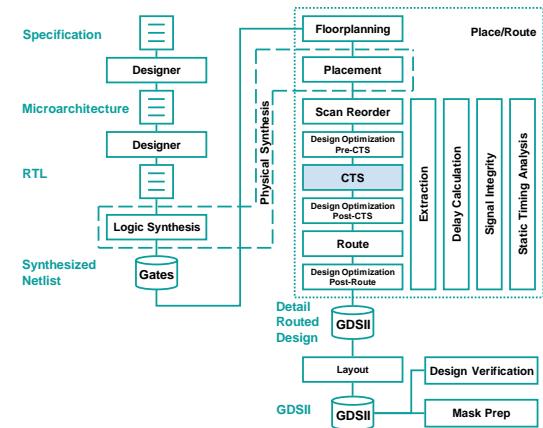


Design optimization is the process of improving the power, performance and area, also called PPA, of the design. Optimization occurs at various stages of the flow. The stage after placement is known as pre-CTS. The stage after clock tree synthesis is known as post-CTS. The stage after routing is known as post-route. In this example, it was discovered that the timing analysis after placement contained violations. Since this is before clock tree synthesis, pre-CTS optimization was run to fix the violations that are based on the estimated interconnect delays.

What Is Clock Tree Synthesis?

Process of inserting buffers in the clock path, with the goal of minimizing clock skew and latency to optimize timing.

Example: We ran clock tree synthesis on the example block and saw a large clock skew due to bad clock constraints. We re-ran clock tree synthesis with better constraints to get an optimal result.



68 © Cadence Design Systems, Inc. All rights reserved.

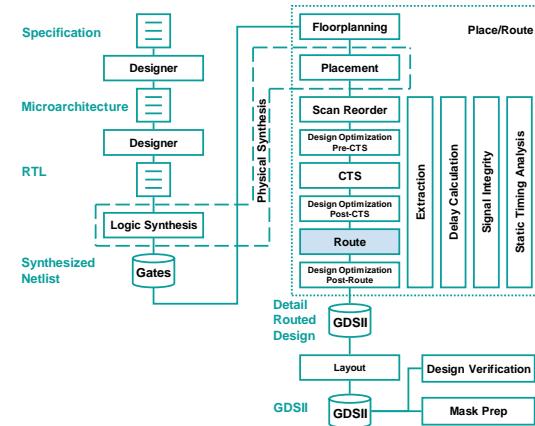


Clock tree synthesis is the step after pre-CTS optimization and is the process of inserting buffers in the clock path to minimize clock skew and latency, to optimize timing. In the example, incorrectly defined clock constraints led to a large clock skew value, and the solution was to re-run clock tree synthesis with corrected clock constraints.

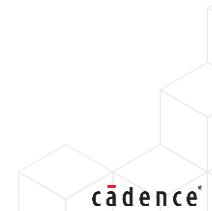
What Is Route?

Process of connecting the pins of the standard cells, macros, and I/Os of a digital design to specific metal layers in the process technology to match the schematic.

Example: We ran a preliminary route on the example block and saw that routing congestion was an issue. To fix it, we re-ran placement with a placement density screen to force a lower utilization in that area and allow for more routing resources.



69 © Cadence Design Systems, Inc. All rights reserved.

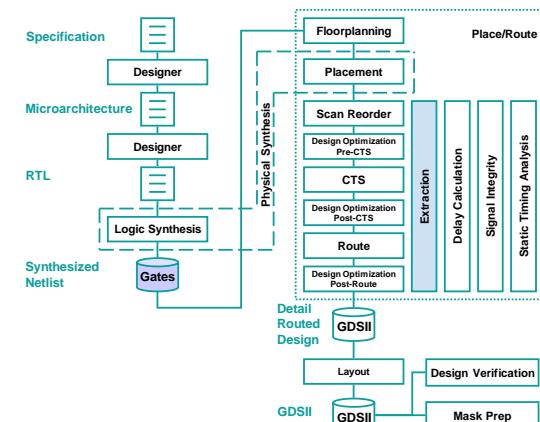


Routing is the process of connecting the pins of the standard cells, macros, and IOs of a design to specific metal layers, in the process technology to match the logical netlist. In the example, an initial route revealed congested routing, which can lead to timing and signal integrity issues. The solution was to run placement with density screens in congested areas, which alleviated placement congestion, which in turn alleviated route congestion.

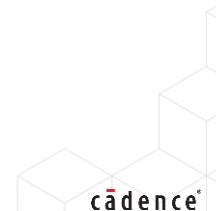
What Is Extraction?

Process of calculating the parasitic resistance and capacitance of the interconnect of the physical design.

Example: Extraction can be performed at various parts of the design with varying accuracy. The most accurate results are achieved when extraction is performed on a fully routed design, because all of the nets are of known metal type and length. There are no estimates for nets at this point.



70 © Cadence Design Systems, Inc. All rights reserved.

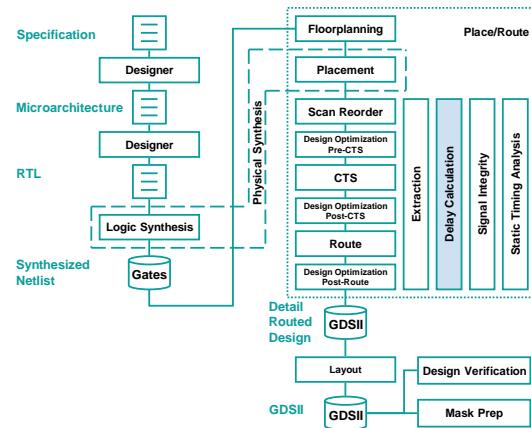


Each interconnect or wire, or net in a design can be modeled as a network of resistance and capacitance. Modeling techniques vary from extraction using 3D field solvers to 2D extraction tools. Field solvers are much more accurate compared to 2D extraction tools but consume a lot of time. Most design tools use something called 2.5D extraction to improve accuracy and speed up runtime and use correlation mechanisms to compare with the 3D field solvers. Extraction can be performed at various steps of the design with varying accuracy. But the most accurate results are achieved when the extraction is performed on a fully routed design because the routing length and the metal type are known at this stage. Extracted values of resistance and capacitance are saved in a Standard Parasitic Extraction Format file.

What Is Delay Calculation?

Process of computing the delay of interconnect and standard cells in a digital design.

Example: In the example design, delay calculation was performed after CTS and also after final route. Using the delay information, we were able to find several timing violations in the design.



71 © Cadence Design Systems, Inc. All rights reserved.

The logo consists of the word "cadence" in a lowercase, sans-serif font, with a red horizontal bar above the letter "e".

Delay calculation is the process of taking the extracted interconnect parasitic RC information and converting it to delay values. These delays are then fed into the timing tool to see if your design is meeting the specifications. The efficacy of the delay-calculation tools can depend on the quality of input parasitic RC information.

What Is Signal Integrity?

Unintended effects on digital signals caused by interconnect parasitic resistance or capacitance that causes noise and/or changes delays.

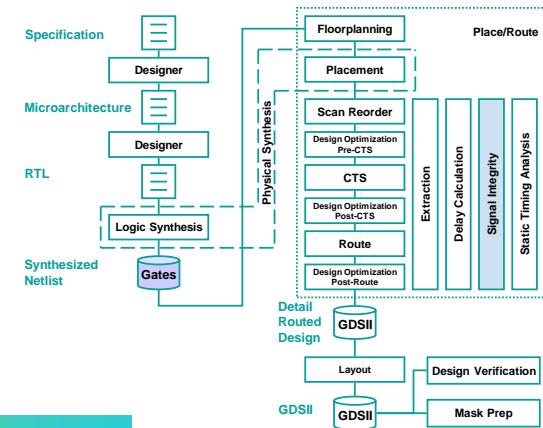
Example: In our example design, we saw signal integrity (SI) effects such as noise-on-delay and glitches due to long nets that were running in parallel.

What is noise-on-delay?

Crosstalk-induced delay or incremental delay due to coupling capacitance.

What is a glitch?

A glitch is a bump or change in value caused by a changing signal affecting a neighboring wire capacitance.



72 © Cadence Design Systems, Inc. All rights reserved.



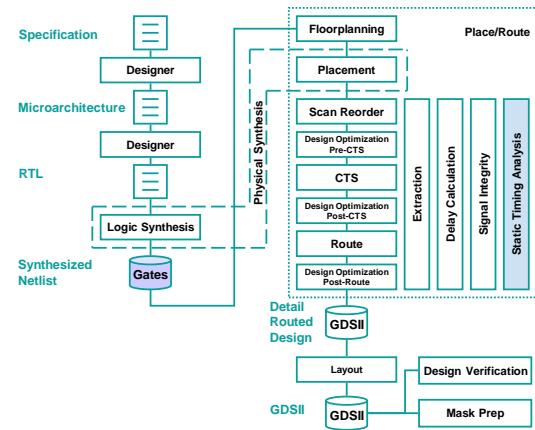
High-speed signals interact with each other and can cause glitches on nearby signals. Coupling capacitance between the nets can cause delays on the nearby nets whose signals are weak. Signal integrity is the process of determining the impact on the timing and functionality of these effects.

What Is Static Timing Analysis?

Process of computing the timing of logically related paths for a digital design without regard to large-scale functional behavior.

STA is the preferred method for timing sign-off, because the majority of ASIC vendors and foundries have adopted it.

- **Example:** To determine the timing of the design, we ran STA after detail route, and saw several paths violating their setup time requirements.



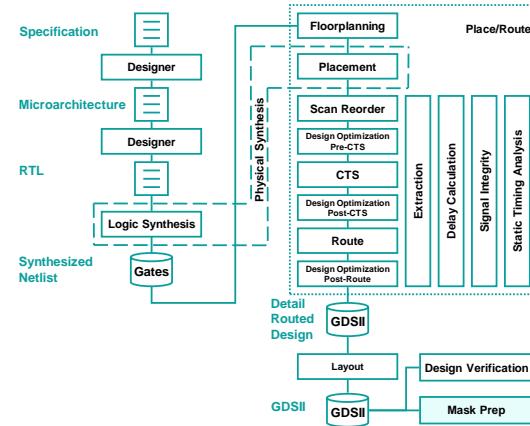
73 © Cadence Design Systems, Inc. All rights reserved.



Static timing analysis is the process of calculating delays of each path through the cells and interconnects to determine whether it meets the required specifications.

What Is Mask Prep?

Process of creating the mask set from the GDSII database to allow chip manufacturing.



74 © Cadence Design Systems, Inc. All rights reserved.



Mask prep is the process of creating masks for manufacturing the design with the GDSII files that are generated by the physical design tool.

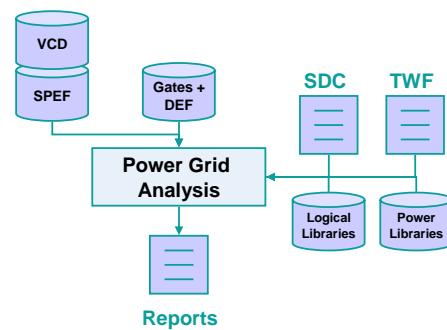
Power Grid Analysis, IR Drop, and EM: Input and Output, Format

- Input

- Gate-level netlist in the Verilog language + DEF
- Power-characterized libraries in tool-specific format
- Timing libraries in Liberty (.lib) format
- Timing constraints in SDC format
- Extraction data in SPEF format
- Timing Windows File (TWF)
- Value-change-dump file (Optional)

- Output

- IR drop reports
- EM reports



** IR – Current Resistance Drop

** EM – Electromigration

In this slide, you see the inputs and outputs for power grid analysis, IR drop, and electromigration. Some of the files are specific to the design, for example, the Verilog gate-level netlist, the design constraints, also known as the SDCs, and the Design Exchange Format or def files. Other files, such as the power-characterized libraries and timing libraries, are technology files and will typically be provided by the foundry that manufactures the chips. The outputs of the analysis are IR drop, and EM reports that you can use to identify and fix the problems in the design.

Terms and Definitions

Floorplanning	Process of deriving the die size, allocating space for soft blocks, planning power, and macro placement.
Placement	Process of placing the standard cells in a floorplanned design.
Clock Tree Synthesis	Process of inserting buffers in the clock tree of a digital design.
Route	Process of connecting the pins of the standard cells, macros, and I/Os of a digital design to specific metal layers in the process technology to match the schematic.
Extraction	Process of calculating the parasitic resistance and capacitance of the interconnect of the physical design.
Delay Calculation	Process of computing the delay of interconnect and standard cells in a digital design.
Static Timing Analysis	Process of computing the timing of logically related paths for a digital design without regard to large-scale functional behavior.
Signal Integrity	Unintended effects on digital signals caused by interconnect parasitic resistance or capacitance that causes noise and/or changes delays.
Design Optimization	Process of using automated algorithms to improve the quality of a digital design.
Physical Synthesis	Process of combining logic synthesis and placement to improve the accuracy of the physical implementation of a digital design.
Design Verification	Process of physically verifying the design rules and backend checks of a design.
Mask Prep	Process of creating the mask set from the GDSII database to allow chip manufacturing.



We need to define many terms for the next slide, which is the overall physical implementation flow we'll cover in the class. Just use this as a reference slide, as we'll cover the details of each step during the lecture.

How Do We Solve the Complex IC Design and Verification Problem?

Divide and Conquer

As we discussed, there are many complex aspects.

- Overwhelmingly complex functional requirements need to be defined correctly and completely in a specification.
- Specification needs to translate into a physical design.
 - This requires multiple levels of abstraction, all of which need to be verified against the original intent.
 - We even need to consider quantum physics effects in the physical layout.
- The physical design needs to be economically manufacturable and testable.

There is almost zero room for error.

- A single mistake is likely to kill the entire project.
- It's a physical entity, not software that can be patched.

A complex IC requires literally hundreds or thousands of hardware engineers to design and verify.



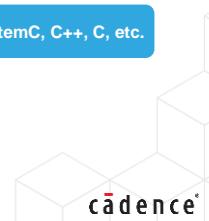
The most effective way to solve complex IC design and verification problems is by divide and conquer. There are many complex aspects to the process, from specifications to implementation. To start with, complex functional requirements need to be defined correctly and completely in a specification. The specification needs to be translatable into a physical design. To translate the spec into a physical design, there are multiple levels of abstraction, all of which need to be verified against the original intent. Because of shrinking geometries, we even need to consider quantum physics effects in the physical layout. Even if it's possible to manufacture the design, the physical design needs to be economically manufacturable and testable. Therefore, there is almost zero room for error. A single unforeseen error can kill the entire project. Since the final output is a physical part, the flaws can be fatal to the project, unlike software that can be patched.

A complex IC can require hundreds or thousands of hardware engineers to design and verify.

What Does Divide and Conquer Mean in This Context?

Each aspect of the Design and Verification is likely to use multiple languages and file formats for one or more:

- Functional concept experimentation and modeling → SystemC, C++, Matlab, etc.
- Functional definition → Skill scripting, SystemVerilog, IPXACT, UPF, etc.
- Execution → Python, Tcl, Linux Shell scripting, VSIF, Cloud/Server Farm management, etc.
- Functional verification with simulation/formal proofs/hardware acceleration/emulation → SystemVerilog, UVM, SVA, PSS, SystemC, C++, C, etc.
- Physical description and timing constraints → ECSM, .lib library format, SDC, IBIS, GDS II, etc.



Each aspect of the design process requires an understanding of various programming languages and file formats, depending on what aspect of the design process is being worked on. C++, SKILL® and Tcl are examples of programming languages, whereas .lib, GDSII and ECSM are examples of file formats.

How Many Software Tools Do We Need?

There are many separate software tools required in the flow.

All of these are integrated into "platforms" or "suites."

A single IC design and verification flow may not utilize all tools listed.

We will explain these different tools and their purpose as the course progresses.

To give an example of scale, an extract from the www.cadence.com A-Z list of tools.

- | | | | |
|---|--|--|--|
| A <ul style="list-style-type: none">• AI Boost NNE 110• Allegro ECAD-MCAD Library Creator• Allegro Package Designer Plus Silicon Layout Option• Allegro PCB Symphony Team Design Option• Allegro X AI• AWR Analyst Software• AWR Microwave Office• AWR VSS Software | <ul style="list-style-type: none">• AI Max Multi-Core• Allegro EDM Solution• Allegro Package Designer Plus SiP Layout Option• Allegro PCB Symphony Team Design Option for IC Packaging• Allegro X Design Platform• AWR AXIEM Analysis• AWR Software Options | <ul style="list-style-type: none">• AI Max NNA 110• Allegro Package Designer Plus• Allegro PCB Designer• Allegro PCB Librarian• Allegro Pulse• Allegro System Capture• Assura Physical Verification• AWR Design Environment Platform• AWR Tx-Line | |
| C <ul style="list-style-type: none">• Cadence Cerebrus Intelligent Chip Explorer• Celsius EC Solver• Certus Closure Solution• Clarity• Clarity 3D Transient Solver• Clarity PCB Extraction• Conformal Equivalence Checker• Conformal Overview• ConnX B10/B20• Controller for PCIe | <ul style="list-style-type: none">• Cadence Specman Elite• Celsius• Celsius PowerDC• Circuit Design• Clarity 3D Solver• Clarity Advanced IC Packaging Extraction• Compute Mesh Spacing for a Given Y- for Viscous Flow in CFD• Conformal Littmus• Conformal Smart LEC• ConnX DSPs• Controller for PCIe and CXL | <ul style="list-style-type: none">• Cadence Verification• Celsius Advanced PTI• Celsius Thermal Solver• Circuit Simulation• Clarity 3D Solver Cloud• Clarity Extraction• Clarity IC Packaging Extraction• Conformal Constraint Designer• Conformal ECO Designer• Conformal Low Power• ConnX 110/120• Controller for CXL• Custom Design Migration | |
| D <ul style="list-style-type: none">• DAM Media Carousel Test• Design IP• Virtuoso Digital Implementation• Virtuoso Integrated Physical Verification System• Virtuoso System Design Platform• Vision P1• Vision Q8• Voltus-XFi Custom Power Integrity Solution | <ul style="list-style-type: none">• DDR/LPDDR PHY and Controller• DDR5, DDR4, DDR3 PHY and Controller | <ul style="list-style-type: none">• Virtuoso Heterogeneous Integration• Virtuoso Layout Suite• Virtuoso Schematic Editor• Virtuoso Variation Option• Vision P6• Voltus IC Power Integrity Solution | <ul style="list-style-type: none">• Virtuoso InDesign DRC• Virtuoso RF Solution• Virtuoso Studio• Vision DSPs• Vision Q7• Voltus-Fi Custom Power Integrity Solution |
| X <ul style="list-style-type: none">• Xcellium Logic Simulator• Xtensa NX Processor Platform | <ul style="list-style-type: none">• xSPI | <ul style="list-style-type: none">• Xtensa LX Processor Platform | |

79 © Cadence Design Systems, Inc. All rights reserved.



There are many separate software tools required in the flow. All of these are grouped into platforms based on the stage where they are used in the design process. A single IC design and verification flow may not utilize all tools listed. We will explain these different tools and their purpose as the course progresses.

Expect to Specialize

No individual can be at an expert level in every tool and every language.

If one had time to learn everything, then before you finished, you would need to start over again.

- Like painting the Sydney Harbour Bridge.

For a career in the IC design verification industry:

- Expect to specialize in a maximum of 3-4 languages/tools at each stage of your career.
- Expect to take 2 years to get good at it.
- Expect to constantly renew your knowledge or face becoming obsolete.
 - After any 5-year window, technology and the how things are done has almost completely changed.



No individual can be at an expert level in every tool and every language. If one had time to learn everything, then before you finished, you would need to start over again. It's a bit like painting the Sydney Harbour Bridge. For a career in the IC design verification industry, expect to specialize in a maximum of 3-4 languages/tools at each stage of your career. Expect to take 2 years to get good at it. Expect to constantly renew your knowledge by expecting major changes every five years, along with frequent incremental changes.

Implementation Summary

- The implementation flow starts with RTL and progresses through a series of varying representations.
- There are many things to consider when coding RTL, both on the front end (from specification to microarchitecture) and the back end, through synthesis and place/route.
- As the design progresses through the flow, more accurate representations of the design are created, thus more accurate measures of timing, area, and power.
- RTL changes during this process to ensure that the design meets its overall requirements and specifications.
- Changes to the RTL very late in the process impact the schedule significantly.



The implementation flow starts with RTL and progresses through a series of varying steps.

There are many factors to consider when coding RTL. On the front end, factors include specification and microarchitecture. On the back end, factors include synthesis and place/route. As the design progresses through the flow, more accurate representations of the design are created, thus, more accurate measures of timing, area, and power are generated. The RTL changes during this process to ensure that the design meets its overall requirements and specifications. Changes to the RTL very late in the process impact the schedule significantly.

Test Your Understanding

True or False

1. Logic synthesis provides an accurate model of the timing paths in a digital design.
2. Static timing analysis is faster and more accurate than SPICE.
3. DFT issues can be a cause for recoding the RTL.
4. All RTL designers should create a floorplan to ensure their block meets its specifications and requirements.
5. Timing closure should not be attempted until the RTL has gone through significant examination through synthesis and floorplanning.



Test your understanding of the section by answering the following questions.

Summarizing So Far...

We have introduced all of the steps in the IC Design and Implementation flow:

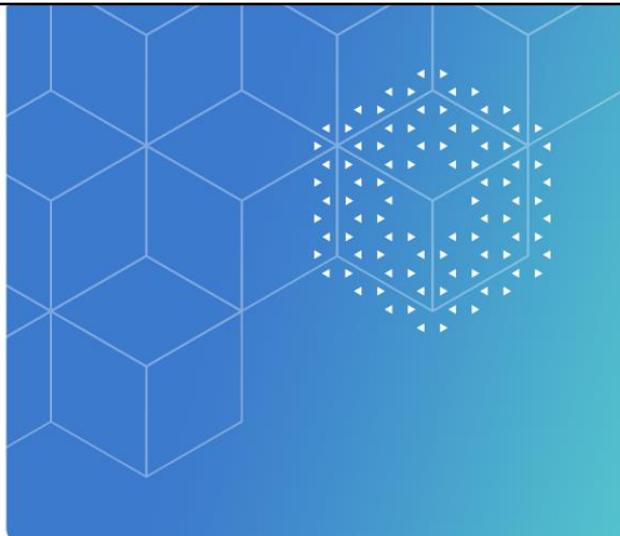
- Specification, microarchitecture, RTL, logic synthesis
- Floorplanning, placement, clock tree synthesis, route
- Extraction, delay calculation, static timing analysis, signal integrity
- Design optimization, physical synthesis
- Design verification, mask prep

Each step in the process, in and of itself, is very detailed, so we will spend the remainder of the course learning more about each step.

We have also explained how and why one has to gain expertise in only one of these streams, not in both, either *Design and/or Verification or Implementation*.



This page does not contain notes.



Submodule 2-1

The Scale of the Challenge

cadence®

In this section we talk about how big the challenge is for designing and verifying a digital I C.

In today's world now the scales are absolutely mind boggling.

So hopefully this presentation gives you some insight into how big the numbers really are, how difficult the challenge really is, and why things are designed and verified in the way they are.

This will hopefully give you some background into the industry and have a deeper understanding of all the other subjects that you talk about.

Submodule Objectives

In this submodule, you will:

- Describe the scale of the Integrated Circuit (IC) design, verification and manufacture challenge.
- Describe what affects the economics of IC design, verification and manufacture.
- Explain how the two points mentioned above dictate the design and verification flow.



So, what we're going to discuss in particular is the economics of digital ICs, because this really drives everything else in reality.

And as economics drive the design and verification, it also dictates the flow we have to go through for design and verification in order that, of course, at the end of the day, people make money.

That's what it's all about.

How to Describe the Challenge

To understand the IC design, verification and manufacture challenge, we need to understand the motivations.

What drives most human activity?



IC Design and manufacturing only become economically viable at huge scales.

Semiconductor sales are approximately \$500 billion US dollars per year.

We will now explain the scale of the whole process.

86 © Cadence Design Systems, Inc. All rights reserved.



The important thing is motivation. And, of course, what drives most human activity is money.

So that's what we have to do is follow the money. IC design and manufacturing only become viable at huge scales, absolutely massive scales.

These are the numbers I'll present to you. Starting with the annual sales figures for semiconductors are 500 billion US Dollars.

So that's a lot of money. In order to understand how come it's that much, we need to understand what the whole process is.

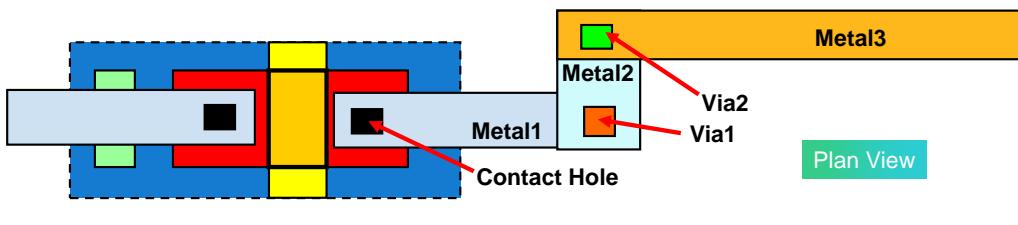
IC Physical Structure

Made of many layers.

- 10-20 layers in leading edge designs.

Silicon polygon shapes form transistors.

Interconnections made with metal tracks on multiple layers.



87 © Cadence Design Systems, Inc. All rights reserved.



The layers above the transistors are normally all metal interconnect.

IC is constructed at the very lowest level with a layer of silicon. These different colors indicate different doping of the silicon, and different materials.

And on top of that, you have a number of metal layers. So that number of layers can be anything up to 20 layers.

Different kinds of chips tend to be manufactured in very different ways.

For example, memory chips, and memory ICs are designed and constructed in a very different way to a CPU or a GPU graphics processor.

So, these polygons, these shapes on the silicon form, what's called transistors,

We're talking about what transistors are in a second, and the interconnects are made on multiple layers, and these layers are connected via something known as a Via.

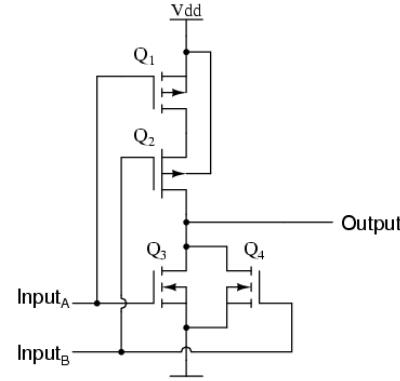
The Via we see here is looking down from the top, the Plan View.

What Is a Transistor?

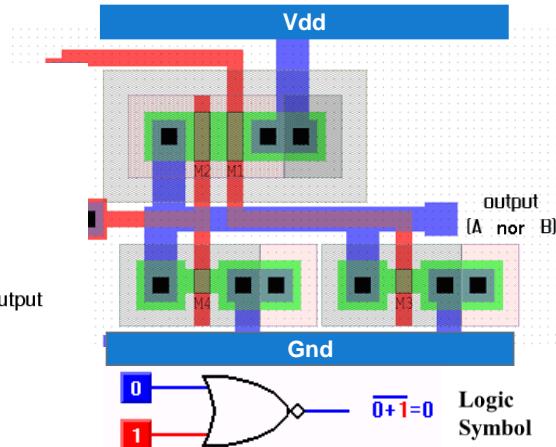
A transistor is a digital switch.

4 CMOS transistors are required to make a NOR gate.

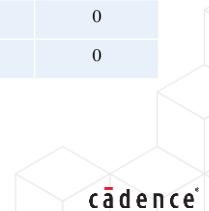
Other “Boolean” functions formed in a similar way.



88 © Cadence Design Systems, Inc. All rights reserved.



Truth Table		
A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0



CMOS has complementary, meaning N and P-doped, transistors.

A transistor is a digital switch. Q1, Q2, Q3 and Q4 are all transistors.

The schematic diagram shown here is known as CMOS complementary mos, and it's complementary because these come in pairs, a complementary pair.

There's an N-type and a P-type transistor.

The N-type in CMOS is with the arrow pointing in, and the P-type is with the arrow pointing out.

This forms a switch, or a logic gate, because if we apply a zero to both inputs here, what happens is we're turning both of these transistors off, and because they come in complementary pairs, it means if Q3 is off, Q1 is on, and if Q4 is off then Q2 is on.

In the case we have 0 0 0 here, these two transistors are off, which means these two are on, and we're connecting up to the power rail here.

So that's the logic. One, if any of these inputs here, A or B, has a one in it, then this transistor gets turned on, and its corresponding pair gets turned off, and we're connected to the ground zero.

So, what we've done with these four transistors here is form a what's known as a boolean gate, which follows this true table.

And if you were to observe a layout of this, an example layout, that's what it would look like.

The polygons on the silicon and these blue and red lines represent the metal layers, the connections, the logic symbol of a NOR gate is like this.

That's what transistor is, Just a switch. It goes on and off depending upon what voltage you put on the gate here. So, the gate is just bits with the arrows on there.

How Many Transistors Per Chip?

In leading-edge designs, 20-30 billion.

- That's approximately 4 for every person on Earth.

How big is a silicon chip (IC)?

- Typically, 180mm^2 approximately $13.4\text{mm} \times 13.4\text{mm}$.

Something the size of your thumbnail has 20 billion transistors and over 200 billion connections that need making.

89 © Cadence Design Systems, Inc. All rights reserved.



So how many of these transistors can you get in the chip? Well, unbelievably, up to 30 billion.

Think about that for a moment. That's more than four transistors for every person on the planet in something the size of your thumbnail.

So that's quite staggering. And if you've got that many transistors, you've got to connect them all with those metal tracks we saw.

You can imagine there might be something in the order of 200 billion connections that need to be made, all in something the size of your thumbnail.

That's quite incredible.

What Does “Process Node” Mean?

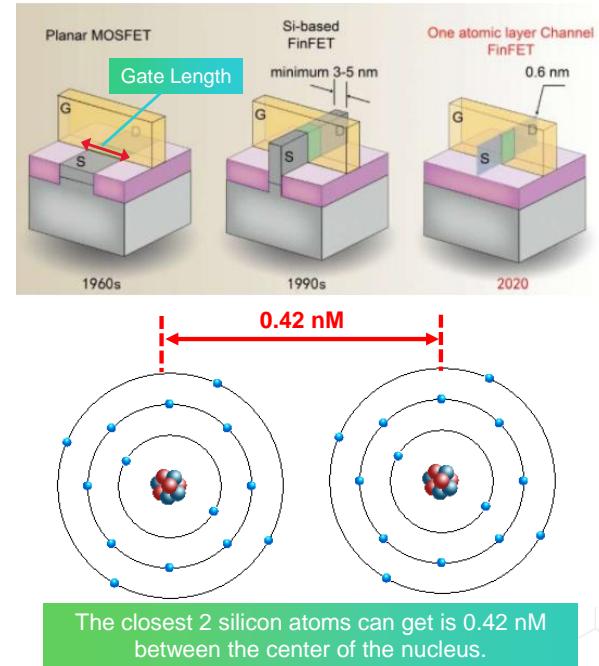
Represents the minimum size of physical features which can be manufactured on the IC.

It used to be a real physical measure of transistor gate length in nanometers.

- When transistors were planar – 2 dimensional.
- However, now things are not so straightforward.
- Transistors are built with different techniques using 3 dimensions.

One manufacturer's 10nM process node may be virtually the same as another's 7nM node.

- Process node has become a mere marketing term.
- A fairer comparison is transistor density per mm².
- For leading-edge designs, typically 100-200 million transistors per mm².



90 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Since September 2021, the smallest most advanced node in mass production is 7nM. Even the transistor density number doesn't tell the whole story. The stated density will not be achievable across the entire chip to routing requirements, low-power requirements, design rules, etc.

A term you often hear in digital IC manufacturers is process node. What does this actually mean? Well, it's one of these things that used to mean something, but now it's really a marketing term. So, it is used to represent a minimum-size feature that could be manufactured on an IC. In the days when transistors were planar, so two-dimensional, this actually meant the gate length.

The diagram shows the order of the 1960s and 1970s and 1980s. This was just how wide that gate was. That's what process node meant. Now, as time went on, the manufacturing techniques became more and more advanced, and transistors are now built in three dimensions in different manners. Here we can see two versions of a three-dimensional transistor now, and we can see that this gate length or the width of that gate is approaching kind of near minimums because, a, if you have two silicon atoms, the closest the nuclei can get to the center of them is 0.42 nanometers. So, this 0.6 is not much bigger than 0.4 to the maximum if the material uses silicon. Because you have different manufacturing techniques in three dimensions, it's hard now to compare what gate length actually means.

There are different ways of calculating this, which of course every manufacturer chooses the way that makes them look best. So, one manufacturer's ten-nanometer process node might be exactly the same as some other companies' seven-nanometer. Note The term really is a bit meaningless.

Now if you want a fairer comparison, then transistor density is what you could use. How many transistors do you have per square millimeter?

And for leading-edge designs in the current year 2023, typically that's 100 to 200 million transistors per square millimeter.

What Is Moore's Law?

It is not a law. It is an observation of what happened previously extrapolated into the future.

Moore's Law states that the transistor density of an IC doubles every 2 years.

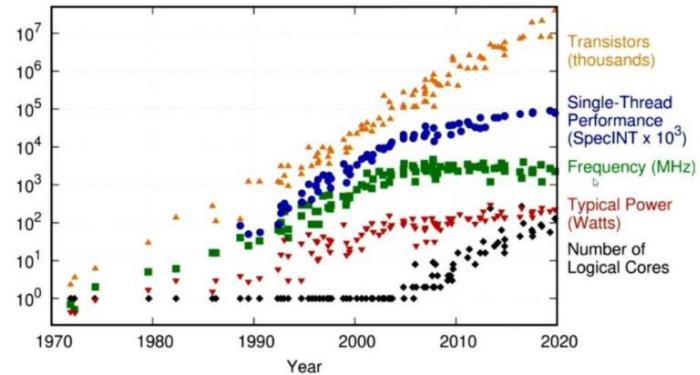
Its been remarkably accurate for 50 years.

Almost linear when plotted on a log graph.

It is unclear how long this will continue, due to physical limits being reached, for example:

- Layers are only a few atoms thick.
- Power density has plateaued.
- Frequency has plateaued.

However, all previous predictions of the end of Moore's law have been proved wrong.



91 © Cadence Design Systems, Inc. All rights reserved.



Now almost everyone has heard of Moore's Law, but what does it really mean?

Firstly, it's not really a law; it's just an observation that was made many years ago by someone called Doctor Moore, of course. And that observation says that approximately every two years, the transistor density doubles.

Basically, you get twice as many transistors in the same area every two years.

So, this is an exponential increase, and that's proved remarkably accurate over decades, 50 years.

This prediction was made all the way back in the seventies. As you can see, if you plot this, this is a log graph, of course, powers of ten going up the Y axis.

This is almost a straight line, which is quite incredible. Another thing this graph tells you is that maybe it's running out of steam; this law is because we can see the frequency is plateaued, and the power density has plateaued.

You can't run these chips any faster as time goes on, or you can't make them consume more power as well because physical limits are being approached.

Layers are only a few atoms thick, and power and frequency plateaued.

However, that's not to say someone will come up with some great idea because all previous predictions of the end of Moore's Law have proved incorrect.

It just keeps going on.

The Important Implication of Moore's Law

The price of putting the same transistor density on an IC halves every 2 years.

- Its exponential price reductions.

Every 20 years, the cost drops by a factor of over 1000.

Functionality which cost millions of US dollars in the 1980s, cost hundreds of US dollars today.



The important implication of that law is the price of putting the same transistor density on the IC halves every two years.

So exponential price reductions. So, every 20 years, the cost drops by a factor of over 1000. And functionality, which costs millions, tens of millions in the 1980s, only cost hundreds today. Before, what could the military and government only afford? Now you can just buy it in your house.

A good example of that would be an Xbox or a PlayStation, you know, flight simulators.

How Are ICs Made?

Manufactured on discs of silicon called wafers.

The small squares in the diagram are individual chips (also called dies).

Typically 300 chips per 300mm wafer.

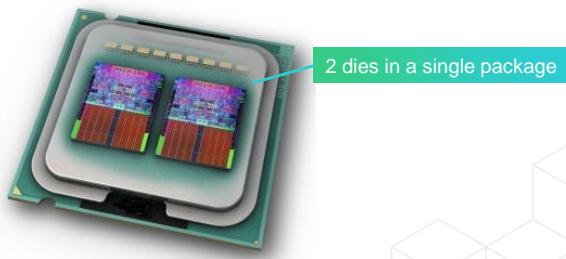
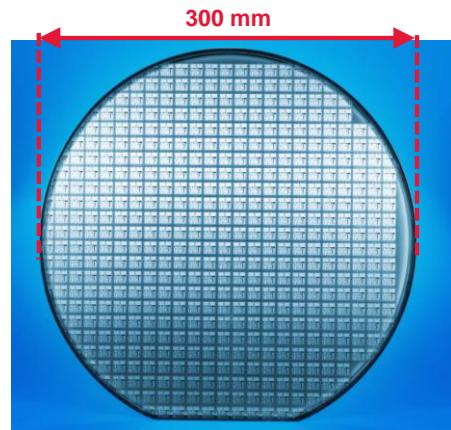
The retail cost of a typical high-end processor on the latest process node is typically \$300-500 US.

The retail cost of each wafer is approximately \$120,000 US dollars.

- Pure manufacturing cost approximately \$6,000 US dollars.

Multiple dies, 8, for example, might be stacked vertically and fitted into one plastic package.

Multiple dies may appear in the same plastic package.



93 © Cadence Design Systems, Inc. All rights reserved.

How are these IC made, then? They come in discs called wafers. This disc is cut out of a long cylinder.

This entire like cylinder would be made from one single crystal, and it's drawn out by rotating it into a big one.

It's a huge, heavy thing.

And it's chopped up into layers like this called wafers.

And what we can see on this diagram is the small squares we see here.

Those are actually individual chips, which are also known as dies.

Typically, on a 300-millimeter wafer. So most modern factories now use 300-millimeter wafers.

That's the diameter of the wafer. You get approximately 300 chips from that.

Now, if you're talking about a high-end processor and you're an individual buying this from your retailer, you're only buying one at a time.

You know, the price for one of these processors on these chips will be something in the order of 3 to 500 US dollars, which means on a wafer, the retail cost.

If you could sell these to individuals at that price, one wafer would be worth 120,000 US dollars.

And manufacturing costs are approximately 6,000 dollars per wafer. You might get these dies packaged up.

So, the black chips were familiar with seen with photographs of printed circuit boards that may contain multiple dies, which are either stacked on top of each other or placed side to side like this.

This shows two dies in a single package so that you might get as many as eight dies, for example, stopped stacked on top of each other.

Where Are Wafers Made?

In a factory called a Semiconductor Fabrication Plant.

- Commonly called a “fab” or a “foundry.”

These are huge plants; some are 1000 meters long or more.

They have highly automated handling and processing facilities.

A high-end fab using the latest process node costs \$10-17 billion US dollars.

It might take over 2 years from the start of building a fab to having reliable mass production.



cadence®

94 © Cadence Design Systems, Inc. All rights reserved.

Would you make these things? You make them in a joint factory known as a fab or a foundry. And these plants are huge. You know, this is a photograph of one of them. And there's something in the order of 1000 meters long, the whole campus.

It's a big factory, employing a lot of people. Although things are highly automated, all the handling is kind of automated in new factories now.

It still needs a lot of people to run this.

If you're building a new high-end fab and in 2022 where we are right now, you'll probably be aware that for the last year or so or more, there's been a silicon chip shortage, Meaning cars can't get manufactured, there's a shortage of Play Stations and laptops and all these kinds of things. You can't just put these things up overnight. It costs in the order of 10 to 17 billion, probably a bit more in today's money. And it takes a couple of years.

Even if you work on these plants, building them 24 hours a day, you know, it's going to be at least two years before you're in a position where you can reliably manufacture chips.

So, you can't just throw these things up overnight. Although there's a massive shortage at the minute, people are rushing to build these factories right now, and it's going to take them a lot of time and money to get these into a state where they can start producing silicon chips to alleviate the shortage.

How Many Wafers Need to Be Made?

Approximately 1.4 billion smartphones are sold every year.

- That's a lot of ICs and that's just smartphones.

Think of all of the other silicon chips required for large markets:

- Cars – Modern cars are estimated to have 40-50% of their cost due to electronics
- Cloud Data centers and Web services
- AI training
- Crypto currency mining
- Games consoles

..and so on

Each leading-edge fab manufactures 80,000 – 300,000+ wafers per month.

These fabs have to operate 24 hours every day of the year to be economically viable.



95 © Cadence Design Systems, Inc. All rights reserved.

How many wafers do we need to make?

Well, last year, 2021, approximately 1.4 billion smartphones were sold. You know, that's quite amazing, isn't it? One for every seven people on the planet just in one year. It's quite staggering. And that's just smartphones. What about all the other things like cars?

The cost of modern cars, something in the order of half the cost, is due to electronics. Cloud Services. Web services. Servers. AI training. Cryptocurrency, mining games consoles. It goes on and on and on. Virtually everything has got a silicon chip in it, from your microwave to your fridge to your television. So, these huge factories will turn out something in the order of tens of thousands to hundreds of thousands of wafers per month. And in order to be economical to get your 17 billion dollars back that it costs to build, you've got to run these things 24 hours a day, every day.

What Must One Do to Get an IC Manufactured?

Supply a fab with the final manufacturing database (GDS II file) for the IC.

- This is known as “taping out.”

The fab will not accept your file if you haven't performed functional and physical verification to their requirements.

- This is known as “signoff.”

The fab may modify your file to improve yield, manufacturability and testability.

- As inevitably there will be manufacturing errors, yield is the % of ICs that work correctly.

How long does it take from giving a fab your GDS II to getting it back the actual IC?

- For a leading-edge process, it typically takes 10-15 weeks to process a wafer.



What do you have to do in order to get this IC manufactured?

If you're designing a silicon chip, you give the fab a final manufacturing database known as a GDSII file. That's just a file format. And that process is known as taping out because it was used in the old days; it used to be done literally on magnetic tape, which is why it's retained that name. Now, you can't just give a file and hope for the best because they don't want to just dispute whether the chip was made correctly or not. You have to go through lots of functional and physical verification signoffs, as they're called, in order to ensure your chip actually will work.

When they manufacture it, and if it doesn't work, then it's the manufacturer's fault, not yours. They won't accept your file if you haven't done all these checks. They will probably stipulate to you what checks you must do. The fab may turn around and modify your file in order to improve yield. When you manufacture something, there will be manufacturing errors inevitably. And the yield means the percentage of the ICs that actually work correctly; they may, because they know their process, they may change your file to improve yield manufacturability and testability.

When these things come off the production line, the factory has to test them. As you can imagine, if you're doing hundreds of thousands a month, you want that test to take as little time as possible. So, if you give your file to this big factory, this fab, how long does it take to get back the actual IC, then in something you can place back on a circuit board? It takes something in the order of 10 to 15 weeks. There are no shortcuts here. That's how long it takes because it's an extremely complicated process.

Immoveable Deadlines

Billions of ICs need to be manufactured every year from 100s of different design companies.

Each fab processes 100,000s wafers per month and operates 24/7 all year round.

- It has to get back the \$10-17 billion US dollars it took to build and run costs to make a profit.

It takes 10-15 weeks to process a wafer.

This means scheduling is a massive issue.

You don't often get to dictate when your IC is manufactured.

- Even though you are a paying customer.

When you are given a date from the fab, you cannot miss it, or you will lose your place.

This means all design and verification must be done to the satisfaction of the fab by an immovable deadline.

- This is the dreaded tapeout.



What we've got then has got billions of ICs, and it needs to be manufactured every year to satisfy demand from hundreds of different design companies. Each fab processes hundreds of thousands of wafers a month, operating 24 seven all year round. And, it's got to get back the money it costs to build, of course, and all the running costs, which are considerable.

Imagine how much energy these places consume. It takes 10 to 15 weeks to process the entire wafer to get the final chip. There are so many things being manufactured by so many different companies. So, what this inevitably leads to is that there are very few companies who can actually create these fabs.

You get dictated to really when your IC is manufactured, and unless you're the biggest of big companies, even though you're paying, you'll get told, we need your file, your GDS 2 file, by this date; otherwise, you've missed your slot, and if you do, too bad, somebody else will fill it for you.

This means all of our design and verification has to be done by that immovable deadline. It's like launching a rocket to Mars. You can't do it anytime you like. You have to wait till all the planets line up. So, this is known as tapeout. And this is a high-pressure time for anyone doing design and verification.

The Price of Failure

What if your chip gets manufactured and doesn't work?

This is catastrophic.

- You have to fix the problem and verify the IC again – this takes time.
- You incur again non-recurring expenses like mask production – \$50-100 million US dollars for the leading-edge process.
- You have to get in the queue at the fab again – costly to jump the queue.
- The fab has to process the fixed design – 10-15 weeks.

Your market opportunity and/or profitability are likely to be gone.

- Everyone wants the latest smartphone, not the one before.

One mistake, out of the billions of things that could go wrong, could kill the entire project.

98 © Cadence Design Systems, Inc. All rights reserved.



What if you get your chip manufactured, and it doesn't work?

It's functionally incorrect. Not that it's been manufactured incorrectly; this is an absolute catastrophe. But imagine that you fix the problem. You've got to verify your fixes again, which will take a long time. You have to incur what should have been non-recurring expenses like mass production.

Mass production is producing their mask for lithography, so it's all extreme ultraviolet masks. These might cost anywhere up to 100 million a time. So that money is gone; once you've made that mask, you've lost your hundred million. And if it's wrong, that money is just gone in the bin effectively, and you've got to rejoin the queue when you've done all that.

Of course, you can pay to jump the queue, but that's going to be extremely costly. You've got to wait another 15 weeks potentially to get the chip back. By this time, it's highly likely your market opportunity and your profitability have gone. That's a disaster.

You can't turn up with your latest smartphone a year late or six months late because nobody wants an old smartphone. They want the newest one. So, all it takes is one mistake out of the billions and billions of things that could go wrong and destroy the entire project.

The Price of Development

It is often said that you need a village to design an IC.

Hiring hundreds of well-paid hardware engineers.

- Likely to be 10x that number of software engineers.

One needs many different software tools for hardware engineers.

- These are costly, tens of thousands or even hundreds of thousands of US dollars per single-user license.
- In the software development world, this is unheard of.

Massive compute requirements – server farms or cloud computing.

Development cost might be \$500 million to \$1 billion+ US dollars for a complex processor.



The cost of development, how much does it take to develop these things?

It's often said you need a village to design it. You need hundreds or even thousands of well-paid hardware engineers, and it's likely to be ten times that number of software engineers. You might need many different software tools for hardware engineers, and these are really costly, and software engineers aren't used to these kinds of costs. It might cost tens of thousands, hundreds of thousands for a single-user license for some tools. You know, that's amazing. Software developers need to get used to this. They're used to downloading free gardening tools or open-source tools. A single license for one application might cost you hundreds of thousands of dollars. Compute requirements are huge, so you need massive server farms or cloud compute environments. So development costs for a chip might be anything up to 1 billion dollars for complex processors.

The Cycle of Development

How long do we have to complete a design?

- Typically, a new version of a complex processor or smartphone is released every 12 months.
- This means that you only have 12 months before you need to have designed another one.
- This also means you can only sell the design for a year before it is obsolete.

A new process node becomes available every 2 years.

100 © Cadence Design Systems, Inc. All rights reserved.



How long do we have to complete a design?

Typically, a new version of a complex processor or a smartphone is released every 12 months or something of that order. So, if you look at that another way, it means you only have 12 months before you need the next one running.

It also gives a window of one year where you can actually sell the thing you've just designed. Additionally, a new process node becomes available every two years. We said know every two years, we can get twice as many transistors in the same area.

How Can We Meet Such Tight Time Tables?

Employ lots of engineers 100s – 1000s.

Staggered/parallel development of processors.

- Two separate teams work on new processor designs concurrently.
 - One team starts three months after the other, for example.
- “Around the world” teams working 24 hours per day.
- You must have already started the next design before the current design has come back from the fab.

Reuse as much of previous designs as possible.

Highly tuned and automated design, development and verification processes.

Extremely rigorous review process (signoff).

Only the companies good at IC design survive.

- One single failed IC could kill a medium-size company.



How do we reach these timescales? By employing lots of engineers. We're talking hundreds or thousands of them have staggered a parallel development of processes. You might have two separate teams working on new processor designs concurrently. One team is working on one processor, and three months later and a different team starts on the next one. So, they have this staggered development working around the world 24 by 7. This is a feature of today's world, and it doesn't matter really where you are. This pandemic has brought that into sharp focus.

It doesn't matter where you work, as long as you've got an Internet connection, people working around the clock. Basically, you already start the next design before the current designs come back from the fab you must have; otherwise, you're never going to finish.

This is just like an endless cycle that lasts 12 months. So most new designs really aren't new. They might be on a new technology, the next processor node, for example, but much of the functional content might be the same as the previous one. You need a development process that is highly tuned and, you know, doesn't allow errors to make verification easier.

A lot of time has been spent, decades have been spent developing these kinds of efficient development cycles and extremely rigorous signoff processes. Both manually and via verification tools. Only the companies which are good at IC design survive. It doesn't take many failed projects to kill any size company. A medium-sized company. One failed design. That's it. You've run out of money.

How Difficult Is the Design and Verification Problem?

Very difficult.

The technology advances really fast.

- A new process node is released every 2 years.
- Designs always get bigger and more complex because a new process node facilitates that.
- Software tools are continually updated to push the envelope of what's possible, or more economical, to the absolute limits.

Here is what you face, constantly:

- Your next design is always bigger and more complex.
- Likely using software tools that have never been used on a working physical ICs previously.
- Likely to be using a process node that has not been used before for a working physical IC.

...and BTW, there cannot be any bugs or your project is likely to be a catastrophic failure.

- Naturally, this could potentially be a massive financial loss.

102 © Cadence Design Systems, Inc. All rights reserved.



How hard is the design and verification problem? Well, it's really difficult.

Technology advances so fast that every two years, you've got a new process.

Node designs are always getting bigger and more complex because you can fit that functionality inside of a chip of the same size, twice as much as you could two years ago.

Software tools are continually updated to push the envelope of what's possible or more economical to the absolute limits all the time. You are working on the absolute limits of everything.

So, this is what you've got constantly. There's no time to relax. Your next design will always be bigger and more complex. You're likely to be using software tools that have never been used on a working chip before. You'll likely be using a process node that's never been used before. And by the way, if you've got any bugs, that's a catastrophe. It results in a potentially fatal financial loss.

Why Verification Is So Important

We previously stated:

- How difficult the problem is.
- How we literally cannot afford any bugs.

This means we have to invest massively in the verification effort.

A good analogy:

"It's like designing a passenger jet using technology that has never been used before. The first test flight it has is when it is full of passengers. "
– Paul McLellan, Cadence Design Systems

75% of the development cost will be verification.

It follows that you only spend 25% of development time designing the IC you actually want to create.

103 © Cadence Design Systems, Inc. All rights reserved.



Thanks and credit to Paul McLellan, Cadence Design Systems

This slide highlights why verification is so important.

We've said how difficult the problem is and how we can't afford any bugs.

There is no version two of hardware, so you have to invest massively in the verification effort.

And a good analogy from one of my colleagues, Paul McLellan.

It's like designing a passenger jet using today's technology that's never been used before because we're always pushing the leading edge. And the first test flight you get is when it's full of passengers. If you were in a position where you were designing a passenger jet and its first test flight was full of passengers, you can guarantee you'll be doing an awful lot of verification before you did that. Typically, you spend three-quarters of the time verifying your design, both functionally and physically. Which follows that you only spend 25 percent of the time actually designing the chip you wanted.

This seems a bit strange, but the reasons for it, we've explained, are all economic reality.

If It's So Hard, Then Why Do Companies Bother?

You already know the answer, right?



The scales are huge; we might be shipping 10s-100s of millions, even billions of one single chip.

We previously said:

- The retail cost of a single IC might be \$300 US dollars – naturally, this will be heavily discounted for bulk buys.
- We might get 300 chips from a 300mm wafer.
 - \$90,000 US dollars per wafer retail.
 - \$6,000 US dollars per wafer fabrication costs, \$20 US dollars per chip.

If we ship 100 million chips, even if we apply a 50% bulk buy discount, that's \$15 billion US dollars against a manufacturing cost of \$2 billion US dollars.

Suddenly, a development cost, even at the upper range of \$1 billion US dollars, still leaves a massive profit.

The risks are high, and the rewards are high.

We can only succeed if:

- Our chip works the first time; hence, we meet the delivery schedule.
- We get there with a better product before our competition does.

This emphasizes how absolutely critical verification is.

104 © Cadence Design Systems, Inc. All rights reserved.



If it's so hard, why does anyone bother with this?

Well, there's a lot of money in it. That's why the scales are huge. You might be shipping tens or hundreds of millions of a chip or even billions. I've been to places before where they shipped over a billion of one chip. I think it was a memory chip. Think of that. 1 billion is a huge number. If the retail cost of a single IC is 300 dollars, that's a lot of money is now around 300 billion dollars. Naturally, people are buying in bulk, so the price they pay is a lot lower. But if we get 300 chips from a 300-millimeter wafer, the retail sale at 50 percent of this 300 would be 90,000.

If the cost of manufacturing is uniform, we're talking 200 million chips, a 50 percent discount for bulk buying. That's 15 billion dollars versus the manufacturing cost of 2 billion dollars. So that's a huge profit, of course, isn't it? So even a development cost of 1 billion dollars still leaves a huge profit as long as we do it right the first time.

The risks are high, and the rewards are high. We only succeed if our chip works the first time, and we meet the delivery schedule, and we get there before our competition.

This is why verification is such a crucial part of the digital design cycle.

What Is Important to Optimize in an IC Design?

To maximize profits, there are 3 conflicting things that we need to trade off.

Power, Performance, and Area

- Abbreviated to PPA

One may also argue that with Internet of Things (IoT) and ever-increasing connectivity, that Security should be added to this list.



In order to maximize profits, there are three conflicting things that need to be traded off. This is known as PPA – power performance and area.

These are the things that affect costs now bring in. This has been around for decades, but one other thing that's become in recent times is security. This Internet of Things seems to have been on the horizon for many years but has not really arrived with the ever-increasing connectivity of everything.

It means that security is as important as all those other things.

PPA – Area

This is very straightforward.

To process a 300mm wafer has a fixed cost.

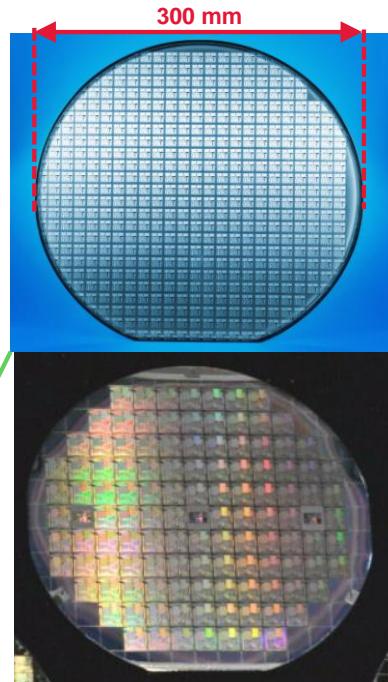
If we make the chip smaller, then we can fit more chips on a single wafer.

If we halve the size of the chip, we approximately halve the manufacturing cost.

Crucial to this are:

- The technology node we choose.
- Optimization of synthesis and optimization of place and route.
 - Namely, how good your EDA tools are.

Same price for each wafer, different cost per chip for each wafer



106 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The first one of those areas is the easiest to explain to process a 300-millimeter wafer that has a fixed cost. So, if we make the chip smaller. Each individual chip has a smaller area; then we can fit more chips on a single wafer.

This wafer has got a fixed cost.

We've got small chips, so we get lots of them on this one. The chips are a bit bigger; we don't get so many. So if we half the size of the chip, then we approximately halve the manufacturing cost. To us, because to process each wafer is the same, it is crucial the technology node we choose will optimize this. These are EDA tools that Cadence produces how good your tools are in minimizing the area of your chip.

PPA – Performance

How many operations can we perform per second, i.e., raw processing power?

System architecture, for example, caching, may have a significant effect.

The most significant variable is clock frequency.

- This is a trade-off – high frequency means high-power consumption.

As technology process nodes have advanced, frequency has plateaued at 2.5-3.5 GHz.

- Due to power density constraints.

Over time, performance has become less important, and power has become more important.

There are many more battery-powered devices than 10 years ago.

One of these heatsinks won't fit in your pocket



Your 13mm x 13mm IC is under here somewhere

107 © Cadence Design Systems, Inc. All rights reserved.



Performance means how many operations can we perform for? Second, that's the raw processing power. Many things affect this. So, for example, system architecture, like having a cache, for example, this has a significant effect. Probably everybody has heard of a cache and got some idea, at least what it does. That's that's part of the chip's architecture, which reduces how long it takes to access memory to get data. Therefore, you can perform more operations in a given time. The most significant variable is the clock frequency. The faster your clock it, the more operations you can do.

A downside is that the higher the frequency, the more power you consume.

So as technology loads of processed on the frequency have really plateaued for kind of server chips and ones where you plug it into the mains electricity, you know, they're stuck at something like three gigahertz for many years now. And that's all due to power density constraints. If you ever took the lid off a kind of old desktop or server or desktop, you find inside there one of these things. A heat sink under there is your like small chip, a 13-millimeter square with this big heatsink and fan on top of it to eliminate all the heat it produces.

So over time, performance has become less important, and power has become more important because there are just so many more battery-powered devices now than before. So you can't carry one of these in your pocket.

PPA – Power

Average Power consumption $P \approx C * V^2 * F$

- Clock frequency (F), Chip Voltage supply (V), transistors in chip (C).
 - Capacitance (C) is a function of the number of transistors and the interconnect length and proximity.
- A system running at 1 GHz consumes twice as much power as one running at 500 MHz.
- V^2 is the reason for the constant drive for lower supply voltages.
 - Typically for complex processors (I/O's 1.4v , core 0.9v).

The limiting factor in how much logic we can fit into a chip and the maximum frequency we can operate at is how much heat we can dissipate away from the chip.

Hence the reason for massive heatsinks and fans on, for example, server processors.

Clearly, we can't have giant heatsinks and fans on portable devices.

108 © Cadence Design Systems, Inc. All rights reserved.



Power is becoming increasingly important to power consumption. P is a multiple of these factors here. The C is the transistors on your chip. So, that's a function of how many transistors you have and all of the different interconnects. The metal interconnects, how close it is to each other, and its length to power supply voltage squared.

The frequency, the clock frequency. If you're running a system at one gigahertz, it's using twice as much power as the same system running at 500 megahertz. V squared because it's a square. That's the reason for a constant drive to get the lower supply voltage for the core.

For IOS, they're typically around about 1.4 volts on advanced kinds of technologies and complex chips, and the core voltage is probably 0.9 volts. The smaller you make that number, the less power you'll consume by a square factor. Small improvements here make big improvements to the power consumption, and the limits, in fact, are really on how much logic we can fit into a chip, and the maximum frequency we can operate at is how much heat can we get rid of.

So, that's the reason for that picture we saw previously of massive heatsinks and fans on, for example, servers processors. Clearly, for any portable device, you can't have a heatsink on it.

Why Is Power So Important?

The obvious reasons:

- Longer battery life for portable equipment.
- Less power is required for supplying the IC and the associated cooling requirements.
 - That's why you see data centers [re-]located in Iceland.

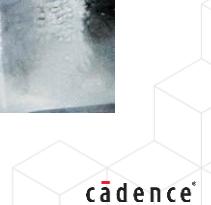
The not-so-obvious reasons:

- The power density has plateaued for advanced process nodes.
- You cannot get rid of the heat quickly enough from a chip made from silicon.
 - Not even with giant heatsinks and fans.

Liquid nitrogen is rather inconvenient



109 © Cadence Design Systems, Inc. All rights reserved.



Why is power so important?

The obvious reasons which are longer battery life for portable equipment or if you've got some kind of data center that's a huge kind of electricity bill for you and because mainly for the cooling or also for running the processor. That's why you see many data centers located in Iceland and northern Canada. The not-so-obvious reasons are that the power density is plateaued for advanced process nodes, which are based on silicon.

For silicon, which is what we've been using since the chip industry started, you can't get rid of the heat quick enough without using new materials. That will take many years to perfect that kind of technology because silicon has been used for so long that the actual processing has been actually perfected for that particular material.

If you want to change materials, there is no single one material, which is much better than any other proposition. It will take a long time for something to dominate and then the process to be made efficient. It doesn't matter what you do; even if you stick a giant heatsink on and you pour in liquid nitrogen over this thing, you still can't clock it any faster without damaging the chip somehow.

Thermal Problems

At the 7 nM node, only 50% of the transistors on a complex processor chip can be active at the same time.

- Otherwise, it overheats and fails.
 - The built-in thermal protection kicks in, chip melts, or the solder melts, and it slides across the PCB on which it's mounted.

We could create ICs with faster clocks, stack many dies on top of each other, squeeze in as many transistors as the technology allows, but can we get rid of the heat and not kill the battery in minutes?

This has to be considered when evaluating and comparing transistor density figures.

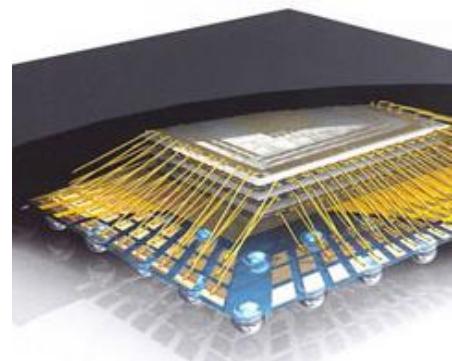
This greatly complicates the design and verification.

We have to ensure that each part of the design is active only at the given time it is needed.

We have to verify all of these different power scenarios.

We even need a new language named UPF (IEEE 1801) to specify the power intent and optimization.

We will discuss low-power design and UPF in the forthcoming lectures.



110 © Cadence Design Systems, Inc. All rights reserved.

The thermal aspect leads to several undesired consequences. At the seven-nanometer node, only half of the transistors on a complex processor can be active at any given time. You can't just clock every transistor at the same time. Otherwise, it will overheat and fail. This has to be considered as well when evaluating transistor density.

That's not the end of the story; it means you can actually clock all the transistors at the same time. So, chips normally have built-in thermal protection. If that fails on, the chip will melt or the solid. The chip gets so hot the solder slides off the board. These thermal problems bring greater complexity to the design and verification.

Now we have to be aware of all these different things. How are we going to get rid of the heat? Will it damage the package and so on? Will it damage the underlying structure of the IC and the high currents that are carried? We have to ensure that each part of the design is only active when it needs to be when it's essential that it must be.

As we've seen, we can stack many different chips on top of each other, and we can clock it as fast as you like and fit all these transistors into a small area.

But the underlying restriction is, really, can we get rid of the heat that we produce? If it produces a lot of heat, then it will either destroy it or it will kill the battery anyway in a too short time to be useful. What we have to do is verify all of these power scenarios and there is a different language called UPF, Unified Power format, which is not really standard, which specifies how we architect the power system, the power architecture, and optimize it. We'll discuss that in a separate lecture session. This is another thing we have to do as well as our functional design. We have to consider all these different power techniques.

How Can We Verify Something This Complex?

How big is the state space? An example of scale considers the following.

One commonly accepted estimate for the number of atoms in the universe is 10^{80} .

This count can be represented using 272 flip-flops, namely 2^{272} .

- With a 3 GHz clock, it would take approximately 6×10^{146} years to observe every value of a 272-bit counter.
- It takes 194 years to observe every value of a “tiny” 64-bit counter.

A complex processor design would have at least 4,000 times that many flip-flops.

- For example, $2^{1,000,000}$ states.

The state space is absolutely massive.

It's unfeasible to think one can simulate and verify every possible scenario over time.

- Using a software simulation tool can be 100,000 times slower than “real life.”
 - For example, it takes 27 hours to simulate 1 second of real-time.

We need a great deal of ingenuity and creativity to bridge that gap.



The question being asked here is how can we verify something which is so complicated. The first step along that road of evaluating how complex something is and how big is the state space.

To give an example of scale, consider that a widely accepted estimate for the number of atoms in the universe is ten to 10^{18} . If we had to count that using a digital counter, then we would need 272 flip-flops. So, 272 flip-flops give us 2 to the 7 to 7 to different permutations of state for that counter, so that different values it could take.

Now, if we had such a circuit with a three gigahertz clock, which is one of the fastest speeds you'll find on a digital IC, it would still take an unfeasibly long time to observe that even in real hardware. We're not talking about software simulators here. We're talking about many times the age of the universe to observe every value of that 172-bit counter. Even if you only have a tiny 64-bit counter, it will still take you nearly 200 years to observe every value with a clock running at three gigahertz in the real hardware. So you can imagine a complex processor would have at least 4000 times that many flip-flops, 4000 times, or 72.

For example, if it had a million, a million is a really high figure for the number of flip-flops in the chip. But even if it was something less like 500,000. Imagine how many states that is. The state space is absolutely huge. This is a crucial thing to understand. It's feasible that you can simulate and verify every scenario over time. Compound with that, these figures given here were for using a three gigahertz clock.

You can't achieve that with a software simulation tool. So, software simulation tools can be anything like 1000 times slower than real life. If you wanted to simulate a second of real-time, it takes you more than a day. You can imagine how long your laptops take to boot up, even if you've got a solid-state drive. So, in order to bridge that huge gap, we need a great deal of ingenuity and creativity, Which is why it can't be automated to as high a degree as one would wish.

Functional Verification Goal States

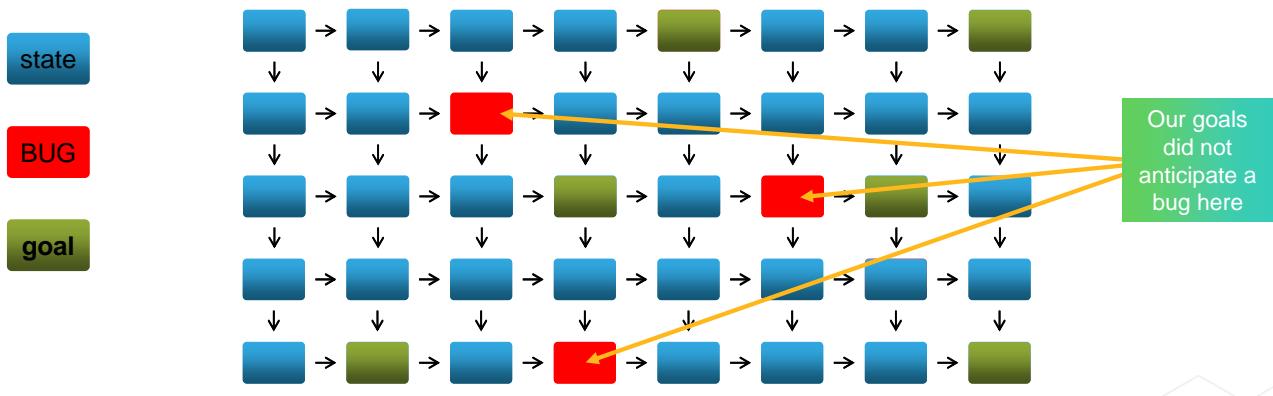
The verification goal is to reveal the bugs.

To reveal bugs, the verification team establishes “goal” states to visit.

We do this based on analyzing the specification, identifying key features, and team experience.

However, we must not limit verification to only the *known* goals! Our goals might be wrong.

Likely to create stimulus randomly following some constraints.



112 © Cadence Design Systems, Inc. All rights reserved.



The verification goal is to reveal the bugs. To reveal bugs, the verification team establishes “goal” states to visit. It determines these states based on analyzing the specification, identifying key features, and applying team experience.

Here, the team has selected some goal states to visit.

If you can imagine all of these different states we have in our design, some of those states will reveal a bug. The ones call it in red. And of course, you don't know where they are because otherwise you just go and fix them.

If verification is aimed at revealing these bugs, then how can we do this?

Well, what we do in the verification team is we accept we can't visit every state; we establish some goal states to visit. These are the ones in green here. These represent the important functionality of our design and the expected functionality of our design. And we do this based on the functional spec, what the key features are, and the experience of the team. Now the problem is, of course, that we're limiting ourselves here. If we only rely upon visiting these goal states bugs which are not at the goal states, we never observe them. If we never require to go through the states to reach the goal states, then we don't observe the bug because we didn't think we'd find one.

So many simulation techniques involve creating stimulus randomly, following some set of rules, which are called constraints.

Scale of Functional Verification

We have to perform 10,000 separate simulation tests in parallel.

- Each test using a different random stimuli.
- Massive compute parallelism, which has to be managed.

Then we must collate the results.

Then we need to be able to identify what designing functionality we actually exercised in each test.

- Did we test all of the design features we said we would?

We need to identify which tests are most efficient at covering our known goals in the shortest time.

When we find bugs, we iterate again using the most efficient tests.

All of this just ends up with functionally correct RTL code.

We haven't even started the physical implementation and verification yet!

Verification is risk management.

- We can't test every possible scenario the design is expected to work under.



Typically for simulation on a given digital design, you're performing tens of thousands of different simulations. Each test will be using a different random seed. When generating the stimulus randomly, a different seed will likely give you a different set of stimuli. And you can imagine how much computation that is; you need tens of thousands of different runs. There's huge parallelism in this, which must be managed because we need to collate the results of all those simulations. And accumulate and aggregate them in order to understand whether we've tested all of the things we should have. Furthermore, if we find bugs, we need to know which tests to rerun in order to verify that we fix the problem.

We need the tests that give us the most information and the least time.

We do this just to end up with functionally correct code. RTL code registers transfer level, which is the level of code you have to write out to design chips with the current technologies. We haven't even started anything else like all the physical aspects of this.

So, with physical implementation, you have to be clear about all kinds of things like electron migration, heat dissipation, and quantum effects. All of these different kinds of things. Verification really is risk management. Verification is not testing every possible scenario; it's managing the risk.

Can We Automate This Process?

Where is AI when you need it?

In principle, it could be automated.

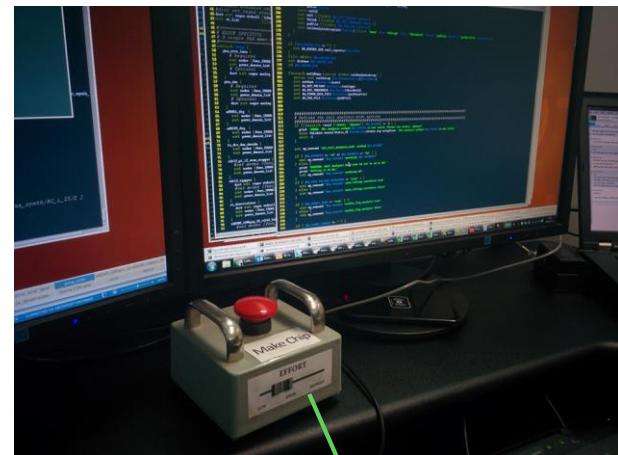
However:

- It would not be as efficient (profitable) on PPA.
- It would require you to foresee and define every possible problem in a hugely complex design.
- Likely that, automation introduces an unpredicted bug.
- Unfortunately, failure is not an option.
 - There is no version 2.

EDA stands for Electronic Design Automation.

The automation deals with the scale of the computation required, collating results, managing verification runs, parallel processing of place and route, parallel processing of physical design rule checks, and so on.

There is nothing even close to human creativity and innovation.



The mythical and magical “Make Chip” button

114 © Cadence Design Systems, Inc. All rights reserved.



If you read the media, you might think, well, I can solve everything in the world. And, of course, no, it doesn't. There's no I that can iron your shirt for you. In principle, this whole EDA process, the digital design, could be automated, but in real life, it wouldn't be as efficient on PPA, therefore not as profitable. It would require that it could foresee every single problem you have in a hugely complex design. The automation will likely not be perfect, so you'll introduce unpredictable bugs. And as we've seen for digital IC design, you can't just go and create a version, too like you would do with software. Failure is probably fatal.

EDA stands for electronic design automation, and automation is dealing with the scale of the computation requirements. Collating results and managing the verification runs parallel processing in terms of all the physical aspects, design, rule checks, etc. And the foundry signed off checks. So that's where the automation comes in. Nothing comes anywhere near close to human creativity and innovation.

There is no magic button. You can write some code, press this button, and get a chip out of the other end. It doesn't exist, and it's not likely to in anyone who's reading this lifetime.

Submodule Summary

Design and verification of Digital ICs are extremely difficult and complex.

It takes many different expert specialists to go from concept to a physical working IC.

- No individual knows every step in detail.

The rewards are high; the risk is high.

Profitability hinges on the correctness and completeness of verification.

Functional correctness is not sufficient.

The design needs to be optimized for Power, Performance, and Area.

The time schedules are very challenging.

There is no replacement close to the creativity and ingenuity of hundreds of engineers.



To summarize, we've seen the design and verification of digital ICs are very difficult and complex. It takes many different types of expertise to go from a concept to a physical IC. Not everyone knows every step in detail. Because every kind of discipline, every kind of specialization is a niche. No one knows how to put everything together in one go. The rewards are high, and the risk is high as well. And profitability hinges on the correctness of the design and completeness of verification.

Getting it right the first time, so just meaning it's functionally correct, is not good enough. The design needs to be optimized for power, performance, area and security.

Another important aspect is described.

Time schedules are very challenging. If you can't do it, someone else will do it faster than you and get all the money.

There's no replacement for human creativity and ingenuity.

Module Summary

In this submodule, you:

- Drew a flow diagram of the entire design flow.
- Identified the distinction between Digital IC design, verification, and implementation.
- Recognized the different stages of front-end design and verification.
- Recognized the different stages of design implementation.
- Identified and appreciated the challenges of scaling, costs, physical attributes, as well as low power and area constraints before tapeout.
- Identified the different processes in the semiconductor industry used to handle the above realistic challenges.



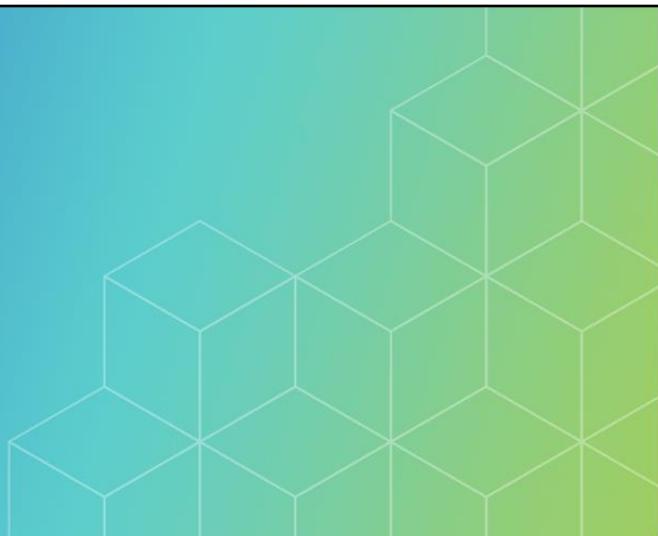
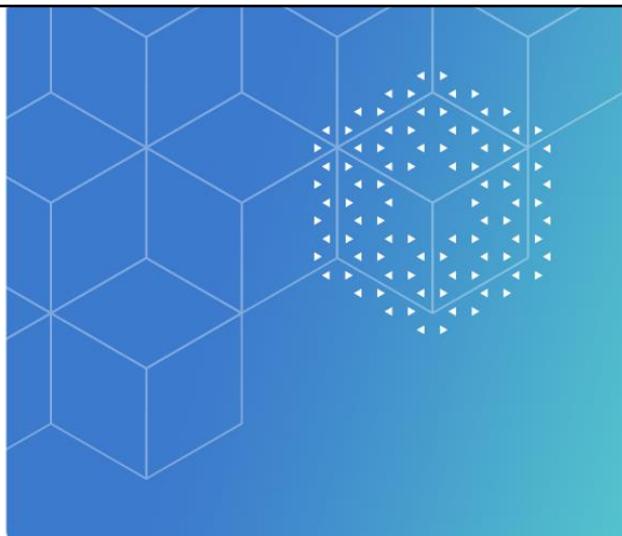
In this module, what we did is we showed an overall flow diagram of the entire design flow. We identified the distinction between digital IC design, verification, and physical implementation.

We recognized the different stages of front-end design and verification and the different stages of design implementation. We have some idea of the scale of the challenge and the economics of it.

Physical attributes give you a scale of how big the problem is, as well as other aspects like low power.

And we identified the different processors used in the semiconductor industry for handling all of these challenges because, of course, they are being managed.

Because chips are still getting made, so can now move on to the next section.



Module 3

Digital IC Functional Design

cadence®

This page does not contain notes.

Module Objectives

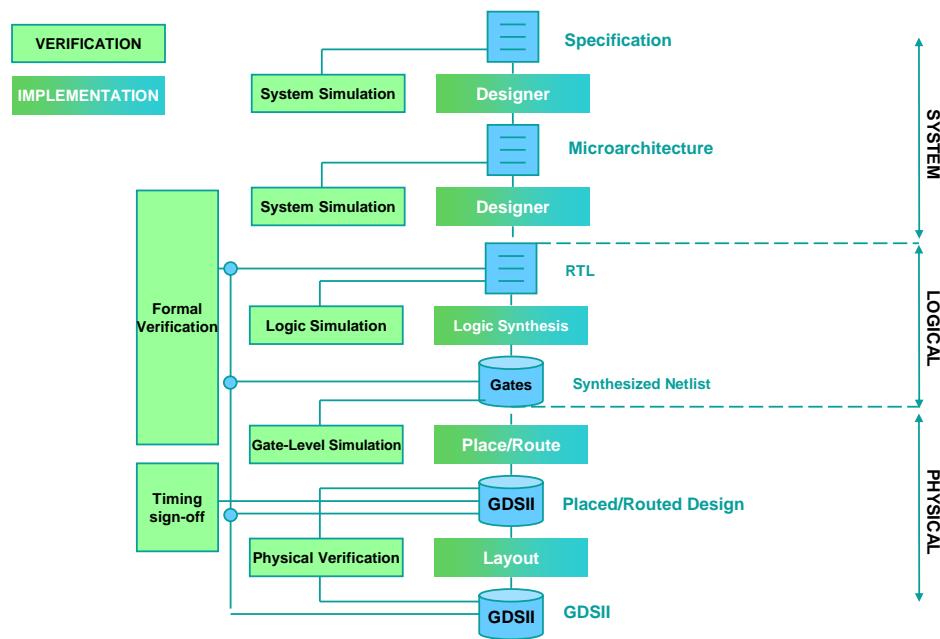
In this module, you will:

- Identify the basic design flow of spec input, coding, and output to synthesis.
- Explain the HDL history.
- Recognize why we were still using RTL and identify the levels of abstraction.
- Identify register versus combinational logic with examples.
- Test their memory on the memory state, latch FF memory cell.
- Define the setup, hold, and propagation delay.
- Identify and appreciate the different realistic design challenges such as low-power, size and area minimization, clock domains and crossing, etc.



This page does not contain notes.

Complete Basic IC Design Flow



119 © Cadence Design Systems, Inc. All rights reserved.

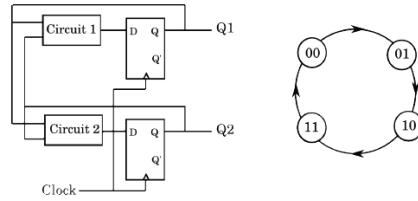


Discuss briefly the three phases of design, including system, logical, and physical.

Example: Design Spec of 2-Bit Synchronous Binary Counter

- The binary counter is an example of a simple synchronous digital circuit. It has no data inputs and no combinational output logic circuit.
- At each clock pulse, the counter takes up a new state and thus goes through a specific count sequence.
- We shall now write a design spec for a binary up-counter with two outputs that go through the following sequence:
 - 00 ->01 -> 10 ->11 -> 00 -> etc.
- The counter should have the following inputs:
 - rst*, which is synchronous, active low
 - clk*
- The counter should have the following output:
 - count (1:0)

The block diagram, structure, and state transition diagram of a 2-bit binary counter (built with two D-type flip-flops) is of the following form:



The truth table for the circuit is as follows:

Inputs tn		Outputs tn+1	
Q1	Q2	D1	D2
0	0	0	1
0	1	1	0
1	0	1	1
1	1	0	0

120 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

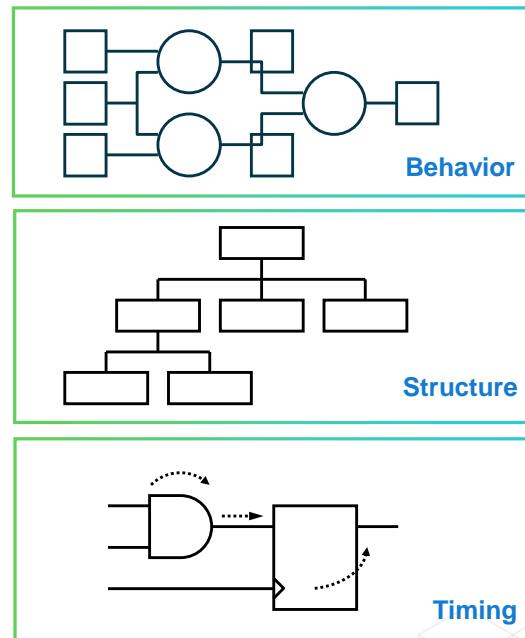
What Is Hardware Description Language (HDL)?

HDL is a programming language with special constructs for modeling hardware, concurrency, and timing.

HDL supports designs at multiple levels of abstraction:

- Behavioral modeling
 - Sequential behavior
 - Parallel behavior
- Structural modeling
 - Hardware component hierarchy
 - Software subroutine hierarchy
- Time modeling
 - An HDL has to model propagation delays, clock periods, and timing checks

HDL typically supports the simulation of estimated design timing.



121 © Cadence Design Systems, Inc. All rights reserved.

cadence®

An HDL is similar to a procedural programming language but also contains constructs to describe digital electronic systems. An HDL contains features and constructs to support descriptions of the following:

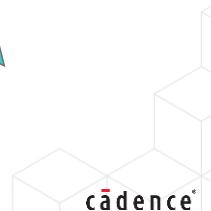
- Behavior – Both serial and parallel. In serial behavior, you pass the output of one functional block to the input of another, which is similar to the behavior of a conventional software language. However, in parallel behavior, you can pass a block output to the inputs of a number of blocks acting in parallel where many separate events happen at the same moment in time.
- Structure – Both physical, such as hierarchical block diagrams and component netlists, and software, such as subroutines. This allows you to describe large, complex systems and manage their complexity.
- Time – Programming languages have no concept of time. An HDL has to model propagation delays, clock periods, and timing checks.

An HDL typically supports multiple abstraction levels. You can describe hardware behaviorally, without and with sufficient detail for logic synthesis, and as a structured netlist of predefined components that can themselves be as simple as a transistor and as complex as another behavioral design.

Why Use a Hardware Description Language?

The benefits of using an HDL include the following:

- You write HDL in plain old ASCII text:
 - Capture the design quickly and easily manage modifications.
- You can design at a higher level of abstraction:
 - Easily explore design alternatives.
 - Find problems earlier in the design cycle.
- Use of a standard HDL enables design reuse:
 - Between design teams and partners and through the design flow.
 - No re-entry, reformatting, or translation.
- The description is independent of the implementation:
 - You can delay the choice of implementation technology.
 - You can more easily make architectural and functional changes.
 - You can more easily adapt the design to future projects.



122 © Cadence Design Systems, Inc. All rights reserved.

The benefits of using an HDL include the following:

- You can capture and modify the design more quickly in an HDL than by schematic capture.
- Earlier design capture at the higher abstraction level means earlier simulation, which facilitates design alternative exploration to produce a more optimal architecture and partitioning and allows a more thorough verification.
- Use of a standard HDL facilitates the reuse of designs from previous projects or from commercial Intellectual Property (IP) providers. You can move or switch between different tools and vendors without re-entry, reformatting, or translation of the design description.
- Your RT-level implementation is almost 100% independent of the implementation technology, so you can defer selection of the target ASIC or FPGA technology until the design is mainly entered, and you can switch technology or vendor with a minimum of a redesign.

Before and After HDLs

Before the Verilog, VHDL or SystemVerilog languages were invented, designs were:

- Small and much less complex.
- Captured using schematics.
- Simulated with SPICE*.

With the advent of these HDL languages, designs are:

- Much larger and more complex.
- Designed using software.
- Simulated with a Verilog/SystemVerilog or VHDL simulator.



**Simulation Program with Integrated Circuit Emphasis*

An HDL Supports Multiple Levels of Abstraction

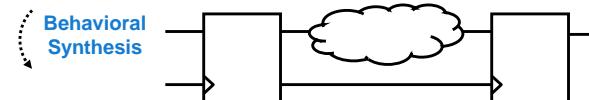
Behavioral level

- Algorithms



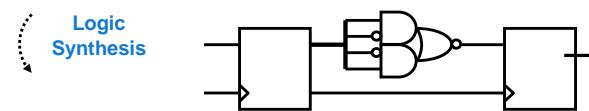
Register Transfer Level (RTL)

- Nets and registers



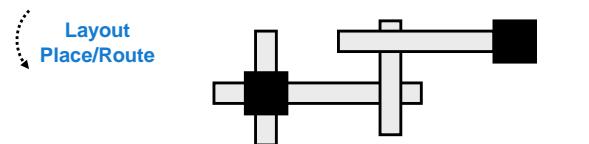
Gate level

- Built-in and user-defined primitives



Switch level

- Built-in switch primitives



124 © Cadence Design Systems, Inc. All rights reserved.

cadence®

You can describe a system as a group of hierarchical models of varying amounts of detail.

An HDL supports multiple levels of such detail. The three main levels of abstraction are listed and described below:

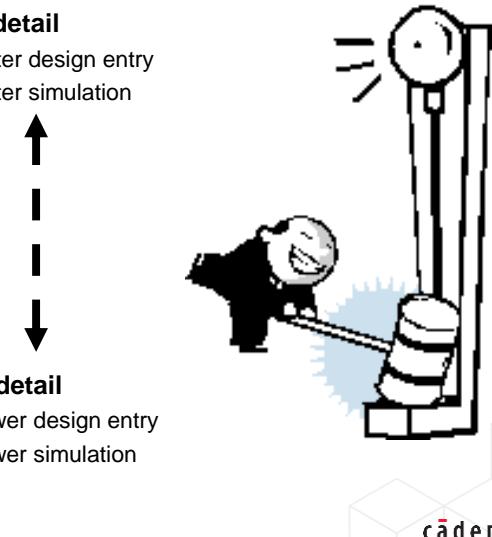
- The behavioral level:
 - You describe the system using mathematical equations.
 - You can omit timing – the system may simulate in zero time like a software program.
- The Register Transfer Level (RTL):
 - You partition the system into combinational and sequential logic, using constructs and coding styles supported by logic synthesis.
 - You define timing in terms of cycles based on one or more defined clock(s).
- The structural level:
 - You instantiate and interconnect predefined components.
 - Can include vendor-provided macrocells.
 - Can include logic primitives built into the language.

Abstraction Level Tradeoffs

Simulation effort is proportional to detail:

- Behavioral level
 - Algorithms
- Register Transfer Level (RTL)
 - Nets and registers
- Gate level
 - Built-in and user-defined primitives
- Switch level
 - Built-in switch primitives

- **Less detail**
 - Faster design entry
 - Faster simulation
- **More detail**
 - Slower design entry
 - Slower simulation



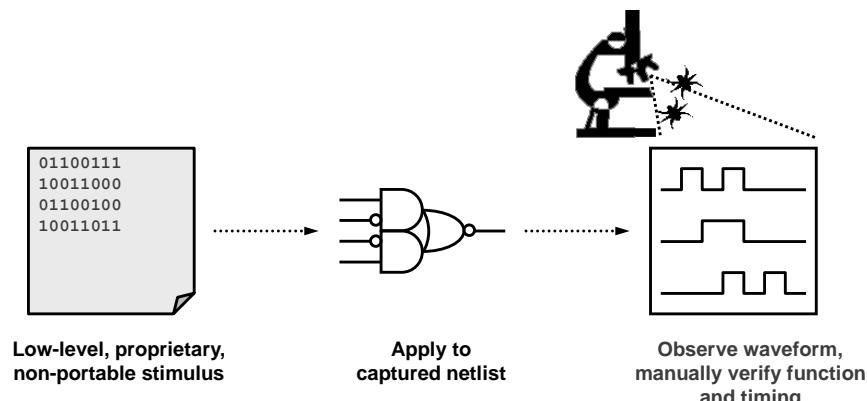
125 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Users of an HDL model at different levels of abstraction:

- Block architects model functionality at the behavioral level for maximum simulation speed and for the ease with which they can quickly modify the architecture as they explore alternatives.
- Block implementers refine the functional blocks to the RT level for a logic synthesis tool.
- Library developers model cells at the gate level for simulation and the physical level for place and route tools.

Legacy Schematic-Based Design



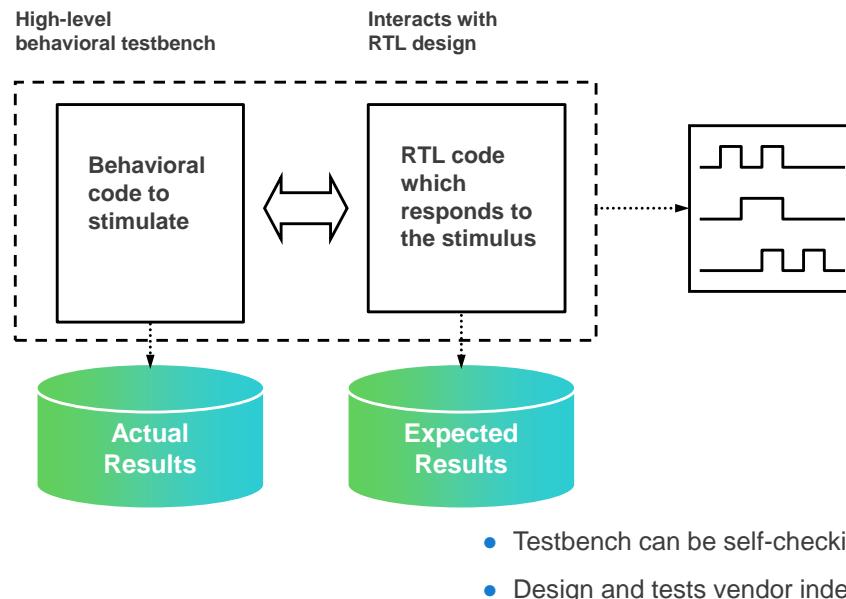
Let us compare to an HDL ...



126 © Cadence Design Systems, Inc. All rights reserved.

Designers used to capture the design intent using a proprietary schematic capture tool, manually developed the test stimulus in a proprietary tabular format, and simulated the design using a proprietary simulator. Those who had money to burn had graphical displays to examine the results, but most of us had plain old text terminals so that we could examine the results only with 0 and 1 characters. We did not use the Z and X characters because those values do not exist in hardware, and we used simulation only to generate expected results for a device test machine.

HDL-Based Simulation



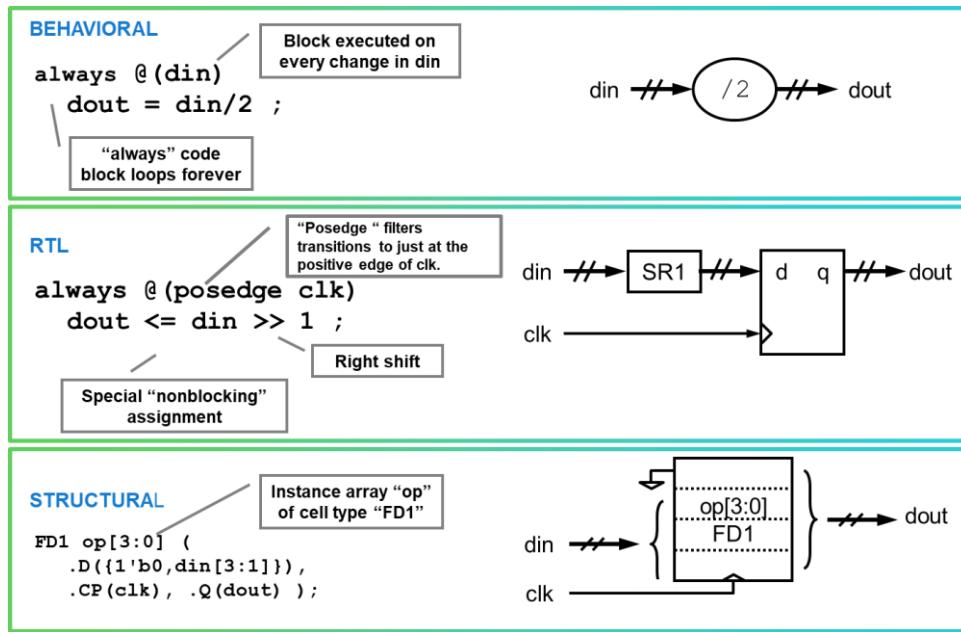
- Testbench can be self-checking
- Design and tests vendor independent

127 © Cadence Design Systems, Inc. All rights reserved.



In more recent times, we capture the design intent and its test using the same standard HDL and simulate them together using a choice of tools running on a wide choice of third-party platforms. We have high-definition, wide-screen displays that we do not have to stare at quite so much because our tests, to a large extent, pinpoint any problems with a high degree of accuracy.

Abstraction Level Example: Divide by 2 Operation



128 © Cadence Design Systems, Inc. All rights reserved.



These code fragments illustrate the three main levels of abstraction:

- The behavioral level of abstraction describes the design behavior with no hint about how the operation is implemented.
- The RT level of abstraction describes the design behavior with sufficient detail that logic synthesis can infer an implementation involving an edge-triggered storage device.
- The structural level of abstraction instantiates and connects a predefined storage device from a macro library, not caring how the component is itself implemented.

What Is RTL Coding?

Converting the design specification set of rules into a high-level design using a Hardware Description Language such as Verilog, VHDL, SystemVerilog, SystemC, etc., is called RTL Coding.

- RTL is an acronym for “Register Transfer Level.”
- Instead of using schematic capture like in the old days to design a complex chip, designers now use an HDL.

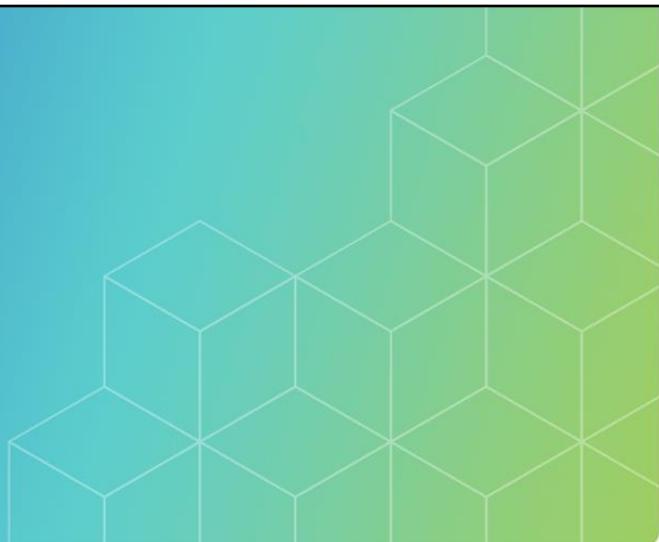
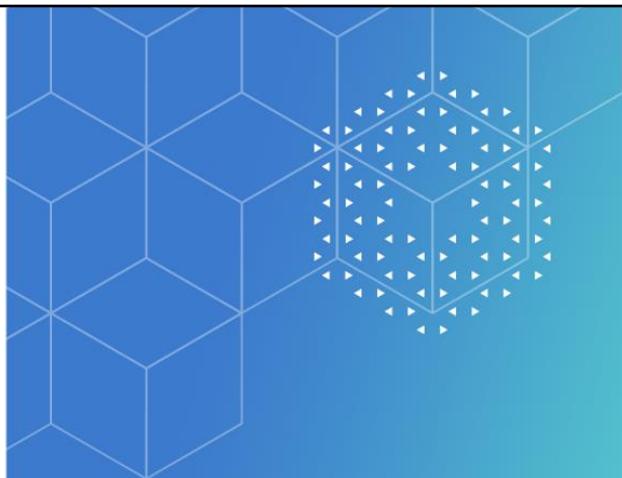
```
module mux (output logic y,
             input logic s,
             input logic i1,
             input logic i0);

    always_comb
        if (s==1)
            y = i1;
        else
            y = i0;

endmodule
```



This page does not contain notes.



Submodule 3-1

Digital IC Functional Design: Hardware Implementation

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

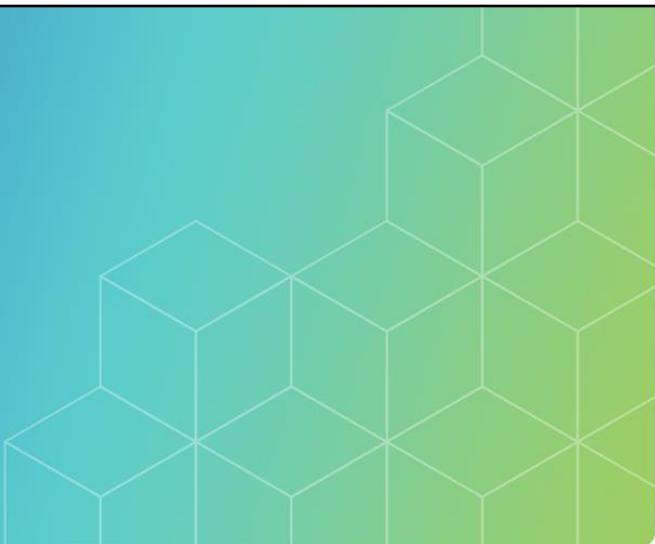
- Explore hardware implementation concepts.

Topics include:

- Introduction
- Data representation
- Combinatorial building blocks
- Clocked building blocks and synchronous design
- Arithmetic building blocks
- Finite state machines
- Memory structures



In this submodule, we will explore basic hardware implementation concepts, starting with how data is represented in hardware. Then we will look at both combinational and clocked building blocks and define some of the basic concepts of synchronous, clocked design. Then we will look at more complex hardware implementations for arithmetic, state machine, and memory structures.



Submodule 3-1-1

Data Representation

cadence®

This page does not contain notes.

Submodule Objectives

In this submodule, you will:

- Look at the way in which data is represented in binary.

Topics include:

- Unsigned numbers
- Signed numbers
- Other signed data formats
- Floating point representations
- Binary coded decimal
- Gray code



In this submodule, we look at the hardware representation of data in binary, including signed and unsigned numbers, floating point representations, and some selected, application-specific representations in Binary Coded Decimal and Gray Code

Binary Representation

Many more digits are required for binary.

Hexadecimal is more convenient to describe binary numbers:

- Each hex digit is 4 binary bits
- Decimal digits for 0-9
- Letters A-F for 10-15

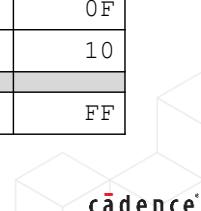
Specific binary widths are named:

Bits	Name
4	nibble, half-byte
8	byte
16	word*

*or data width of a processor, e.g., 32 or 64-bits

Decimal	Binary								Hex
	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
	128	64	32	16	8	4	2	0	
0	0	0	0	0	0	0	0	0	00
1	0	0	0	0	0	0	0	1	01
2	0	0	0	0	0	0	1	1	02
9	0	0	0	0	1	0	0	1	09
10	0	0	0	0	1	0	1	0	0A
11	0	0	0	0	1	0	1	1	0B
15	0	0	0	0	1	1	1	1	0F
16	0	0	0	1	0	0	0	0	10
255	1	1	1	1	1	1	1	1	FF

134 © Cadence Design Systems, Inc. All rights reserved.



Certain binary widths may have specific names, although the names may not be consistently used.

An 8-bit binary is referred to as a byte with 4 bits as a nibble or half-byte. Nibbles can be represented in hex. Most PCs and game consoles are based on an Intel X86 processor architecture, which designates 16 bits as a WORD. Multiples of 16 bits are references as multiples of words, e.g., 32 bits is DWORD (double word) and 64-bit QWORD. Microsoft Windows still uses these terms.

However, in hardware, strictly speaking, a WORD is the fixed-width data handled by the processor. Therefore, for a modern 64-bit processor, a word is 64 bits.

Binary requires many more digits to represent numbers, as every digit is a power of 2 rather than, as in decimal, a power of 10. For example, we need 8 binary digits to represent decimal numbers from 0 to 255. It is more convenient to describe binary numbers using hexadecimal (or hex), where each hex digit represents 4 binary bits. The range of 4 bits is 0 to 15, and hex uses decimal digits for 0-9 and the letters A to F for 10 to 15. The table shows selected 8-bit values in decimal, binary, and hex. Specific binary widths are named. 8 bits is a byte, and 4 bits, a single hex digit, is a nibble or half-byte. 16 bits can be referred to as a word, and multiples of 16 bits as multiples of the word; for example, 32 bits is a double word or DWORD. However, this naming is inconsistent as a word can also refer to the data width of a processor, which could be 32 or 64 bits.

Binary Unsigned Data

N bit *unsigned* binary number has a range 0 to 2^N-1 .

In an implementation, the “width” of a word must be sufficient to cope with the biggest number encountered.

Number of bits(N)	Max value (2^N-1)
8	255
16	65535
32	4,294,967,295

Question: What happens if a word is not wide enough?



For unsigned, positive binary numbers, the number of bits defines the range of numbers represented. So, 8 bits have the range 0 to 255 decimal and 16 bits 0 to 65535. What happens if a bus or connection is not wide enough to store the required data? Well, typically, the data is truncated, and information is lost. For example, if you add one to a byte containing hex FF (decimal 255), then we need 9 bits to store the result, but with only 8 bits, the result rolls over to 0.

Binary Signed Data

2s complement representation for negative numbers.

- MSB determines sign of number.
 - "0" for positive, "1" for negative.
- To find -number, subtract from 0 and truncate to N bits.
 - Easiest way to do this is invert number and add 1.

+6 → 0110

invert 1001
add 1 1010

-6 → 1010

$$\text{-number} = \underline{\text{number}} + 1$$

Number of bits(N)	Range $-(2^{N-1})$ to $+(2^{N-1}-1)$
8	-128 to 127
16	-32,768 to +32,767
32	-2,147,483,648 to +2,147,483,647



Note: Any pattern of bits is a 2s complement number, which determines its arithmetic value.

To represent both positive and negative numbers called signed data, the most common format is 2's complement. Here the Most Significant binary Bit (or MSB) is reserved purely to define the sign of the number. If 0, the number is positive, and if 1, it is negative. To convert a positive binary value to negative, we subtract the value from 0 and truncate the result to the number of bus bits. An equivalent and easier way is to invert the binary value and add 1. So, to convert +6 to negative, we invert 0110 to 1001 and add 1 to give 1010. As the MSB is reserved purely for the sign information, this restricts the range of a signed bus. So 8bits represents from -128 to +127. Obviously, the hardware needs to understand if a binary value is a 2s complement or unsigned data and interpret it correctly.

Examples: Signed and Unsigned

For 8-bit numbers:

Number	Binary	2s Complement	Number
128	10000000	10000000	-128 (!)
0	00000000	00000000	0
1	00000001	11111111	-1
127	01111111	10000001	-127

biggest positive number

note: no +ve equivalent

0 assumed to be positive

Unsigned Arithmetic		Signed Arithmetic		Signed Arithmetic	
1111000	120	01111000	120	01111000	120
1000	+ 8	11111000	- 8	1000	+ 8
10000000	128	01110000	112	10000000	-128 !

Extra bit required
for sign information

Overflow!!

137 © Cadence Design Systems, Inc. All rights reserved.



These examples represent limiting cases of classes Max, Min, Zero, and a few other cases. Note that -2^{N-1} converts to itself (no positive equivalent) as does 0.

Here are some signed and unsigned examples using 8-bit data. The MSB sign bit in 2s complement restricts the range, so an 8-bit signed bus can hold -128 but not +128. The biggest positive number is 127. Signed and unsigned representations of 0 are identical, and as the signed MSB is not 1, zero is assumed to be positive in the 2s complement. We need to be careful about bit widths in arithmetic. In the unsigned example, adding 120 to 8 gives 128, which requires 9 bits. If the arithmetic was implemented in only 8 bits, we would lose the MSB, and the result would be 0. In signed arithmetic, we can add a 2s complement number to perform a subtraction. So, adding 2s complement of 8 to 120 subtracts 8. Here we rely on truncation to 8 bits to maintain the sign of the result. However, overflow can still be a problem in signed arithmetic. The last example shows that adding +8 to 120 overflows the result into the MSB, overwriting the sign information, and giving the result -128.

Other Signed Data Representations

1s complement

- Negative number is a simple inversion of a positive number:
 - Simple to implement
 - Arithmetic has exceptions

Off-set binary

- Most negative number in the range is treated as 0:
 - Arithmetic simplified
 - Conversion difficult for large ranges

Decimal	-128	-37	0	37	127
Offset Binary	00000000	01011011	10000000	10100101	11111111

138 © Cadence Design Systems, Inc. All rights reserved.

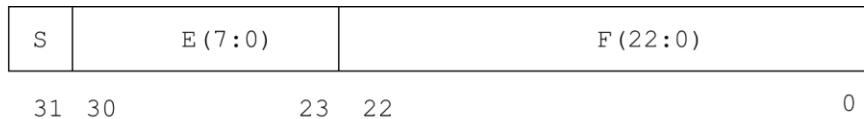


1s complement arithmetic is not as straight forward as 2s complement.

Try $1 - (-1)$ in 1s complement.

There are other signed number representations, but these are rarely used. 1s complement simply inverts a positive number to make it negative. This is simple to implement but gives issues in arithmetic. For example, there are positive zero and negative zero as two different values. Offset binary treats the most negative number in a range as zero and offsets all the other values from this. For the range -128 to 127, -128 is represented as zero, and all values increment from this. The arithmetic is simple, positive values only, and offset is good for ranges where bounds are not powers of two, but conversion into and out of the offset coding can be difficult for wide ranges.

Floating Point: IEEE Standard 754



IEEE single format consists of 3 fields in 32 bit word

- F - 23-bit fraction
- E - 8-bit biased exponent
- S - single bit sign

Represents data via the equation:

$$(-1)^s \times 2^{E-127} \times 1.F$$



Single-Format Bit Pattern Value

$0 < e < 255$ $(-1)s \times 2^{e-127} \times 1.f$ (normal numbers)

$e = 0; f \neq 0$ $(-1)s \times 2^{-126} \times 0.f$ (subnormal numbers)

(at least one bit in f is nonzero)

$e = 0; f = 0$ $(-1)s \times 0.0$ (signed zero)

$s = 0; e = 255; f = 0$ $+INF$ (positive infinity)

$s = 1; e = 255; f = 0$ $-INF$ (negative infinity)

$s = u; e = 255; f \neq 0$ NaN (Not-a-Number)

(at least one bit in f is nonzero)

For large and fractional number ranges, we need more efficient formats. One example is the floating-point IEEE 754 standard. For 32 bits, the format defines the lowest 23 bits, F as the fractional bits. Then the next 8 bits are a biased exponent E, in that, you subtract 127 from E and raise 2 to the result. The MSB S is the sign bit. The equation shows how to convert IEEE754 to decimal. Remember, the fractional bits are still powers of 2, so 0.1 binary is $0 + 2^{-1} = 0.5$ decimal.

Examples: Single Format

Number	Bit Pattern (hex)	Decimal Value
+0	00000000	0.0
-0	80000000	-0.0
1	3f800000	1.0
max value	7f7fffff	3.40282347e+38
min normal	00800000	1.17549435e-38
min subnormal	00000001	1.40129846e-45

IEEE 754 also defines:

- Double precision format (64-bit)
 - 52-bit fraction; 11-bit biased exponent; 1-bit sign
- Accuracy requirements on operations and conversions
- Floating-point exceptions



Accuracy requirements on floating-point operations include:

- Add, subtract, multiply, divide, square root, remainder, and round numbers in floating-point format to integer values, convert between different floating-point formats, convert between floating-point and integer formats, and compare.
- Five types of floating-point exceptions: invalid operation, division by zero, overflow, underflow, and inexact.

Here are some IEEE 754 examples in 32 bits (called binary32). IEEE 754 can represent normal numbers in the range of approximately 3.4×10^{38} to 1.17×10^{-38} , although there is a subnormal mode that allows numbers down to 1.4×10^{-45} . Binary32 is a single precision format, but IEEE754 also defines a 64-bit double precision format with 52 fractional bits. The standard not only defines the number format but also accuracy requirements on arithmetic operations, such as multiplication, division and square roots, and conversions to and from integers and between different floating-point formats. The standard also defines the result, and indication, of arithmetic exceptions such as dividing by zero, overflow and underflow.

Binary Coded Decimal

- Numbers can be represented as a “coded” equivalent of decimal.
- Each decimal digit (0...9) is stored as 4 bits.
- A N-digit number is made up from $4 \times N$ bits.

Binary	Decimal
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010...	<i>Not used</i>
1111	

Example

9832 = 1001 1000 0011 0010
 9 8 3 2



Multi-digit operations/arithmetic are possible but can be cumbersome.

Some data formats are more application specific. For example, Binary Coded Decimal, or BCD. This format uses 4bits to represent a single decimal digit in binary format, with the values 1010 to 1111 not used. Multiple 4-bit nibbles are used for multi-digit decimal numbers. BCD's advantages are in its close correspondence to decimal. So, the format is ideal for clock and counter displays. A disadvantage is that arithmetic on multi-digit values is more complex.

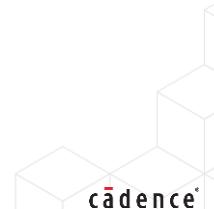
Gray Code

Gray code is a cyclic code.

- Only one-bit changes from state to state.
 - No erroneous states during transitions.
 - Lower power used in transitions.
- Simple relationship for conversion:
 - $\text{gray}(N) = \text{binary}(N+1) \text{ xor } \text{binary}(N)$

Binary	Gray
000	000
001	001
010	011
011	010
100	110
101	111
110	101
111	100

142 © Cadence Design Systems, Inc. All rights reserved.



In Gray code, only one bit changes state from state to state:

- No indeterminate or false states
- Few bits changing, therefore lower power utilisation

There is a simple relationship to convert binary to gray, e.g.:

- $\text{gray}(0) = \text{binary}(1) \text{ xor } \text{binary}(0);$
- $\text{gray}(1) = \text{binary}(2) \text{ xor } \text{binary}(1);$

Gray code is used mainly for simple counters and state vector encoding.

Another application-specific format example is Gray code. One disadvantage of counting in binary is that multiple bits must change in the transition from one value to the next. For example, counting from 3 to 4, three bits must toggle. Hardware delays may mean that not all bits change at the same time, leading to intermittent or false values during the transition. With Gray coding, only 1-bit changes from state to state, giving a safer transition. Fewer changing bits also give a lower power utilization. There is a simple conversion algorithm from binary to Gray code, as shown.

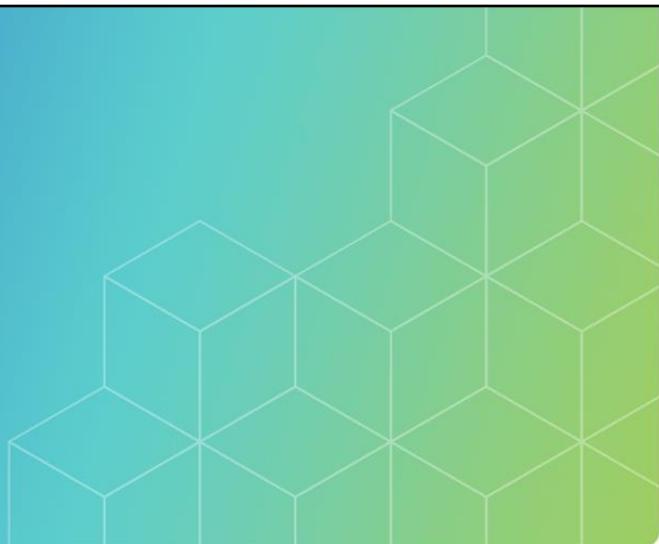
Submodule Summary

In this submodule, you learned about:

- A wide choice of formats for representing data in binary.
- The choice for particular implementation is determined by:
 - Range of data
 - Operations to be carried out on data
 - Need for standard formats
 - Destination of data
 - Power and reliability requirements



In summary, there is a wide choice of formats for binary data representation. The choice of format is determined by the range of values to be represented; the data operations required, whether you need standard formats; what the data is used for and the power and reliability requirements.



Submodule 3-1-2

Combinatorial Building Blocks

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Construct simple combinational logic.

Topics include:

- Basic gates
- Multiplexors
- Decoders
- Encoders
- Priority encoders



In this module, we will construct simple unclocked, combinational logic using basic gates. We will look at multiplexors, decoders and encoders.

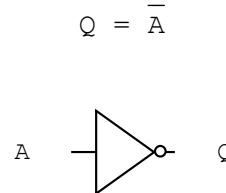
Basic Gates: Invertor

All combinational logic is constructed from simple gates:

- These are constructed from simple transistor circuits.
 - Technology dependent.

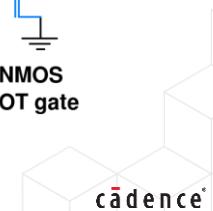
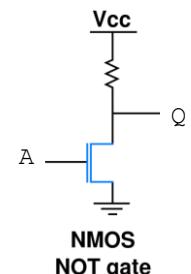
The simplest is the Invertor (NOT).

- Arithmetic symbol
 - Used for defining and manipulating logic with arithmetical equations (boolean algebra).
- Schematic symbol
 - Used for defining logic with diagrams (schematics).
- Truth Table
 - Used for defining gate or logic functionality.



A	Q
0	1
1	0

Transistor Implementation



146 © Cadence Design Systems, Inc. All rights reserved.

All combinational logic is constructed from simple logic gates, which are themselves constructed from simple transistor circuits. The exact implementation depends on the transistor technology. The simplest gate is the invertor implementing the NOT operator. This simply inverts the input, so a logic 0 becomes a logic 1 and vice-versa. There are various representations of an inverter for different purposes. The arithmetic symbol is used in defining and manipulating binary arithmetic (called Boolean algebra) and uses an over score or overline. The schematic symbol is used in graphical circuit diagrams, called schematics, and a truth table defines the behavior of the gate. A simple NMOS technology implementation uses a single transistor for an invertor. For a logic 1 input, the transistor turns on and connects the output to ground. For a logic 0 input, the transistor turns off, and the resistor pulls the output to high.

AND and NAND Gates

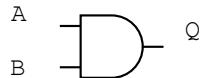
AND

Multiplication of bits

- Arithmetic symbol

$$Q = A \cdot B$$

- Schematic symbol



- Truth table

AND

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

147 © Cadence Design Systems, Inc. All rights reserved.

NAND

AND with inverter

- Arithmetic symbol

$$Q = \overline{A \cdot B}$$

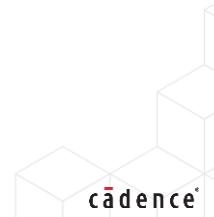
- Schematic symbol



- Truth table

NAND

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0



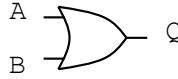
The AND gate is the multiplication of inputs. If A and B are 1, then the output is 1, else 0. The arithmetic symbol is a dot and the slide shows the schematic and truth table. The output of AND can be inverted to form a single NAND (Not-AND) gate, which has the symbols and truth table as shown. A circle on the output of the schematic symbol indicates the inversion.

OR, NOR, XOR, and XNOR Gates

OR

- Addition of bits

$$Q = A+B$$



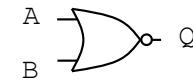
OR

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

NOR

- OR with invertor

$$Q = \overline{A+B}$$



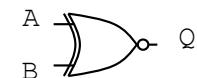
NOR

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	0

XNOR

- XOR with invertor

$$Q = \overline{A \oplus B}$$



XNOR

A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

cadence®

XOR

- Exclusive OR

- Output 1 if inputs are different

$$Q = A \oplus B$$



XOR

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

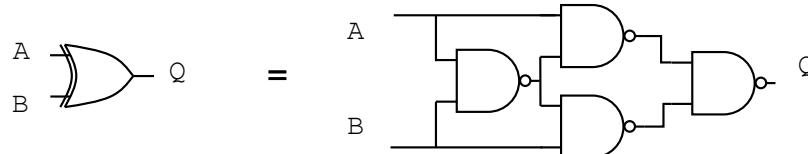
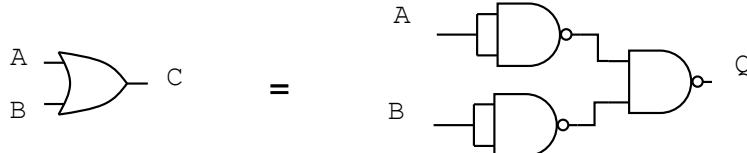
148 © Cadence Design Systems, Inc. All rights reserved.

The OR gate is addition of inputs. If either A or B are 1, then the output is 1 else 0. The arithmetic symbol is a plus sign, and the schematic and truth table are as shown. We add an invertor to an OR gate to form NOR. There is variant of OR called XOR or exclusive OR. Here the output is 1 if the inputs are different. The symbol is a plus enclosed in a circle. The output of XOR can be inverted to form an XNOR.

Functional Completeness

NAND and OR gates are functionally complete.

- All other logic can be built with either NAND or OR.
- Entire processors can be built just with either gate.



149 © Cadence Design Systems, Inc. All rights reserved.



All combinational logic is built from the simple gates we have shown. However, real-life hardware implementations are even simpler than this. They rely on the concept that NAND and OR gates are functionally complete. This means that all logic gates can be built with combinations of either NAND or OR. For example, an inverter is a NAND gate with both inputs connected together. An OR gate is two NAND invertors followed by a NAND. So, hardware can be implemented with a vast array of identical NAND gates, which are then connected together to give the required logic. In fact, entire processors can be built with just NAND or OR gates.

Multiplexor

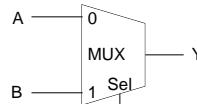
Multiplexor selects A or B input based on the value of SEL.

X is don't care
(could be 1 or 0)

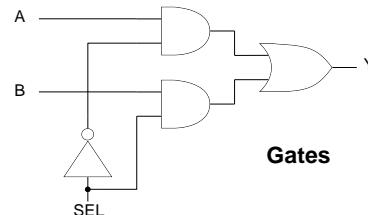
A	B	SEL	Y
p	X	0	p
X	q	1	q

Propagation delay

Truth Table



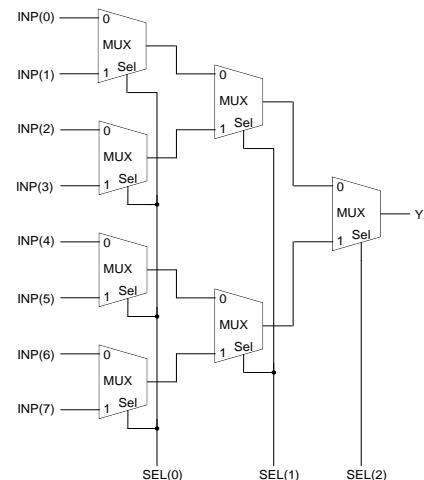
Schematic



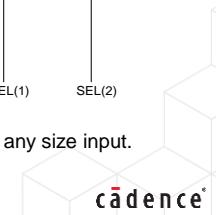
Gates

$$Y = A \cdot \overline{SEL} + B \cdot SEL$$

Arithmetic



MUXs can be scaled for any size input.



150 © Cadence Design Systems, Inc. All rights reserved.

MUX connects the 0 input to Y when SEL is 0 and the 1 input to Y when SEL is 1.

The gate level AND-OR circuit points to a specific implementation, but a multiplexer can be constructed in a number of different ways.

Note: MUX can be used to implement logical gates by various connections:

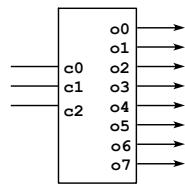
- Connect A and SEL to give AND gate

Note that building the 8 input MUX from 2-input MUXes results in a logical tree-like structure. Again, this is only one way of implementing the multiplexer.

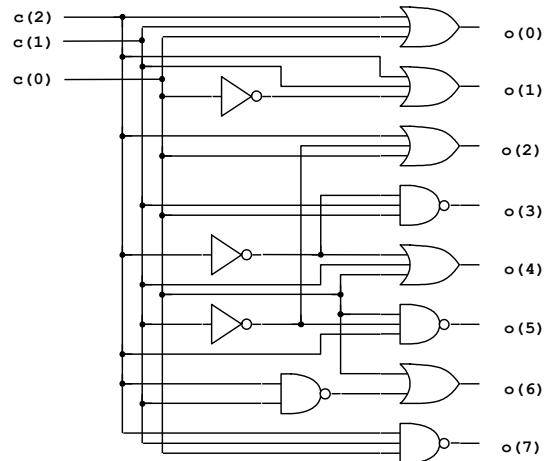
We don't describe designs in terms of basic gates; we use larger blocks, and here is one example, the multiplexor or MUX. A simple MUX uses the value of the input SEL to pass either the A or B data input to the output Y. The implementation is simple, as you can see from the gate diagram, although a MUX can be implemented in many ways. Note that the truth table uses additional symbols, as well as 1 and 0, for simplicity. For example, when SEL is 0, input A is selected, and the value of A (p is either 1 or 0) is output on Y. As B is not selected, we don't care about its value, indicated by X. There is also a time delay between the inputs changing and output updating, as the logic values propagate through the multiplexor gates. A MUX can be scaled for any size of the input. This example shows an 8 to 1 MUX with a 3-bit selector, implemented in 3 stages of 2 input MUXes. If SEL is 4 (binary 100), INP(4) is selected.

Decoder Architecture

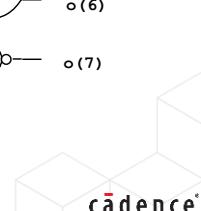
Decoder usually selects one output based on the value of the input.



c2 c1 c0	o_7	o_6	o_5	o_4	o_3	o_2	o_1	o_0
0 0 0	1	1	1	1	1	1	1	0
0 0 1	1	1	1	1	1	1	0	1
0 1 0	1	1	1	1	1	0	1	1
0 1 1	1	1	1	1	0	1	1	1
1 0 0	1	1	1	0	1	1	1	1
1 0 1	1	1	0	1	1	1	1	1
1 1 0	1	0	1	1	1	1	1	1
1 1 1	0	1	1	1	1	1	1	1



151 © Cadence Design Systems, Inc. All rights reserved.



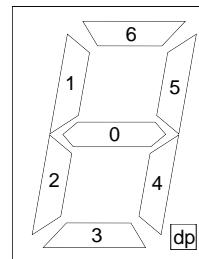
Note that architecture is superficially similar to the multiplexer – however, gate-level implementation is very different.

The example truth table converts a binary representation into a “one-cold” representation.

Another example of a combinatorial building block is a decoder. A decoder typically has more output than input bits and directly processes the inputs to produce the output. Although the architecture is superficially similar to the MUX, the gate-level implementation is very different. This example converts a binary input c to clear a single output bit of o , i.e., if c is 3, o_3 is zero, and all other bits are one. This binary representation, where only one bit can be zero, and all others are 1, is known as one-cold. The inverse, where only 1 bit can be 1, and all others are 0, is called one-hot. There are further variants; for example, “one-hot or all-cold” allows all bits to be 0 as well as 1 bit to be 1.

7-Segment Decoder Example

For viewing on a 7-segment LED display, a number must be translated into a pattern of lit segments.



Relationship depends on display

input	o7	o6	o5	o4	o3	o2	o1	o0
0000	1	1	1	1	1	1	1	0
0001	0	0	1	1	0	0	0	0
0010	1	1	1	0	1	1	0	1
0011	1	1	1	1	1	0	0	1
0100	1	1	1	1	0	0	1	1
0101	1	1	0	1	1	0	1	1
0110	1	1	0	1	1	1	1	1
0111	0	1	1	1	0	0	0	0
1000	1	1	1	1	1	1	1	1
1001	1	1	1	1	0	0	1	1
1010...	undefined							
1111								

Values >9 undefined →

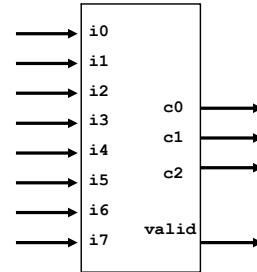


Encoder

Encoder usually outputs value based on input value or a combination of inputs.

- E.g., one-hot to binary with valid.

i7	i6	i5	i4	i3	i2	i1	i0	c2	c1	c0	valid
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	0	0	0	1	1
0	0	0	0	0	1	0	0	0	1	0	1
0	0	0	0	1	0	0	0	0	1	1	1
0	0	0	1	0	0	0	0	1	0	0	1
0	0	1	0	0	0	0	0	1	0	1	1
0	1	0	0	0	0	0	0	1	1	0	1
1	0	0	0	0	0	0	0	1	1	1	1



The opposite of a decoder is an encoder, which typically has more inputs than outputs. This encoder has a "one-hot or all-cold" input pattern and encodes the one-hot bit as a binary output. It also has an additional valid output which is cleared for the all-zero input, but set otherwise, allowing us to distinguish between the i0 one-hot and the all-cold inputs.

Priority Encoder

Inputs have a priority.

- Encoder returns a value corresponding to the highest priority input that is set.

pri7	pri6	pri5	pri4	pri3	pri2	pri1	pri0	c2	c1	c0	valid
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	1	X	0	0	1	1
0	0	0	0	0	1	X	X	0	1	0	1
0	0	0	0	1	X	X	X	0	1	1	1
0	0	0	1	X	X	X	X	1	0	0	1
0	0	1	X	X	X	X	X	1	0	1	1
0	1	X	X	X	X	X	X	1	1	0	1
1	X	X	X	X	X	X	X	1	1	1	1

X is don't care
(could be 1 or 0)



Unlike previous cases, the priority encoder is more naturally expressed using if..then..else, because it has the element of prioritization.

Classic use of a priority encoder is to change the output of a chain of comparators in a flash analog to digital converter into a binary word. A 10-bit A/D requires a 1024 to 10-bit priority encoder.

A priority encoder is an encoder where the input bits have a defined priority. Here, higher inputs bits have priority over lower bits. We use X in the truth table to indicate don't care where the bit could be 1 or 0. So if pri7 is one, we don't care about the other inputs, the output is 111 and valid is set. If pri7 is 0, then we check the next highest priority bit pri6. If set, the other bits are ignored, and the output is 110. The lowest priority bit is pri0, which is only checked if all other bits are zero. Finally, an all-cold input outputs zero, but with valid cleared. Therefore, the output corresponds to the highest priority bit set in the input.

Submodule Summary

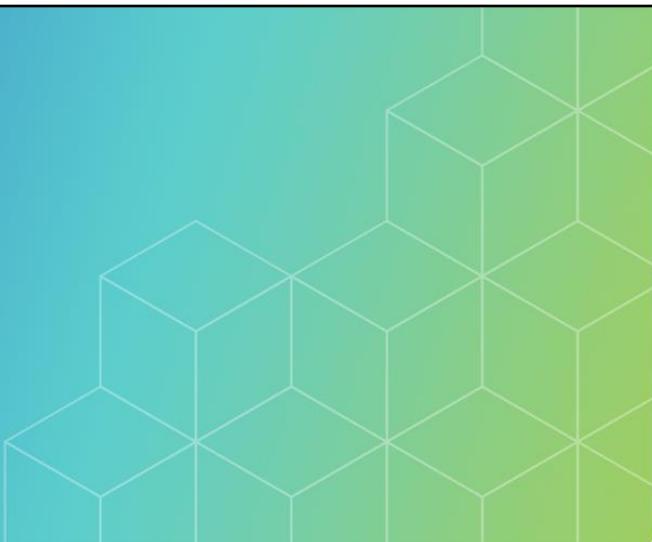
In this submodule, you learned:

All combinatorial logic is built from standard blocks:

- Which, in turn, are constructed from basic gates.
 - For which we can exclusively use functional complete gates like NAND or OR.
- MUXes conditionally select between different data inputs depending on select control inputs.
- Decoders typically convert a small number of inputs into a larger number of outputs.
- Encoders typically convert a large number of inputs into a smaller number of outputs.
- Considering input priority for a priority encoder.



In summary, all combinational logic is built from standard blocks, which are, in turn, are constructed from basic logic gates. The hardware implementation can use exclusively NAND or OR gates as these are functionally complete. MUXes select between data inputs depending on a select control input. Decoders typically have more outputs than inputs, and encoders have more inputs than outputs. If the inputs of an encoder are prioritized, then this is a priority encoder.



Submodule 3-1-3

Clocked Building Blocks and Synchronous Design

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Identify logic storage devices and their use.

Topics include:

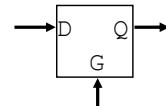
- Latches and flip-flops
- Latch versus flip-flop based design
- Combinational and synchronous timing
 - Set up and hold time
- Clock delays
- Effective clock periods



This submodule will look at storage devices. We will compare latches and flip-flops. We will look at combinational and synchronous timing and explore setup and hold times for registers. We will also examine clock delays and effective clock cycles.

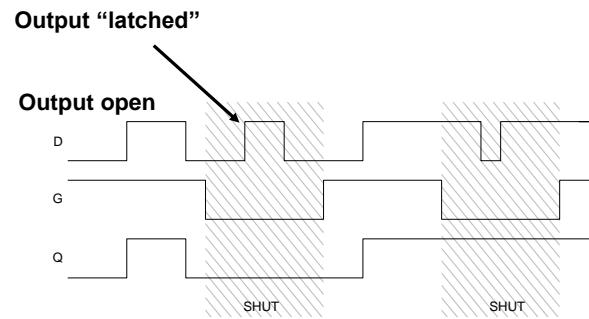
Latch

D	G	Q
X	0	Previous value
0	1	0
1	1	1



Stores the state of the input when there is a 0 on the gate input.

level-sensitive

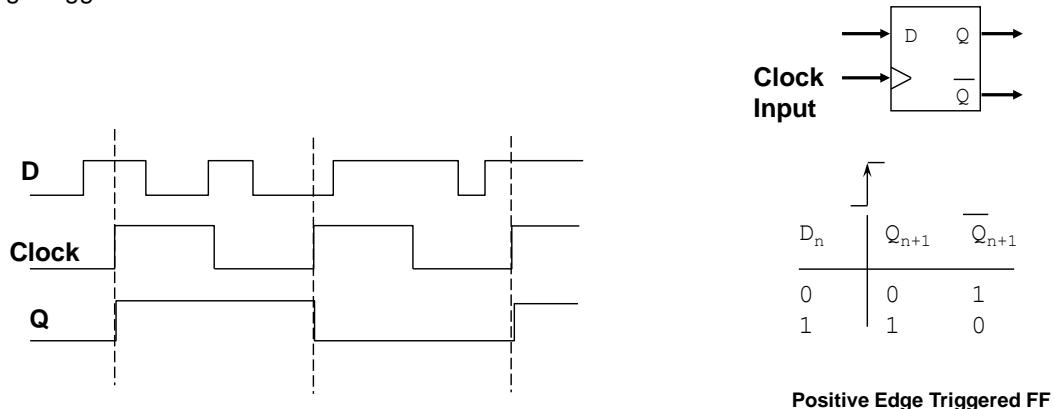


The basic storage element is the latch. This has a gate input G, and if the gate is high, then changes on the input D are passed to the output Q, and the latch is open. When G goes low, then the latch is shut, and the last value on D is “latched,” stored internally and driven on Q. While the latch is shut, changes on D are ignored. The latch is level sensitive in that the latch is open while G is high and shut while G is low.

Flip-Flop

Most commonly used flip-flop is the D-type.

- Stores the state of the input when there is a transition of the clock input.
- Edge-triggered.

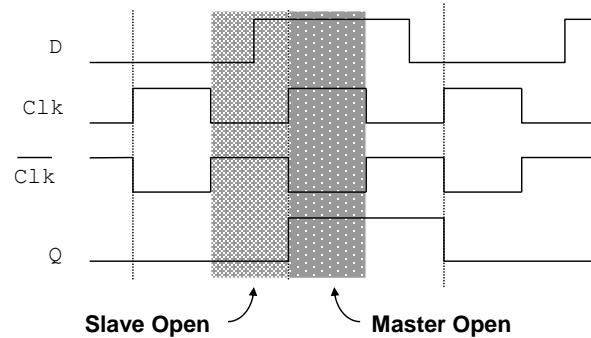
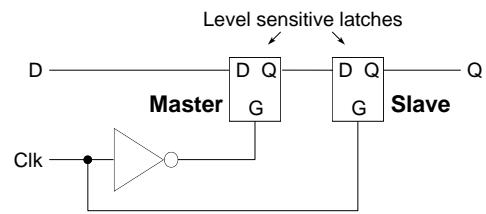


However, latches are rarely used. The most common storage device is the flip-flop, of which the most common variant is the D-type flip-flop. This has a clock input, and the flip-flop samples the input on a clock transition, stores the value internally and drives the value on the output Q and the inverse on the \bar{Q} bar. The flip-flop is edge-triggered in that a clock transition samples the input rather than a clock level. Flip-flops can be clocked on either the positive edge of the clock (0 to 1 transition) as here, the negative edge, or in some cases, both edges. Flip-flops clocked on both edges are used in Double-Data Rate (DDR) designs, typically where high data rates are required, such as memory access.

Flip-Flop Implementation

One implementation is the master-slave flip-flop.

- Consists of two cascaded latches.



160 © Cadence Design Systems, Inc. All rights reserved.



A flip-flop can be implemented as a master-slave with two cascaded latches. The clock is inverted to gate the first master latch, which is open while the clock is low. The second slave latch is gated directly from the clock and is open while the clock is high. While the clock is low, the master is open, and the slave holds the value of the output Q. When the clock goes high, the master shuts, the slave opens, and the input is sampled and driven on the output Q by the master. When the clock goes low, the master opens, the slave shuts, and the output of the master is sampled and driven on the output Q by the slave.

Flip-Flop or Latch-Based Design?

Flip-flops are convenient

- Described easily
- Fewer timing issues
- Easier test implementation
- Expensive in terms of area

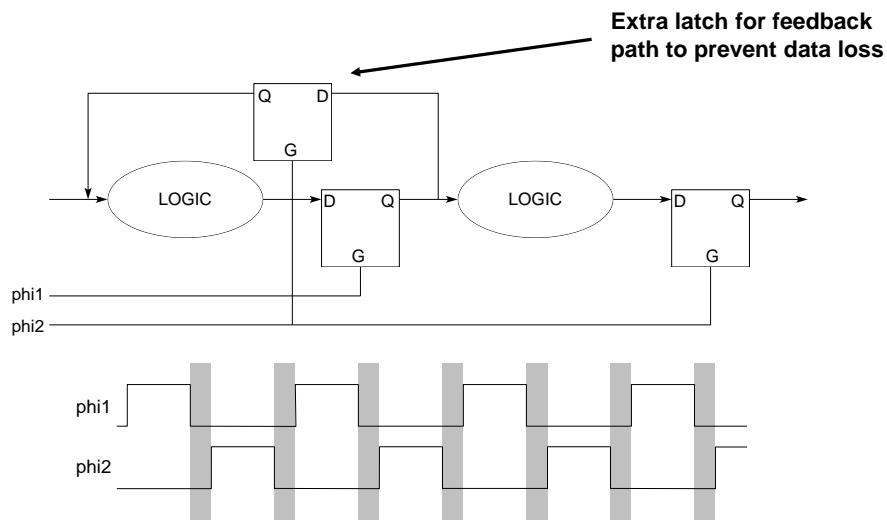
Greater efficiency can be achieved using latches

- Area savings of up to 50%
- Need 2-phase non-overlapping clocks
- Design is more difficult
- Test implementation more difficult



Although designs can be either flip-flop or latch-based, flip-flop designs are standard. They can be described more easily, are implemented with fewer timing issues, and are more easily tested. However, flip-flop designs are up to twice as large as latched designs in terms of area. Latch-based designs are more difficult to design as they need 2-phase, non-overlapping clocks, and testing is more difficult.

Latch-Based Design



162 © Cadence Design Systems, Inc. All rights reserved.



Here is the clocking scheme for a latch-based design. It requires two clocks, phi1 and phi2, which are of opposite phases and non-overlapping. The clocks are connected to alternate latches so that when phi1 is high, the two logic blocks are connected, but data is blocked by the latch clocked by phi2. The phi1 latch shuts before the phi2 latch opens, so data is passed sequentially through the logic blocks by the alternate latches. Note that if a logic block requires a feedback path, we need an extra latch gated from the other clock to hold the feedback data and prevent an open loop.

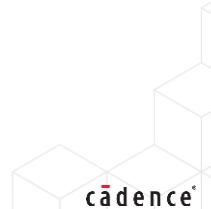
Latch Design Issues

Latches are “open” for part of the time.

- Flip-flops transfer data at edges.
- If the logic before the storage element undergoes spurious transitions, an open latch will allow these to propagate.

This may:

- Cause problems in later parts of the circuit.
- Increase power consumption.



Another issue with latch designs is that latches are open for part of the time. Any transitions on the logic before the latch will be propagated to the next logic block. This could cause problems in logic downstream from the latch. The transitions also increase power consumption. Flip-flop designs sample data at the rising or falling edge, which filters out spurious data transitions and gives us a more stable design with lower power consumption.

Combinational Timing

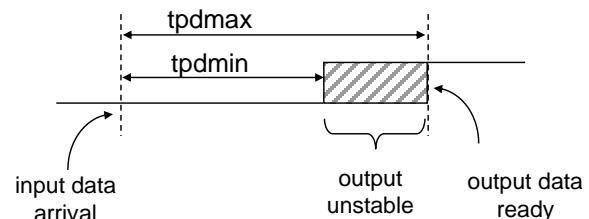
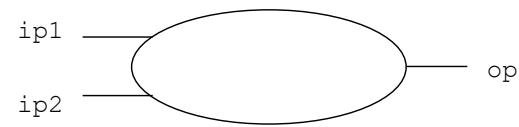
Each gate in combinational logic has a delay:

- Propagation delay tpd

Typically, many logic paths from inputs to outputs:

- Shortest path is minimum delay $tpdmin$
- Longest path is maximum delay $tpdmax$
- Between $tpdmin$ and $tpdmax$, outputs are changing

$tpdmax = 20\text{ns}$
 $tpdmin = 15\text{ ns}$
delay input to output = 20 ns
latency
new data input every 20 ns
throughput



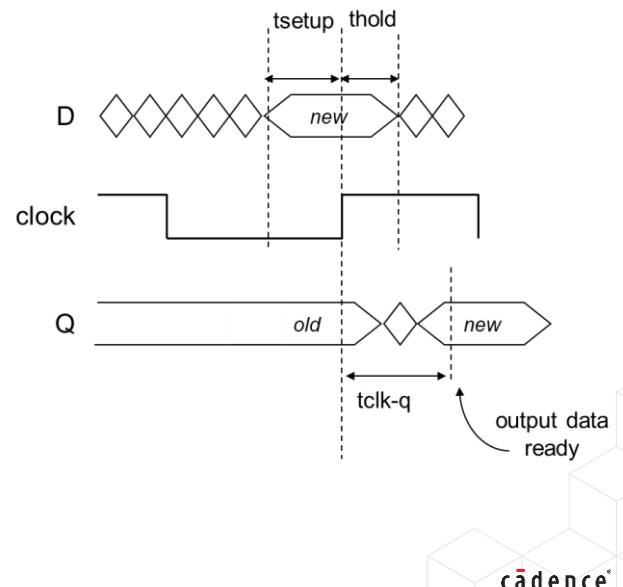
Let's talk about timing and, firstly combinational timing. Each basic gate has a propagation delay tpd between the inputs changing and the outputs updating. The more gates in a logic path from input to output, the greater the delay. However, there are typically many logic paths through a combinational block. The shortest path gives the minimum delay $tpdmin$, and the longest path the maximum $tpdmax$. Therefore, from data arriving at the input to a block, the outputs will be changing between $tpdmin$ and $tpdmax$ after the arrival time. The outputs will not be ready until after $tpdmax$. The maximum delay from input to output, $tpdmax$, is called the latency. If we sample output data as soon as it is ready, then new data can arrive at a minimum of $tpdmax$ after the previous data, and this gives us the throughput of the block.

Register Timing: Setup, Hold and Clock-to-Q

A register samples D and outputs Q on the clock edge.

Sampling requires that the input remains stable for:

- Setup time before the clock rising edge (tsetup).
- Hold time after the clock rising edge (thold).
- If setup or hold times are violated, the register samples a changing value.
 - No guarantee of value stored.
 - Metastability problems.



Output appears on Q as a short delay after the clock edge.

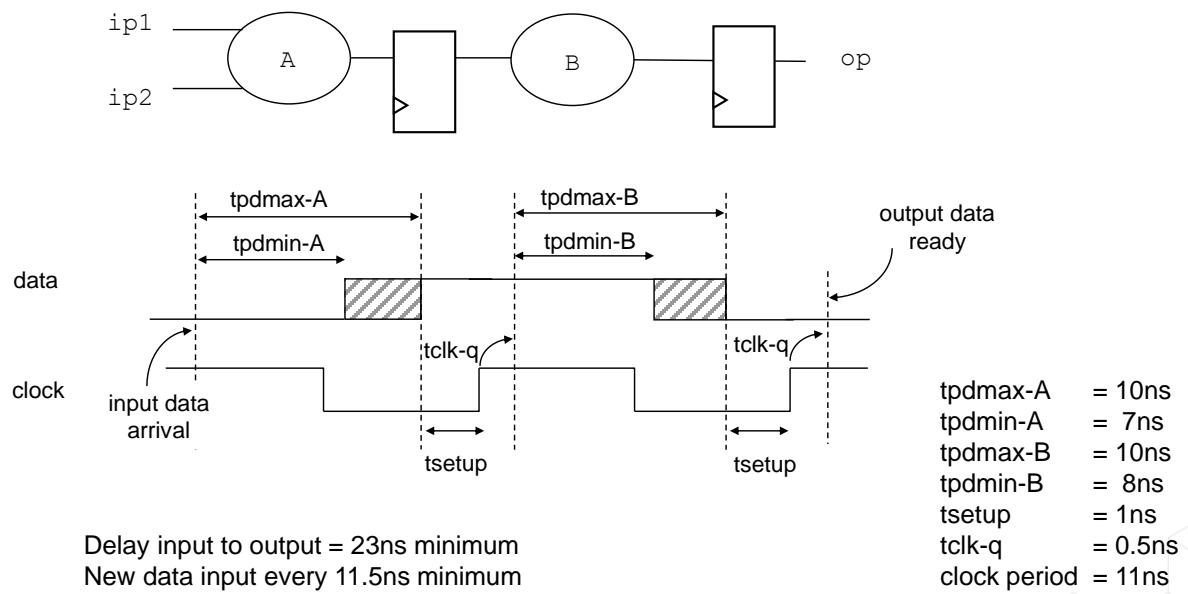
- Clock to Q delay (tclk-q).

If setup and hold times are violated, then the register is trying to sample the input D while it is changing. Sampling a changing value can lead to either an indeterminate value being sampled (either 0 or 1, but we don't know which) or meta-stability. In metastability, the input D value sampled may be between the voltage thresholds for logic 0 or logic 1. This may result in the transistors of the register no longer behaving as switches but turning partially on. The metastability condition can then be passed to subsequent logic in the design. Metastable conditions on the register output may not last long but could last long enough to cause problems.

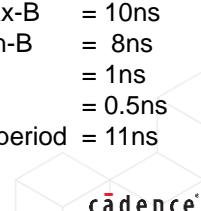
Metastability can be a specific issue for asynchronous inputs, which are not synchronized to the clock and, therefore, may violate the setup and hold times. There are standard techniques for handling such inputs, such as double-registering. This uses two registers in series on the input, with the aim that if the first register has metastability, there is a full clock cycle for the output to stabilize before being sampled by the second register.

Flip-flops, or registers, have their own timing constraints. The input D is sampled on the rising clock edge, but registers require D to be stable for a time before the edge called the setup time (tsetup). D must also be stable for a time after the clock edge, called the hold time (thold). If either setup or hold time is violated, the register samples and outputs a changing value which may be between the logic voltage thresholds. This is called metastability. An intermediate voltage between thresholds can result in the transistors of a gate no longer behaving as switches but partially turning on and propagating the value to other logic in the design. A metastable condition may not last long, as the voltage is dragged to one of the logic levels by noise, crosstalk, or other electronic effects in the hardware, but the condition may last long enough to cause spurious transitions and other problems. The final timing characteristic of registers to consider is clock to Q delay tclk-q. The output Q changes a short delay after the clock edge.

Sequential Timing



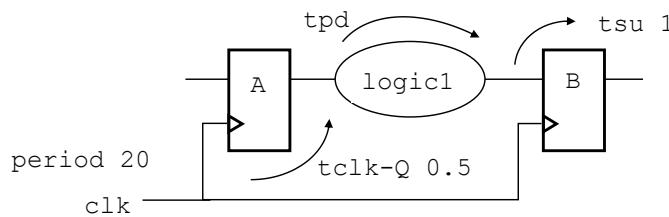
166 © Cadence Design Systems, Inc. All rights reserved.



Putting together combinational propagation delay and register timing allows us to start to build a timing model for the design.

By putting together combinational and register timing, we can analyze design timing. Tpdmax for logic A is 10ns, and the setup time for the register is 1ns, so new data could arrive at A with a minimum of 11ns apart. Data is released from the register after 0.5ns (tclk-q), the maximum delay through B is 10ns (tpdmax-B), and the data must arrive 1 ns before the clock edge (tsetup). Therefore, the minimum clock period for logic B is 11.5ns. The delay from input to output is tpdmax-A + tsetup + tclk-q + tpdmax-B + tsetup + tclk-q (for the output register) = 23ns.

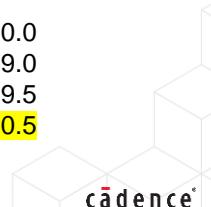
Setup Time



$tpd_{max} = 18$	
- clk rising edge(1):	0.0
- data op from A:	0.5
- data op from logic1:	18.5
-	
- clk rising edge(2):	20.0
- data expected at B:	19.0
- data arrival at B:	18.5
- slack	+0.5

$tpd_{max} = 19$	
- clk rising edge(1):	0.0
- data op from A:	0.5
- data op from logic1:	19.5
-	
- clk rising edge(2):	20.0
- data expected at B:	19.0
- data arrival at B:	19.5
- setup time violation	-0.5

167 © Cadence Design Systems, Inc. All rights reserved.



If tpd , the *longest* propagation delay through logic1, is 18ns, then:

- A rising edge of the clock arrives at the clock pin of register A at, say, 0ns.
- This releases data on the Q output of register A at 0.5ns (clock -Q delay).
- Data arrives at the input of register B at 18.5ns (18ns propagation delay in logic1).
- The next rising edge of the clock arrives at the clock pin of register B at 20ns.
- The setup time sets the latest possible time for the data to be ready on the input of B at 19ns (clock arrival – setup time).
- Data is present at 18.5ns, must arrive by 19ns, giving 0.5ns of slack in the timing path.

If tpd , the *longest* propagation delay through logic1, is 19ns, then:

- The data arrives at the input of register B at 19.5ns.
 - $T_{clk-Q} + \text{propagation delay in logic1}$.
- The next rising edge of the clock arrives at the clock pin of register B at 20ns.
- The setup time sets the latest possible time for the data to be ready on the input of B at 19ns (clock arrival – setup time)
- Data is present at 19.5ns but must arrive by 19ns, giving a setup timing violation of 0.5ns in the timing path.

The clock period and register setup time restrict the maximum propagation delay of logic. Consider this design with a clock period of 20ns, tsetup 1ns and t_{clk-Q} 0.5ns. If the tpd_{max} of logic1 is 18 ns and a clock-rising edge arrives at register A at, say, 0ns. Data is output from A at 0.5ns (t_{clk-Q}) and arrives at register B at 18.5ns. The next clock edge arrives at B at 20ns. Tsetup is 1ns, so the latest the data can arrive at B is 19ns (clock period – tsetup). If the data is present at 18.5ns, that gives 0.5ns of slack in the timing path. If, however, logic1 tpd_{max} is 19ns, then the data arrives at register B at 19.5ns. This is 0.5ns into the setup time, so there is a 0.5ns timing violation on B.

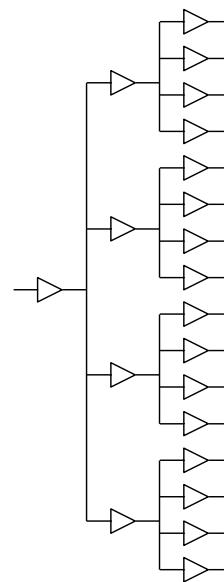
Clock Distribution

Clock distribution is very important:

- A single clock may route to every register in the design.
- Long, heavily loaded nets.
- Major delay and power considerations.

Special techniques for clock distribution:

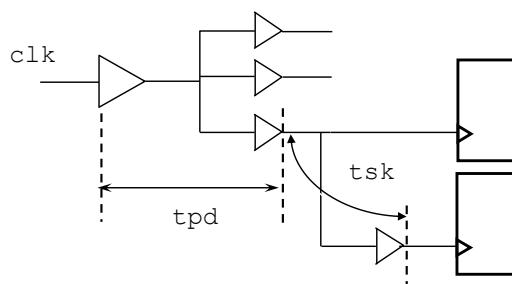
- FPGA: dedicated fast global routing.
- ASIC: clock tree inserted during layout.
 - Must be “load balanced.”
 - Every branch sees the same load (approx.).
 - To minimize propagation delay, transition time, and, therefore, power.
 - Special tools or manual tweaking.
 - Several architectures.



168 © Cadence Design Systems, Inc. All rights reserved.

Before we look at hold time, we need to address clock distribution. Every register in the design could be connected to a single clock, so clock nets tend to be very long and heavily loaded. As length and load contribute to power and delay, there are special techniques for optimizing clock distribution. Field-Programmable Gate Arrays (FPGAs) have fast, dedicated global connections built into the technology, but for ASIC devices, the clock connections must be inserted during layout. The aim is to balance the clock tree so every branch sees about the same register load and to minimize propagation delay, transition time and power. There are several architectures that can be used, and special software tools, sometimes aided by manual tweaking, to implement these architectures.

Modeling Clock Trees



Two parameters to model:

- tpd: propagation delay through the clock tree.
- tsk: skew between branches.

Positive skew affects hold timing.

Negative skew affects setup timing.

There are two parameters to consider when modeling clock trees. First is the propagation delay between the clock input and the arrival at a register. Second, is the skew, which is the difference between clock propagation for adjacent registers. Positive skew indicates the clock arrives at a source register before the target register. This affects hold timing. In negative skew, the clock arrives at a target register before the source register, and this affects setup timing.

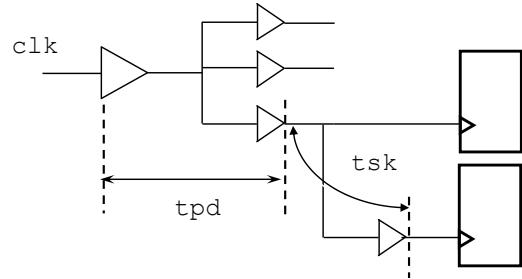
Modeling Clock Trees

Two parameters to model:

- tpd: propagation delay through the clock tree.
- tsk: skew between branches.

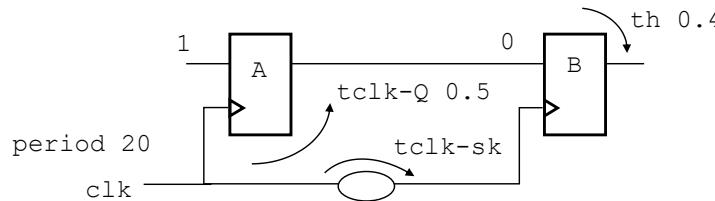
Positive skew affects hold timing.

Negative skew affects setup timing.



There are two parameters to consider when modeling clock trees. First is the propagation delay between the clock input and the arrival at a register. Second, is the skew, which is the difference between clock propagation for adjacent registers. Positive skew indicates the clock arrives at a source register before the target register. This affects hold timing. In negative skew, the clock arrives at a target register before the source register, and this affects setup timing.

Hold Time



$t_{clk-sk} = 0$

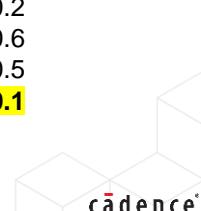
- clk rising edge(1) at A: 0.0
- data op from A: 0.5
- data arrival at B: 0.5

- clk rising edge(1) at B: 0.0
- data expected at B: 0.4
- data arrival at B: 0.5
- slack +0.1

$t_{clk-sk} = 0.2$

- clk rising edge(1) at A: 0.0
- data op from A: 0.5
- data arrival at B: 0.5

- clk rising edge(1) at B: 0.2
- data expected at B: 0.6
- data arrival at B: 0.5
- hold time violation -0.1



Hold time violations are not so common, but a typical example may be in the timing of a shift register where the clock net is subject to positive skew, i.e., there is no logic between the registers, and the clock edge arrives at the source register *before* the target register.

If clock skew (t_{clk-sk}) is 0ns:

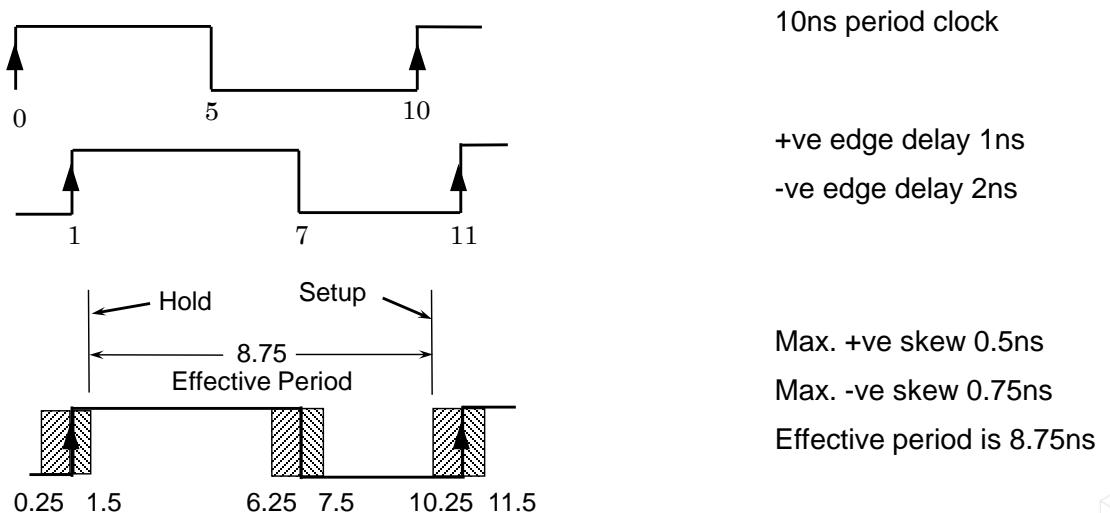
- A clock edge arrives at register A at, say, 0ns and releases a '1' onto the output.
- The '1' arrives at the input of register B at 0.5ns (t_{clk-q}).
- The clock edge arrives at register B at 0ns.
- This means the data '0' currently at the input of B must remain unchanged until 0.4ns.
 - Clock arrival + hold time.
- The '0' must remain until 0.4ns is overwritten at 0.5ns, giving 0.1ns of slack in the timing path.

If the clock skew (t_{clk-sk}) is 0.2ns:

- The '1' arrives at the input of register B at 0.5ns (t_{clk-q}).
- The **same** clock edge arrives at register B at 0.2ns due to clock net delay.
- The data '0' currently at the input of B must remain unchanged until 0.6ns.
 - Clock arrival + hold time.
- The '0' must remain until 0.6ns, but is overwritten at 0.5ns, giving a hold time violation of 0.1ns in the timing path.

Hold time violations are not common, but a typical example may be for adjacent registers with no combinational logic, where the clock has positive skew and arrives at the source register before the target register. Consider this design with a clock period of 20ns, thold 0.4ns and tclk-q 0.5ns. The input of register A is 1 and B is 0. The clock edge arrives at A at say 0ns and releases a 1 from A at 0.5ns (t_{clk-q}). The 1 arrives at register B at 0.5ns. If the skew t_{clk-sk} is 0ns, then the clock edge arrives at B at the same time as A, 0ns. The current input at B, 0, must remain unchanged until 0.4ns (thold). The 1 from A does not arrive until 0.5ns, so we have a slack of 0.1ns in the timing path. If, however, the clock skew is 0.2ns, the clock edge arrives at B at 0.2ns. The 0 input at B must remain unchanged until 0.6ns (clock arrival + thold) but is overwritten by the 1 arriving from A at 0.5ns, giving a hold time violation of 0.1ns.

Effective Clock Period



Clock propagation and skew can mean registers see an effective clock period that is different from the period of the input clock. If we have a 10ns period clock rising at 0ns, first, we apply propagation delay to the clock edges. It is common to have different delays for positive and negative edges, and this is technology specific. We take the worst-case scenario of a register with the maximum positive skew, with the latest arriving clock edge, feeding a register with the maximum negative skew, with the earliest arriving edge. The latest the edge arrives at the first register is 1.5ns (edge + propagation delay + maximum positive skew). The earliest the edge arrives at the second register is 10.55ns (edge + propagation delay - maximum negative skew). Therefore, the effective clock period is $10.25 - 1.2 = 8.75\text{ns}$, or 12.5% less than the actual clock period.

Submodule Summary

In this submodule, you learned:

Latches and flip-flops are the basis of most real designs.

- Latches are area efficient but difficult to design.
- Flip-flop designs are simpler but up to 50% bigger.

Synchronous design with a single clock is relatively simple.

- Timing easily understood.
- Implementation straight-forward.
- Synthesis easier.
- Real life is different.
 - Asynchronous inputs.
 - Multiple clock domains.

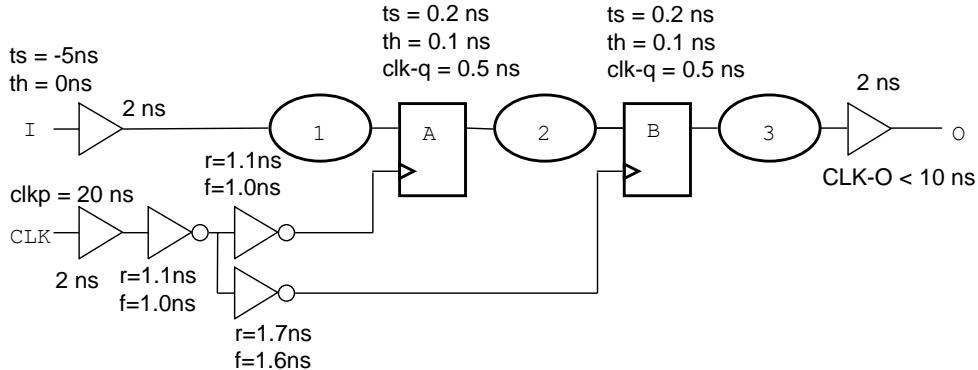
Clock delays must be allowed for in the design.

- Skew and propagation delay can reduce the effective clock period.



In summary, latches and flip-flops storage elements form the basis of most designs. Although latches are more area efficient, latch-based design is more difficult, and so is rarely used. Flip-flops are easier but consume more area. Synchronous designs with a single clock have simple timing and a straightforward implementation, making synthesis much easier. However, real-life designs frequently have asynchronous inputs and multiple clock domains to consider. Remember that clock propagation delay and skew must be allowed for in the design, as they can alter the effective clock period.

Test Your Understanding



- What is the clock propagation and skew at registers A and B?
- What is the effective clock period for logic 2?
- Work out the maximum and minimum delays for clouds 1, 2 and 3.

174 © Cadence Design Systems, Inc. All rights reserved.

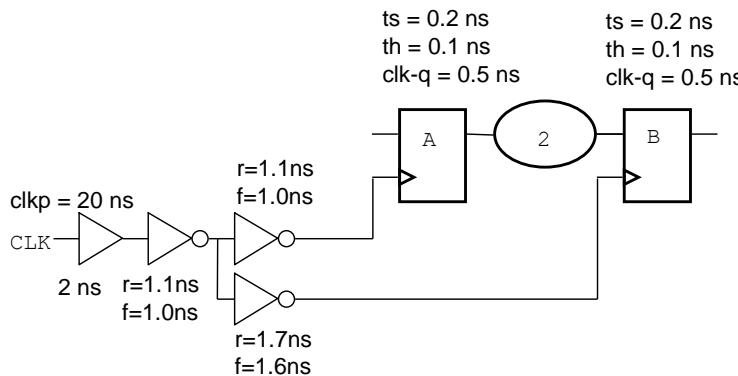


Assume registers A and B are rising edge triggered.

So, to test what we learned in this module, here is a quick exercise. Pause here, write down your answer, and check the results on the next slides. Note that r is a rising edge delay, and f is a falling edge delay. ts is the setup and thold the hold time.

Test Your Understanding

(continued)



- What is the clock propagation and skew at registers A and B?
- What is the effective clock period for logic 2?
- Work out the maximum and minimum delays for cloud 2.



The clock path includes inverters as well as buffers; therefore, when accumulating propagation delays, remember that a rising edge on the input to an inverter becomes a falling edge on the output.

Rising edge clock propagation delay at register A = $2 + 1.1 + 1.0 = 4.1\text{ns}$

Rising edge clock propagation delay at register B = $2 + 1.1 + 1.6 = 4.7\text{ns}$

Therefore, the clock skew at B with respect to A is $+0.6\text{ns}$, and there is no negative skew between A and B.

The effective clock period for logic 2 is, therefore, $20 + 0.6 = 20.6\text{ns}$. This shows that positive skew between two registers can result in an effective clock period greater than the actual clock period.

The maximum delay for cloud 2 relates to setup times. First clock edge arrives at A at 4.1ns and the next clock edge at B at 24.7ns

Data released is from A at 4.6ns . It must arrive at B no later than $24.7\text{ns} - T_{\text{setup}} = 24.5\text{ns}$. Therefore, the maximum delay for 2 is $24.5 - 4.6 = 19.9\text{ns}$.

Minimum delay for cloud 2 relates to hold times. First clock edge arrives at A at 4.1ns , and the same clock edge arrives at B at 4.7ns .

Data released is from A at 4.6ns . It cannot arrive at B earlier than $4.7\text{ns} + T_{\text{hold}} = 4.8\text{ns}$. Therefore, the minimum delay for 2 is $4.8 - 4.6 = 0.2\text{ns}$.

This shows that although positive clock skew can increase the effective clock period, it does so at the expense of a minimum propagation delay between registers. So, there must be some logic in cloud 2, and A and B could be not be part of a shift register.

Assume rising edge triggered registers and the CLK rising edge is input at 0ns . The edge arrives at A at $0 + 2 + 1.1$ (rising edge) + 1.0 (falling edge) = 4.1ns . Edge arrival at B is $0 + 2 + 1.1$ (rising) + 1.6 (falling) = 4.7ns . Therefore, with respect to A, the CLK propagation delay is 4.1ns , and the positive skew at B is 0.6ns . Therefore, the effective clock period for logic2 is 20.6ns

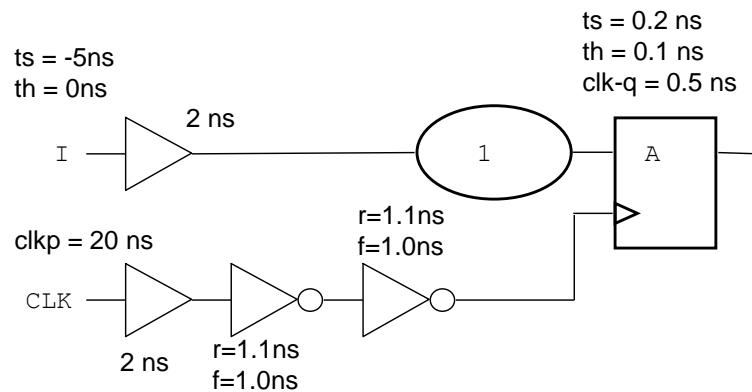
Maximum delay for cloud 2. CLK edge arrives at A at 4.1ns . Data released from A at $4.1 + 0.5$ ($t_{\text{clk-q}} = 4.6\text{ns}$).

Next clock edge arrives at B at $20 + 4.7 = 24.7\text{ns}$. Latest arrival for data at B must be $24.7 - 0.2$ ($t_{\text{setup}} = 24.5\text{ns}$). Therefore, the maximum delay for A is 24.5 (setup at B) – 4.6 (data release from A) = 19.9ns .

Minimum delay for cloud 2. Data released from A at 4.6ns . Same CLK edge arrives at B at 4.7ns , and data must be stable until $4.7 + 0.1$ ($t_{\text{hold}} = 4.8\text{ns}$). Therefore, the minimum delay for logic2 is 4.8 (hold at B) - 4.6 (data release from A) = 0.2ns . The positive skew at B with respect to A gives us an effective clock period greater than the actual clock period at the cost of a minimum delay requirement for cloud 2.

Test Your Understanding

(continued)



- Work out the maximum and minimum delays for cloud 1.

176 © Cadence Design Systems, Inc. All rights reserved.



The maximum delay for cloud 1 relates to setup times.

Latest data arrival time at the input is 5 ns , and at cloud 1 are, 7 ns .

The next clock edge at A at 24.7 ns , and the setup time means data must arrive at A by $24.7 - 0.2 = 24.5\text{ ns}$. Therefore, the maximum delay for cloud 1 is $24.5 - 7 = 17.5\text{ ns}$.

The minimum delay for cloud 1 relates to hold times.

Earliest data can be changed at the input at 0 ns , which changes it at cloud 1 at 2 ns . The first clock arrives at A at 4.1 ns , and so the data cannot change at A before $4.1 + Thold = 4.2\text{ ns}$.

Therefore, the minimum delay for cloud 1 is 2.2 ns .

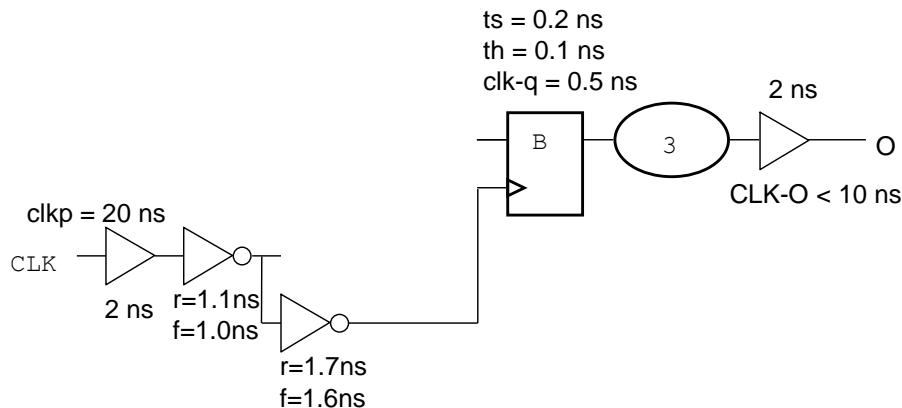
This again shows that positive clock skew can increase the effective clock period at the expense of a minimum propagation delay requirement.

CLK edge arrival at A is 4.1 ns . Data arrival at I is -5 ns with respect to the clock, so data arrives at cloud 1 at $-5 + 2 = -3\text{ ns}$. Setup time for A is $4.1 - 0.2$ ($t_{setup} = 3.9\text{ ns}$). Therefore, the maximum delay for cloud 1 is $3.9 - (-3) = 6.9\text{ ns}$.

Data removal from I is 0 ns , so removed from cloud 1 at 2 ns . Hold time for A is $4.1 + 0.1$ ($thold = 4.2\text{ ns}$). Therefore, the minimum delay for cloud 1 is $4.2 - 2 = 2.2\text{ ns}$.

Test Your Understanding

(continued)



- Work out the maximum and minimum delays for cloud 3.

177 © Cadence Design Systems, Inc. All rights reserved.



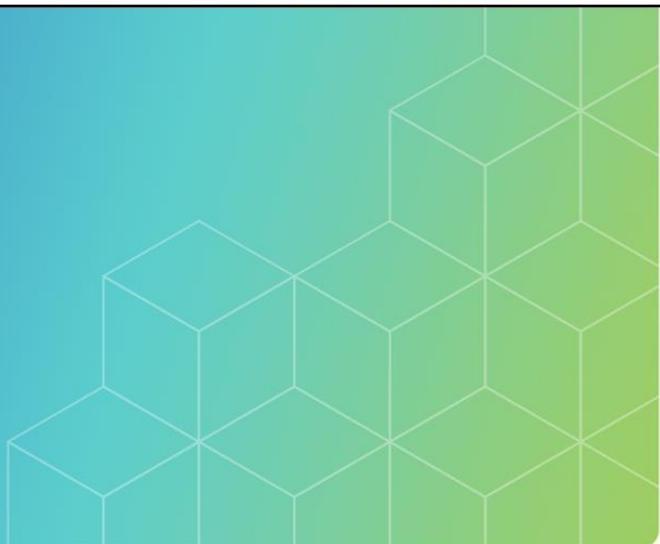
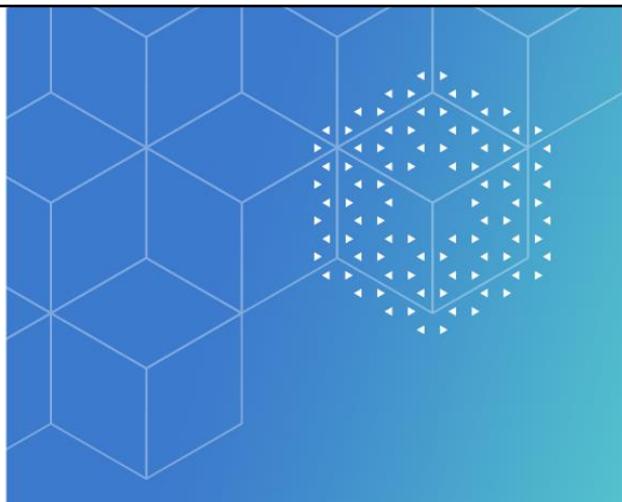
The first clock reaches B at 4.7ns, and data is released at 5.2ns.

Clock to output requirement is less than 10ns, so data must be output from cloud 3 at 8ns. Therefore, the maximum delay through cloud 3 is $8 - 5.2 = 2.8\text{ns}$.

As there is no minimum clock-to-output requirement, there is no minimum delay for cloud 3. If cloud 3 were absent, the requirements would still be met.

Output requirement is a maximum delay of 10ns from CLK to O. CLK edge arrives at B at 4.7ns. Data released from B at $4.7 + 0.5$ ($t_{\text{clk-q}}$) = 5.2ns. Data must be released from cloud 3 at $10 - 2 = 8\text{ns}$.

Therefore, the maximum delay for cloud 3 is $8 - 5.2 = 2.8\text{ns}$. As there is no minimum delay requirement on the output, there is no minimum delay for cloud 3. So, cloud 3 could be removed, and the specification would still be met.



Submodule 3-1-4

Arithmetic Building Blocks

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Identify the basic building blocks for implementing binary arithmetic using synthesis.

Topics include:

- Addition
- Subtraction
- Implementation examples

All other arithmetic operations (multiplication, division, trigonometry) can be implemented with combinations of adders and other simple logic (e.g., shift registers).

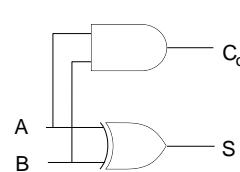


This submodule will look at basic building blocks used by synthesis for binary arithmetic. We will look at addition, subtraction, and selected implementation examples. Note that all other arithmetic operations can be implemented with combinations of adders and simple logic, such as shift operations.

Half-Adder

Adds two binary digits producing sum and carry.

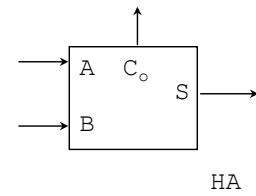
Truth Table		S	C_o
A	B		
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1



Gate Implementation

Boolean Equations

$$\begin{aligned} S &= A \oplus B_i \\ C_o &= A \cdot B \end{aligned}$$



Schematic Symbol



Most basic building block for performing binary arithmetic.

The most basic arithmetic block is the half-adder. This adds two single inputs, A and B, to give a sum S and carry C output, as shown by the truth table. The truth table gives us the simple gate-level implementation of an AND gate for the carry generation and an XOR gate for the sum.

Full-Adder

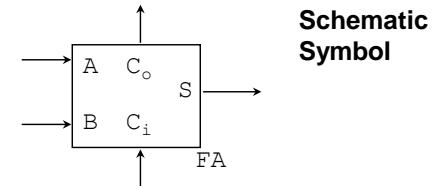
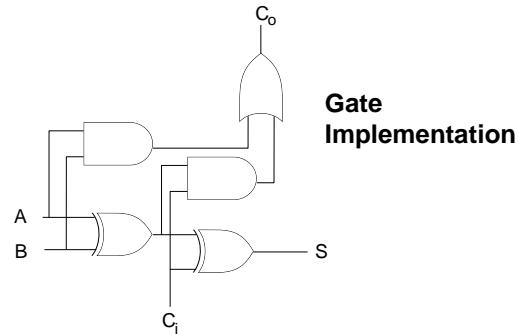
Add carry-in to make a full adder.

Truth Table	A	B	C _i	S	C _o
	0	0	0	0	0
	0	0	1	1	0
	0	1	0	1	0
	0	1	1	0	1
	1	0	0	1	0
	1	0	1	0	1
	1	1	0	0	1
	1	1	1	1	1

Boolean Equations

$$S = A \oplus B \oplus C_i$$

$$C_o = A \cdot B + C_i \cdot (A \oplus B)$$



181 © Cadence Design Systems, Inc. All rights reserved.



Full adder required to make efficient N-bit adder.

To make a full-adder, we must expand the truth table for a third input, carry-in. From the gate implementation, we can see the full-adder is simply 2 half-adders, one to add the inputs A and B, and the other to add this result to carry-in and produce the full-adder sum. The full-adder also has an OR gate to or the carrys from the two half-adders to give the full-adder carry-out.

Implementation of Addition

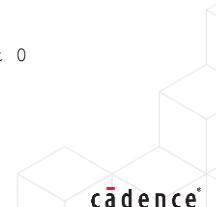
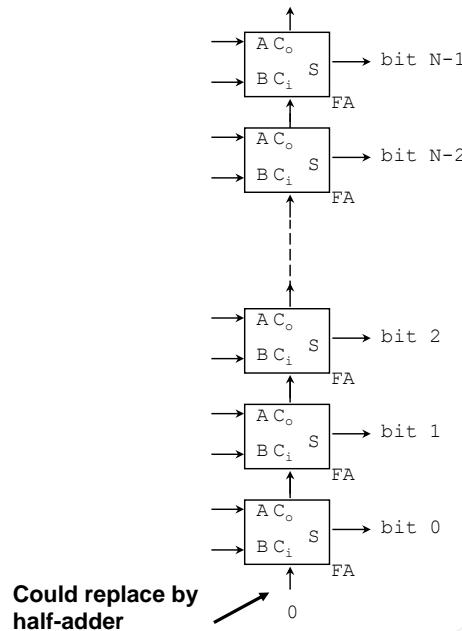
For N-bit addition, cascade N full adders.

- Ripple carry adder architecture.

Relatively slow:

- Assuming gates all have the same delay, the delay from C_i to C_o is two unit delays.
- Delay from LSB to C_o of bit $N-1$ is $2N$ unit delays.
 - For example, for an 8-bit adders, the delay is 16 units.

Question: How could you do a subtraction?



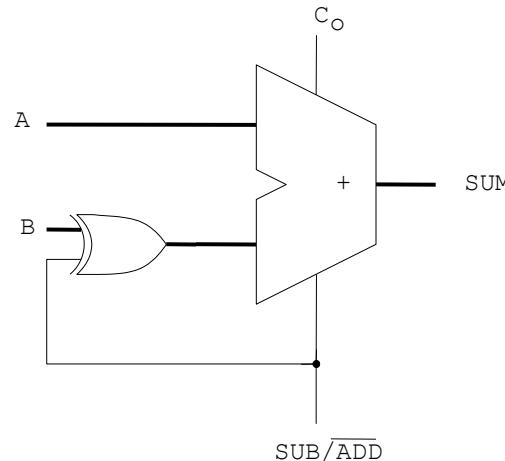
To implement N-bit addition, we can cascade N full adders. Each full-adder adds a bit of the two inputs with the carry-in from the previous bit, generates the sum, and passes the carry-out to the adder for the next bit. At the Least Significant Bit (LSB) bit 0, the carry-in is wired to 0, or we could replace the bit0 full-adder with a half-adder. This architecture is called a ripple carry adder, as the carries ripple up the adder cascade. This makes the architecture relatively slow. If we assume all gates have the same unit delay, then the propagation delay from carry-in to carry-out for each stage is two units. In the worst-case scenario of the carry from bit0 cascading all the way to bit $N-1$, the overall adder delay is $2N$ units. For an 8-bit adder, the delay is 16 units. This performs addition; how can we implement subtraction?

Implementation of Subtraction

Can be done using half or full-subtractor.

Can also be done by remembering that $-B = \overline{B+1}$

Full Subtractor	B_i	A	B	D	B_o
	0	0	0	0	0
	0	0	1	1	0
	0	1	0	1	1
	0	1	1	0	0
	1	0	0	1	1
	1	0	1	0	0
	1	1	0	0	1
	1	1	1	1	1



183 © Cadence Design Systems, Inc. All rights reserved.



$$A - B = A + (B)2s\ complement = A + \sim B + 1$$

Implement by inverting the data into B input and set C0 to 1.

To make a combined adder/subtractor, pass the B inputs through an XOR gate, combining each bit with ~add/sub to provide controllable inversion and sourcing C0 from ~add/sub.

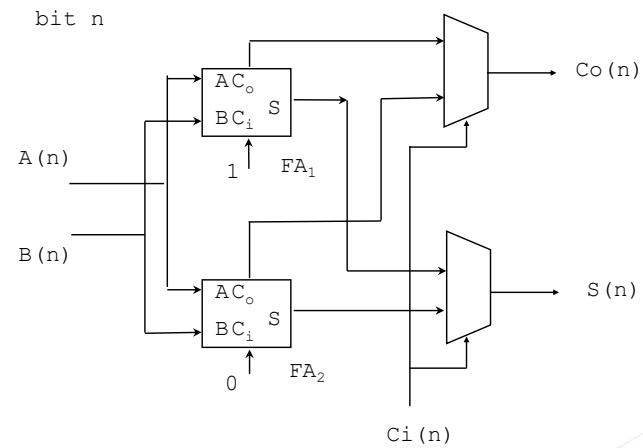
We could implement half and full subtractor blocks to implement a ripple-borrow subtractor. However, remember that in 2s complement, to make a number negative, we invert and add one. So, $A - B$ is $A + 2s$ complement of B , which is $A + \sim B + 1$. We can add one at the carry-in of the bit0 adder, and we could also use this bit to invert B using an XOR gate. If SUB/ADDbar is 1, we are inverting B and adding one at bit0, so the result is $A - B$. If SUB/ADDbar is 0, B is not inverted, the carry-in is 0, and the result is $A + B$. Therefore, the same architecture can be used for both addition and subtraction.

Faster Addition: Carry Select Adder (CSA)

Many different architectures to speed up addition.

For example, Carry Select Adder (CSA).

- Two ripple carry adders for each bit:
 - One hard-wired for carry in of 1
 - $C_{i1} = '1'$
 - One hard-wired for carry in of 0
 - $C_{i2} = '0'$
- Carry-in now selects between pre-computed values via multiplexers.
- Delay C_i to C_o determined by the speed of the multiplexer.
- CSA architectures commonly used by specialist datapath compilers.



184 © Cadence Design Systems, Inc. All rights reserved.



Carry-Select Adder trades area for speed.

CSA architecture is quite fast.

- Carry may still ripple from LSB to MSB, but since they carry can select from pre-computed addition, speed is only restricted by delay through a multiplexor.

There are many different adder architectures with better speed than the ripple-carry adder. Here is one example, the Carry Select Adder (CSA). This architecture implements two full adders for each bit, one hardwired for a carry-in of 1 and the other for a carry-in of 0. The carry-in for a stage now selects between these pre-calculated values via a MUX. A carry may still ripple up from bit0 to bitN-1, but now the delay is determined by the speed of the MUX, which is easier to optimize. So, a CSA architecture is faster but over twice the size of a ripple carry adder. In reality, CSA may be restricted to 4-bit slices of the addition, with ripple carry used between slices. This gives a good balance between area, speed, and complexity. To access a CSA architecture in synthesis, you may need a specialist datapath synthesis tool.

Submodule Summary

In this submodule, you reviewed various methods of implementing basic arithmetic:

- Half adder
- Full adder
- Ripple Carry adder
- Subtraction
- Carry Select adder

There are time/area tradeoffs for different approaches.



For simple operations, the synthesis tool can infer arithmetical operations from code and select architecture from time/area constraints.

For more complex operations, the structure and architecture of the operation must be built by the designer.

In summary, there are many options for implementing basic arithmetic in hardware. We have seen how half-adder blocks can be built into full-adders for a ripple carry architecture adder, which can also be used for subtraction. We also saw an alternative adder architecture called the Carry Select Adder. Each different architecture has trade-offs for time and area.

Test Your Understanding

- What distinguishes a half-adder from a full-adder?
- What is an advantage of a ripple-carry adder architecture over a Carry Select Adder?
- What makes the Carry Select Adder a fast implementation?

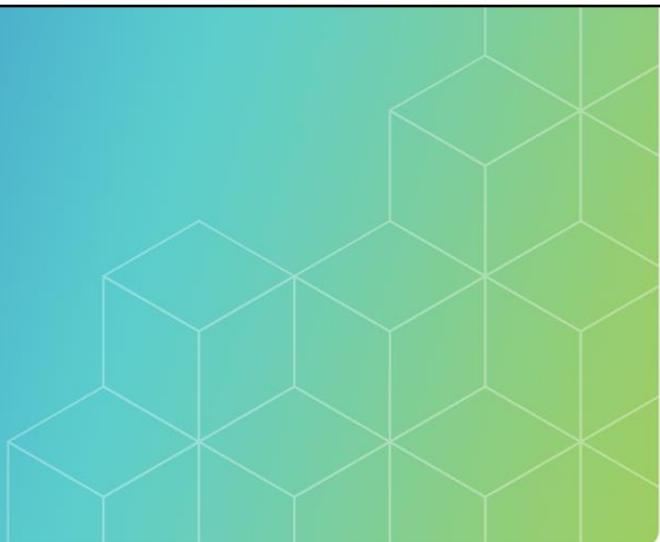
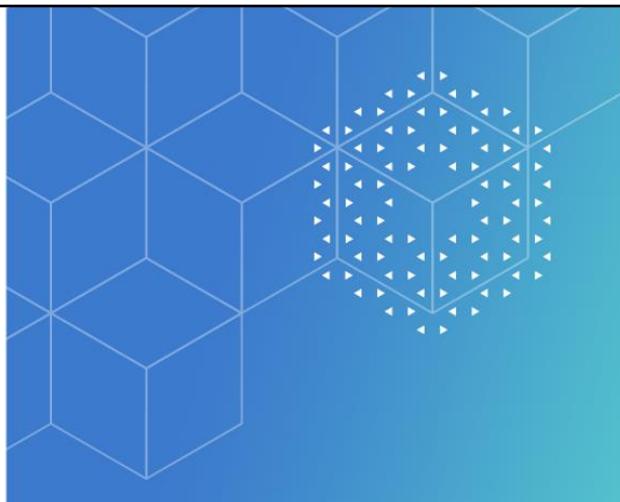


To test what we learned in this module, here is a quick exercise. Pause here, write down your answers, and check against the solutions.

The difference between a half-adder and a full-adder. A half-adder adds two inputs. A full-adder is two half-adders plus an OR gate to add two inputs plus a carry-in.

The advantage of a ripple-carry adder is that it is smaller than a Carry Select Adder.

The Carry Select Adder is a fast implementation as it uses MUXes to select between pre-calculated bit values. This is faster than cascading the carry through a ripple carry adder chain.



Submodule 3-1-5

Finite State Machines

cadence®

This page does not contain notes.

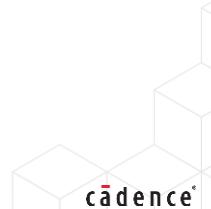
Submodule Objective

In this submodule, you will:

- Define Finite State Machine (FSM) and different ways of implementation.

Topics include:

- FSM basics
- Types of FSM
- FSM structure
- State encoding



This submodule will look at Finite State Machines (FSMs) and different implementations. We will look at FSM basics, different types of FSMs, and optimization of the FSM structure and state encoding.

What Is a Finite State Machine?

An FSM is a control path.

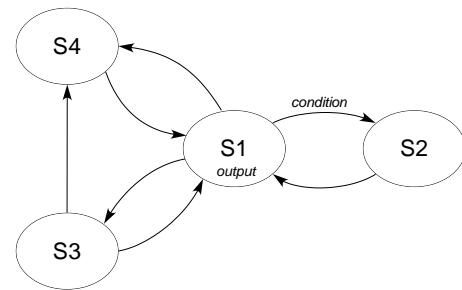
- It is clocked, i.e., sequential.
- It has a number of defined “states” that it can be in.
- Transition between states are determined by the combination of the current state and the current inputs to the FSM at a clock edge.
- Outputs from the FSM can be a function of:
 - Current state only (Moore)
 - Current state and the current inputs (Mealy)



An FSM is predominantly used to design sequential controllers and control paths. It is a clocked block with an internal state chosen from a defined list of allowed values. The FSM transitions from one state to another based on the current state and the inputs. The FSM outputs are defined by the current state only in a Moore FSM or the current state and the inputs for a Mealy FSM, as we shall see.

Moore FSM Representation

- S1 . . . S4 are the states of the FSM.
- Directed arrows are allowed transitions between states.
- *condition* is the input condition that must be satisfied to undergo a specified state transition.
- *Output* is the set of outputs associated with the state.
 - Requires a separate state for each combination of outputs.
- Transitions occur on the clock edge.



190 © Cadence Design Systems, Inc. All rights reserved.



FSM can only be in one of the defined states.

In a Moore FSM, the outputs are functionally dependent only on the current state.

- Hence output values are allied to states.

The diagram shows only one condition and output.

In reality, all states will have an associated set of outputs, and all transitions should be labeled with a transition.

- Unlabelled transitions are automatically taken at the next clock, i.e., unconditionally.

An FSM is defined using a state diagram. This defines the allowed states of the FSM, S1 to S4. Arrows define the allowed transitions between states, and the transitions can be dependent on an input condition. The output values for each state are defined, and in a Moore FSM, the outputs are a condition of the state only. Therefore, we need a separate state for each unique combination of outputs.

Remember, state transitions are synchronous to the clock edge. Although this diagram shows only one condition and output, in reality, all states will have an output, and all transitions should be labeled with a condition unless they are truly unconditional.

Mealy FSM Representation

Similar representation except ...

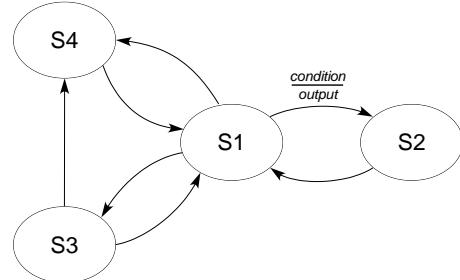
Associated with each transition is:

condition
output

condition defines when the transition occurs.

output is the set of outputs associated with the *inputs and state*.

- Each state can have different output combinations.
- Fewer states are required!



191 © Cadence Design Systems, Inc. All rights reserved.



In a Mealy FSM, the outputs are functionally dependent on the current state and inputs.

Condition and output are associated with the transition.

- Output is dependent on the input (defines condition) and the state (where the arrow is directed to).

A Mealy state diagram is very similar to a Moore. However, as the outputs in a Mealy depend on the inputs as well as the current state, the transition condition is also associated with a set of outputs. As each state can produce different outputs, we no longer need a separate state for each output combination, and a Mealy FSM can usually be implemented in fewer states.

FSM Example: 2-Bit Clearable Counter

2-bit counter

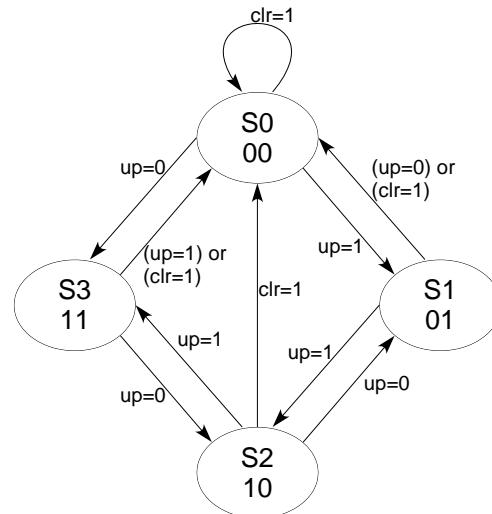
- 4 states

Fully synchronous

Clear count to 00 when $\text{clr}=1$

Count up when $\text{up}=1$

Count down when $\text{up}=0$



$\text{clr}=0$ conditions not shown for clarity

Question: What kind of FSM is this?

192 © Cadence Design Systems, Inc. All rights reserved.



When $\text{UP}=1$, states follow in a clockwise direction: 00, 01, 10, 11

When $\text{UP} = 0$, states follow anti-clockwise

All states are connected to S_0 (00).

- When $\text{CLR} = 1$, S_0 is the next state.

By changing the outputs associated with the states, the type of counter can be changed.

- For example, $S_0=00$, $S_1=01$, $S_2=11$, and $S_3=10$ gives a Gray counter.

Only the output is decoding changes.

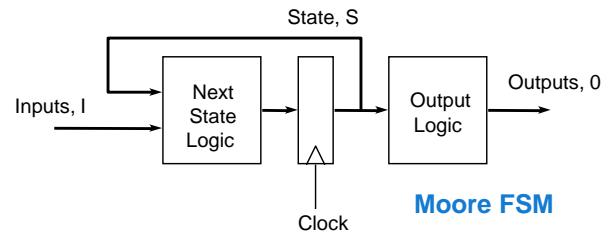
Here is an example of a state diagram for a 2-bit clearable counter. If clr is 1, then we move to state S_0 and output 00. While $\text{up} = 1$, we move clockwise through the states $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_0$, and the output counts up. When $\text{up} = 0$, we move anticlockwise through the states $S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow S_0$, and the output counts down. Remember, up is sampled, the state changes and the outputs are made on the clock edge. Note that the $\text{clr}=0$ conditions are not shown for clarity. What kind of FSM is this? Since the outputs are only a function of the current state, it is a Moore FSM.

Implementation of an FSM

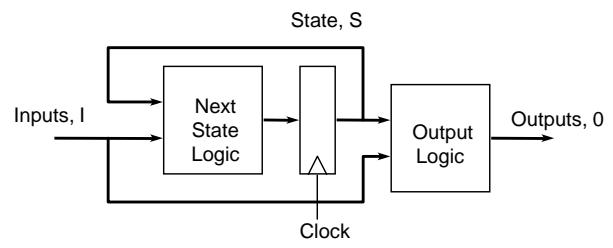
Next state logic decides the next state based on the current state and inputs.

Output logic decodes state (or states and inputs) to produce outputs.

Register stores current state.

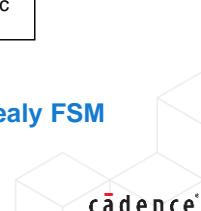


Moore FSM



Mealy FSM

193 © Cadence Design Systems, Inc. All rights reserved.



For a Moore FSM, the output logic depends only on the state.

For a Mealy FSM, the output logic depends on inputs as well as state.

To implement an FSM, we need three main blocks. The next state logic selects the next state by examining the current state S and the inputs I . On the clock edge, the next state becomes the current state held in the state register. The output decode logic generates the outputs from the current state for a Moore FSM or the state and the inputs for a Mealy FSM.

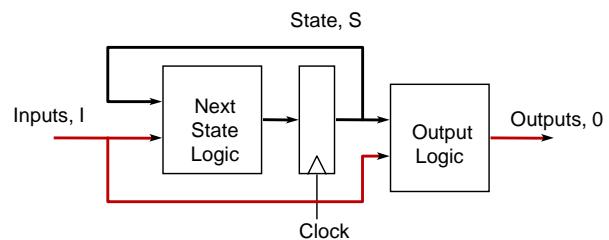
Mealy FSM Outputs

Purely combinational path from inputs to outputs.

- Outputs updated as soon as inputs change.

High probability of glitches on outputs.

- Unsynchronized inputs
 - Input arriving at different times causes intermediate transitions on outputs.
- Glitches on inputs
 - Fed straight through to outputs.
- Different paths in output logic
 - Inputs arrive at the same time, but different paths to output with different timing cause intermediate output transitions.



Mealy FSM



Unsynchronized inputs may be caused by FSM inputs being generated from different logic sources. Different path lengths will introduce differing delays on the inputs, even if all are generated from registered logic with the same clock.

Inputs may be generated by purely combinational logic or even directly from the outputs of a Mealy FSM. Therefore, the inputs may glitch before reaching a steady state, and these glitches can affect the FSM outputs.

Even if the inputs are synchronized and glitch-free, different logic paths in the output logic may introduce skew between the inputs and hence glitches in the outputs.

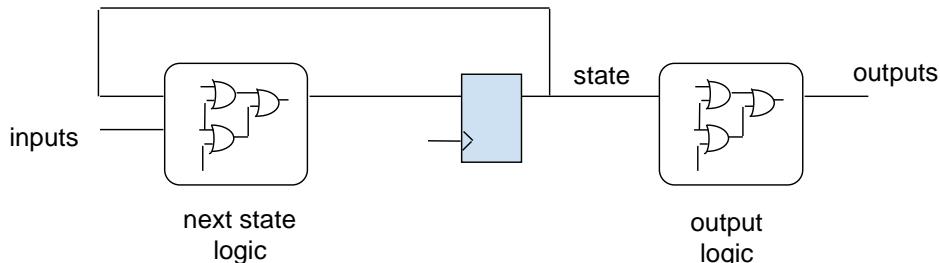
The issue with a Mealy FSM is that there is a pure combinational path from the inputs to the outputs. Therefore, changes in the inputs can immediately update the outputs. This gives a high probability of glitches or spurious transitions on the outputs from several possible sources. The inputs may be unsynchronized. If the inputs come from different logic sources, they may not all arrive at the same time, causing the outputs to glitch as they first settle on an intermediate state before being updated by a late arriving input. Even if the inputs are synchronized, any glitches on the inputs are fed straight through to the outputs. Finally, even if the inputs are synchronized and glitch-free, different timing paths in the output logic may introduce skew between the inputs and hence intermediate transitions on the outputs.

FSM Implementations: Combinatorial Outputs

Characteristics

- Simple implementation
- Slower clock to output
- No protection for Mealy outputs

Moore FSM



195 © Cadence Design Systems, Inc. All rights reserved.



Requires additional logic levels

- Slower clock to output

Let's look at some standard FSM implementations. The simplest form is a single register block for the state and combinational blocks for the next state logic and outputs. Disadvantages are possible longer delays from clock to output, depending on the size of the output logic, and no protection for Mealy outputs due to the direct combinational path to the output.

FSM Implementations: Registered Outputs

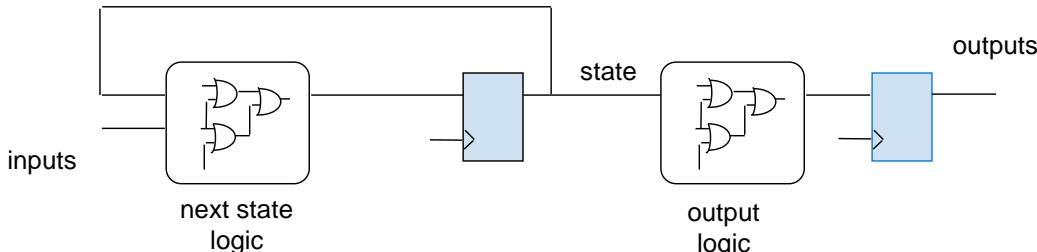
Requires extra flip-flops for all the outputs.

- Removes glitches on outputs.
- Useful for Mealy machines!

Creates a clock cycle latency on each output.

- Can your design afford it?

Moore FSM



196 © Cadence Design Systems, Inc. All rights reserved.



Good design practice

- Isolates time-critical combinational logic paths

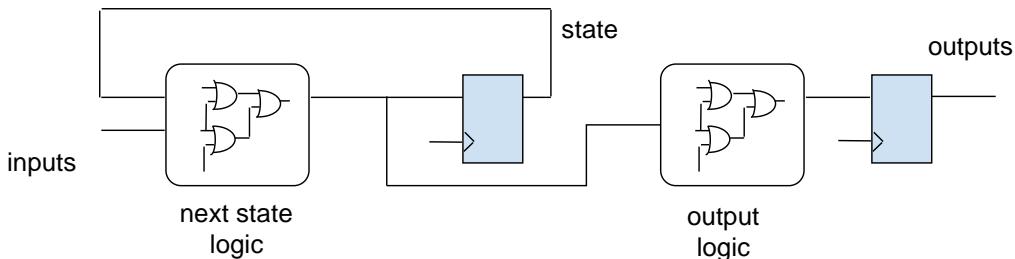
A standard option to remove glitches is to register the FSM outputs. This breaks the direct combinational path to outputs for a Mealy machine, and so prevents output glitches. Another advantage is zero delays from clock to output. The disadvantage is an extra clock cycle delay for the output. When the state changes, the outputs for that state are not available until the next clock cycle. This could be a problem for your design.

FSM Implementations: Advanced Registered Outputs

Outputs are decoded from the Next State information.

- Removes the clock cycle latency on the outputs.
- Affects the maximum working frequency.
 - Next state and output decode logic are cascaded.

Moore FSM



197 © Cadence Design Systems, Inc. All rights reserved.



Affects the max working frequency

- Next state logic and output logic are cascaded
- Some tools can merge the logic

The extra clock cycle latency for registered outputs can be removed by decoding the outputs from the next state logic rather than the current state register. Now a clock loads the next state into the state register and the outputs for that state into the output register on the same cycle. The disadvantage is a longer timing path from inputs to outputs through the cascaded next state and output logic. This will limit the maximum working frequency of the FSM.

State Bits Decoded Outputs

Output values correspond directly to state vector bits.

- Requires careful encoding of the state bits.

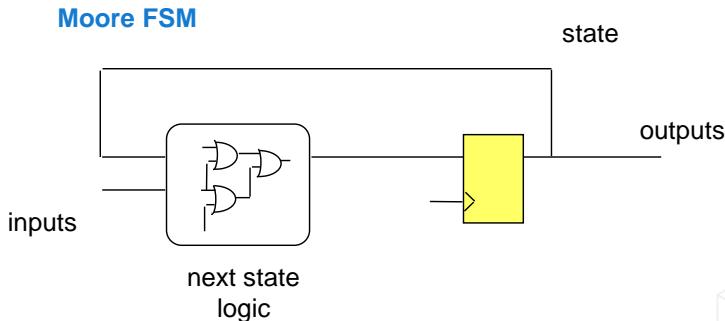
Best implementation for performance.

- Minimizes or removes output logic.

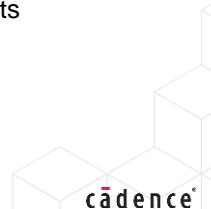
Harder to implement.

- Might not be possible for some designs.

Harder to maintain.



198 © Cadence Design Systems, Inc. All rights reserved.



Works better than other solutions

- Good area and performance
- Harder to implement
- Harder to maintain

An ideal solution for Moore machines may be to remove the output decode logic entirely by directly driving the outputs from bits of the state vector. This requires careful encoding of the state vector and may be hard, or even impossible, to implement and hard to maintain. Removing the output logic generates outputs on the right clock cycle without delay and gives the best performance.

Other State Encodings

Possible to choose any pattern of bits to represent state.

- May simplify the next state or output decode logic.

Common choices:

- Gray coding
 - Cyclic code
 - Only one bit changes per change of code
 - Low power and low noise
 - Best for predictable state transition patterns
- One-hot encoding
 - All bits zero except one non-zero bit
 - Good for output decode
 - High register count
 - N states requires an N bit state variable

	Binary	Gray	OneHot
	000	000	00000001
	001	001	00000010
	010	011	00000100
	011	010	00001000
	100	110	00010000
	101	111	00100000
	110	101	01000000
	111	100	10000000



One-hot

- Faster implementation at the expense of the area
 - State vectors are longer

Gray codes

- Prevent intermediate ‘spurious’ states from being generated
- Low power

Modifying the state encoding is a key method for optimizing an FSM by simplifying the next state or output decode logic, reducing spurious state transitions, or reducing power. There are several common choices. As previously mentioned, Gray code is a cyclic code where only one-bit changes from one value to another. This reduces power and noise. Even in an FSM with state-bit encoded outputs, clock skew between the state registers may mean the state bits don't all change at the same time. So, with binary encoding, switching from a state encoding of 3 to a state encoding of 4 may actually transition through the encoding of 1 and 0 if individual bit changes are not synchronized. As only one-bit changes in Gray code, intermediate encodings are avoided. However, Gray code works best when there is a predictable sequence of state transitions. One-hot encoding is a good alternative. With one-hot encoding, each state is allocated a specific bit, that bit is set for the state, and all other bits are zero. This gives a 2-bit change for any state transition, so minimizing power and noise in arbitrary state sequences. Output decode can also be minimized. The cost is a higher register count since we need a register for every state. So, 8 states need 8 registers with one-hot encoding, compared to 3 registers with binary. In register-rich FPGA technologies, one-hot encoding is the standard.

Complex FSMs

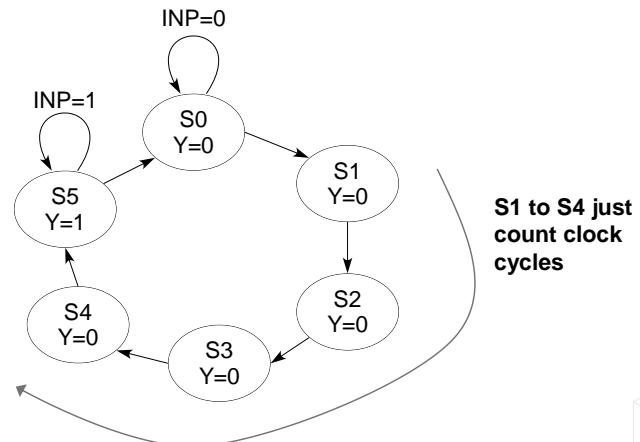
Embed data path elements in FSMs to simplify the design.

- For example, counters to count clock cycles.

The state diagram shows 4 states “delay” before output Y goes high.

- What if the delay changes to a 50-cycle delay?

Simplify state diagram by adding counter . . .



200 © Cadence Design Systems, Inc. All rights reserved.



FSM can contain many states that perform relatively mundane functions.

- E.g., waiting for N-clock cycles

Add datapath elements (e.g., counters) to reduce overall complexity.

The example shown adds a delay of 4 clock cycles.

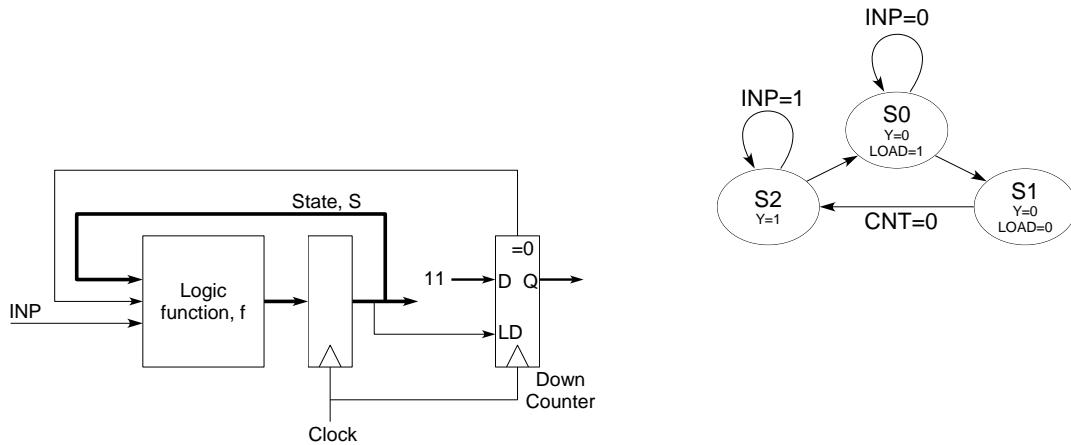
- Could be represented easily as a standard FSM.

However, if the delay were changed to 100 clock cycles, another 96 states would need to be added.

There are many other techniques to optimize FSMs besides state encoding. For example, it may be possible to simplify a complex FSM by embedding data path elements. The state diagram shows that when INP goes high, states S1 to S4 are unconditional transitions and simply count clock cycles until Y is set. A small number of delay cycles could be implemented with separate states, but what happens if we need a 50-cycle delay? We could use a counter to count cycles and simplify the FSM.

Simplified State Representation

Counter could be used at different places in the state diagram for different purposes.



201 © Cadence Design Systems, Inc. All rights reserved.



Counter is held in load state in S0.

When the state changes to S1, the load input is removed, and the counter begins to count down.

When count=0 the FSM moves to S2.

If a delay of 100 was required, the 2-bit counter could simply be extended to 7 bits.

We add an output to the FSM called a load, which is held high in state S0. When we change the state to S1, LOAD is released, and the counter starts counting down cycles. When the count is zero, we transition to S2 and set Y. We could use different load values in the counter at different places in the state diagram for different purposes. The counter reduces the number of states in the state diagram and simplifies the FSM.

Submodule Summary

In this submodule, you learned:

- Finite State Machines are the clearest and simplest way of designing control.
- Supported by a well-defined design process and synthesis.
- Can be augmented by adding datapath elements to process data or to provide explicit support for issues such as delays.



In summary, Finite State Machines are a clear, simple way of implementing control signals. They are supported by a well-defined design process with standard optimization techniques and dedicated synthesis tools. FSMs can be simplified by adding data path elements such as counters too, for example, process data or implement delays.

Test Your Understanding

- What are the three major functional blocks in an FSM?
- What is one-hot encoding?
- The outputs of a Mealy FSM are a function of what?
- Name a disadvantage and an advantage of using a Mealy FSM over a Moore FSM.



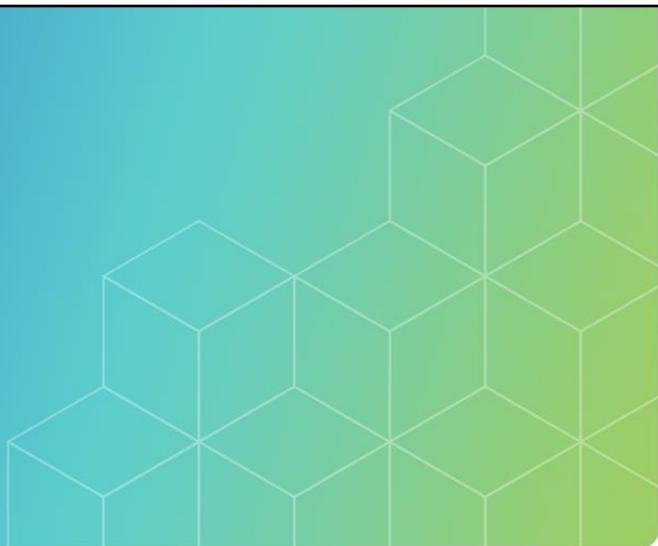
To test what we learned in this module, here is a quick exercise. Pause here, write down your answers, and check against the solutions.

The three major blocks in an FSM are the next state and output decode logic and the state register.

One-hot encoding is a pattern where only one bit of the state vector is set, and all other bits are cleared.

The outputs of a Mealy FSM are a function of the state and the FSM inputs.

An advantage of a Mealy over a Moore is that the Mealy one can usually be implemented in fewer states, as it does not need a separate state for every unique combination of outputs. A disadvantage of Mealy is that the combinational path from inputs to outputs has a high probability of introducing glitches in the FSM outputs.



Submodule 3-1-6

Memory Structures

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Identify various memory structures and organizations.

Topics include:

- Memory compilers
- Memory organization
 - Synchronous memory
 - Dual ported memory
- LIFO and FIFO
- Modeling memory

205 © Cadence Design Systems, Inc. All rights reserved.



In this submodule, we will identify various forms of memory structure and their organization. We will look at memory compilers and memory organization for synchronous and dual-port memories. We will explain LIFO and FIFO blocks and finish by examining memory caches.

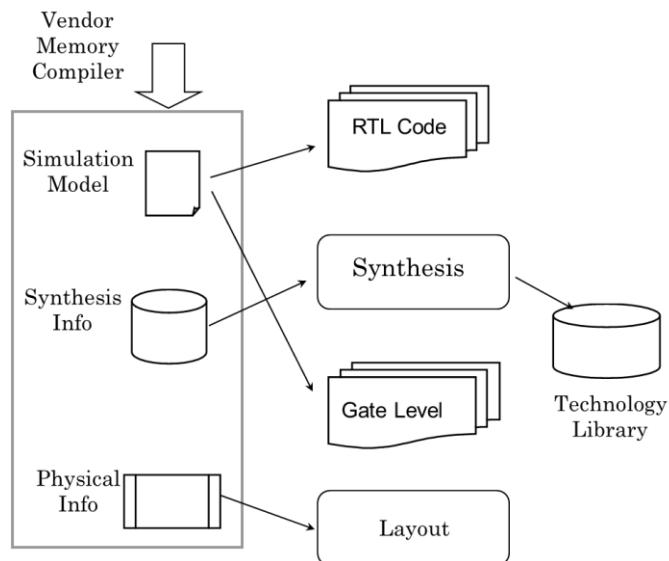
Memory Compilers

Synthesis tools cannot generally infer memory blocks from code.

- Memory must be instantiated as a component.

Models generated by vendor-specific memory compiler.

- Not very portable – tied to a vendor.
- Technology must be selected before models can be generated.
- Models will need to be changed for new technology.



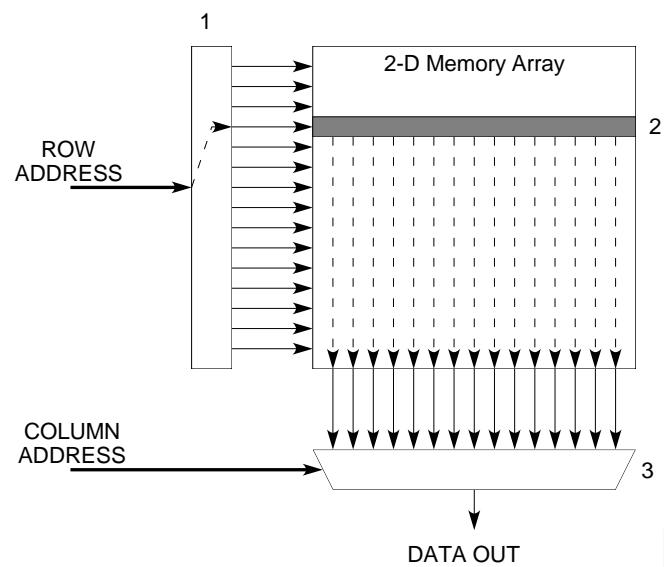
206 © Cadence Design Systems, Inc. All rights reserved.

cadence®

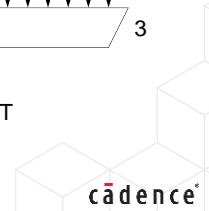
Synthesis tools cannot generally synthesize memory blocks directly from HDL code. Memory must be created as a separate component block. For verification and simulation, the memory component is mapped to a high-level behavioral model. For synthesis, we need some memory timing information, but the memory itself is excluded from synthesis. In layout, we replace the memory block with a prebuilt vendor implementation. So, we need memory models for RTL and gate-level simulation, synthesis timing information, and a layout implementation. All these models are generated by a vendor-specific memory compiler. The models are vendor and technology-specific, so you need to select these before generating the models, which limits the portability of your code. If the technology changes, we need new models.

Structural Organization of Memory

- Half of address is decoded to select a whole row of “cells” in the array.
- The data from the row propagates to the edge of the array.
- The other half of the address is used to select an individual datum.



207 © Cadence Design Systems, Inc. All rights reserved.



Memory is organized as a 2-dimensional array of data. To access an individual location, half the memory address is decoded to select a row in the memory array. This row of data propagates to the edge of the array, where the other half of the address is decoded to select an individual data item.

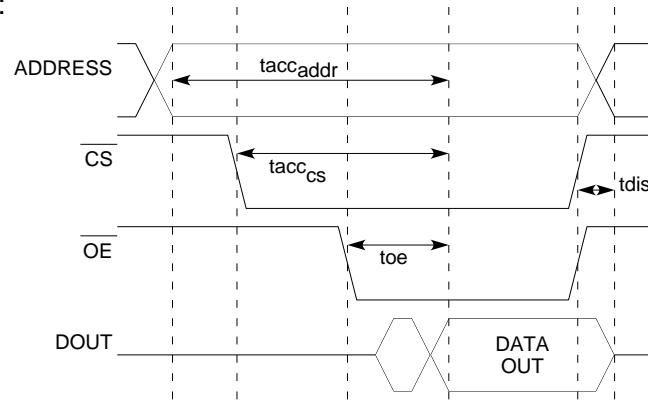
Memory Timing

Accessing is relatively slow

- Memory designed for density, not for performance

For example, read access time is defined for:

- Change of address
- Change of device select (CS)
- Change of output enabling (OE)



208 © Cadence Design Systems, Inc. All rights reserved.

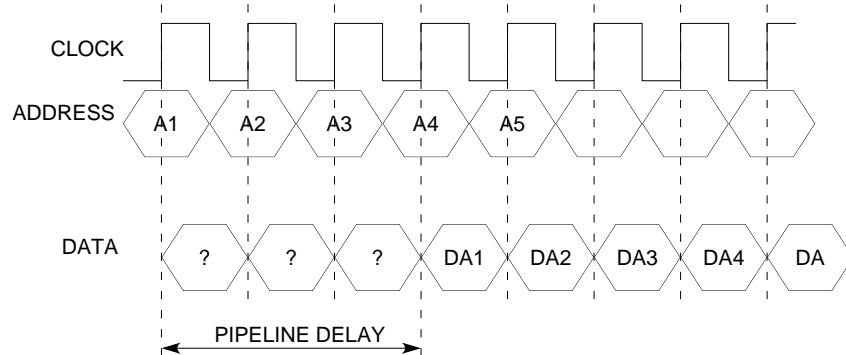


Memories are built for storage density, not for performance, therefore, memory access is relatively slow. Vendor timing diagrams define the timing of various access scenarios. For example, a read operation where the address changes or the device selects, or the output enables. We may need additional logic around a vendor-specific memory to synchronize the vendor-specific timing to the timing used by the design.

Synchronous Memory

Synchronous memories use clocks.

- Synchronizes memory with the associated processor.
- Memory throughput can be improved.
 - Especially for contiguous memory accesses, e.g., cache transfers.
- Introduces “pipeline” delays.



209 © Cadence Design Systems, Inc. All rights reserved.



Memory access can be simplified using synchronous memories, where memory operations are synchronized to the system or associated processor clock. This allows memory throughput to be improved, particularly for accessing blocks of data in adjacent addresses. Synchronous memories have internal register banks on address and data buses. The downside is that synchronous memories introduce pipeline delays due to these internal register banks. For example, read data will not be available in the same clock cycle as the address is applied but in a future clock cycle, by which time the address input has moved on. Here the pipeline delay means the data for address A1 is not output until 3 clock cycles after A1 is written to the input, by which time we are applying address A4. Many standard bus protocols use pipelining to take advantage of synchronous memory.

Modeling Synchronous Memory

Memories often have a complex instruction set.

- Internal counters to transfer data in contiguous blocks.

Each stage of an access should be pipelined for the best performance:

- Row decoder
- Array access
- Final data multiplexing

Access time:

- Within a row, one datum per clock cycle after latency.
- Between rows, latency can be hidden.



Synchronous memories often have complex instruction sets. For example, to allow burst mode operations that access memory in blocks of 8 or 16 addresses. A burst mode access may provide a single address, and internal counters automatically access a set number of locations from the address. For the best performance, every part of the memory organization, row access, data propagation, and output data MUXing can be pipelined with register banks. By storing a whole row of data in an output register bank, we can easily read a block of data from the row by simple indexing. A synchronous memory may also preload data from adjacent rows into separate output register banks in anticipation that the addresses of a memory access will spread over several rows. This hides the normal row access delay due to data propagation.

Dual Port Memory

Memory can be designed to support two separate ports:

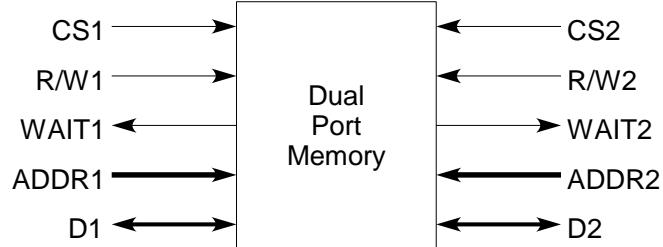
- Inherently at the transistor level.
- “Time-sliced” access to a normal memory array.

Ports may be different:

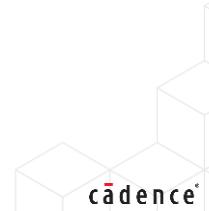
- Both read/write.
- One write, one read.

Control of access is important.

- Simultaneous access to a single location from both ports can give indeterminate behavior.



211 © Cadence Design Systems, Inc. All rights reserved.



In some designs, dual port memory is more useful than a single port. Dual port memory has two separate sets of access signals, such as address, data, read/write, etc. Support for the two ports can be added to the memory at the transistor level, or we can time-slice the two access ports to a normal single port array. Both ports don't have to be read/write (although that is common). A dual port implementation is simpler if the design needs one port to be write-only and the other read-only. The key issue in implementation is to prevent simultaneous access to a single location from both ports. Simultaneous access can lead to indeterminate behavior due to race conditions. For example, the result of writes from both ports to the same location depends on which write completes first. Access may be easier to control in a time-sliced normal memory rather than a transistor-level implementation.

Stack: Last-In-First-Out Memory

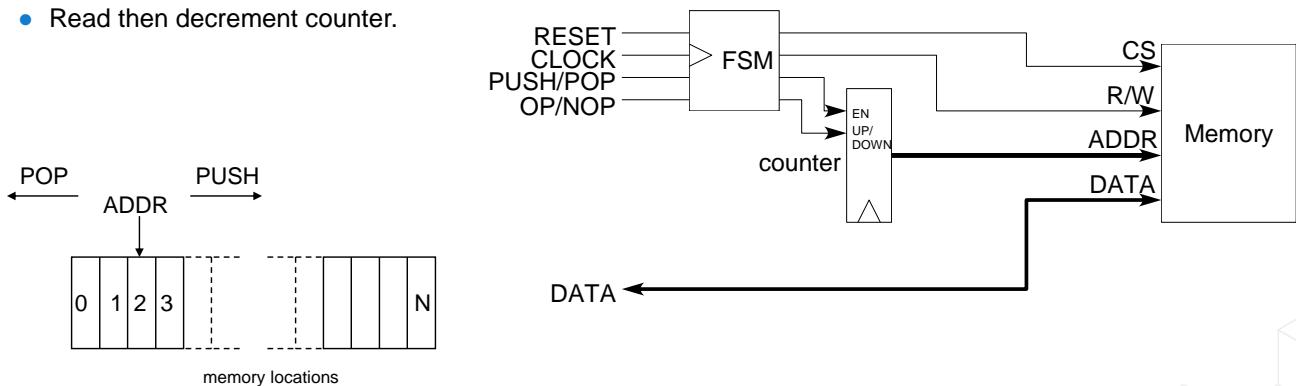
Implemented with an up/down counter.

PUSH

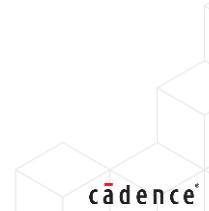
- Increment counter then write.

POP

- Read then decrement counter.



212 © Cadence Design Systems, Inc. All rights reserved.



Another useful memory-based component is a stack, or a Last-In-First-Out (LIFO) memory. Data written to a stack is stored in order of writing, and a read accesses the last data written. With a memory-based stack, the memory generates the address using a counter which can be incremented or decremented. A stack write is called a push, and this increments the counter and writes to the memory at the counter address. A stack read, or pop, reads the memory at the current counter address and then decrements the count. So, the counter provides the address to the top of the stacked data. We need limits on the counter to prevent overflow at the end of the address space and to detect error conditions like a pop from an empty stack (count = 0) or a push to a full stack (count = max).

First-In-First-Out Memory (FIFO)

Used as delay, queue, or “rate buffer.”

- Data written in sequentially at tail of queue.
- Data read sequentially from head of queue.

Implemented as memory or shift register.

Shift Register FIFO suffers from “fall through.”

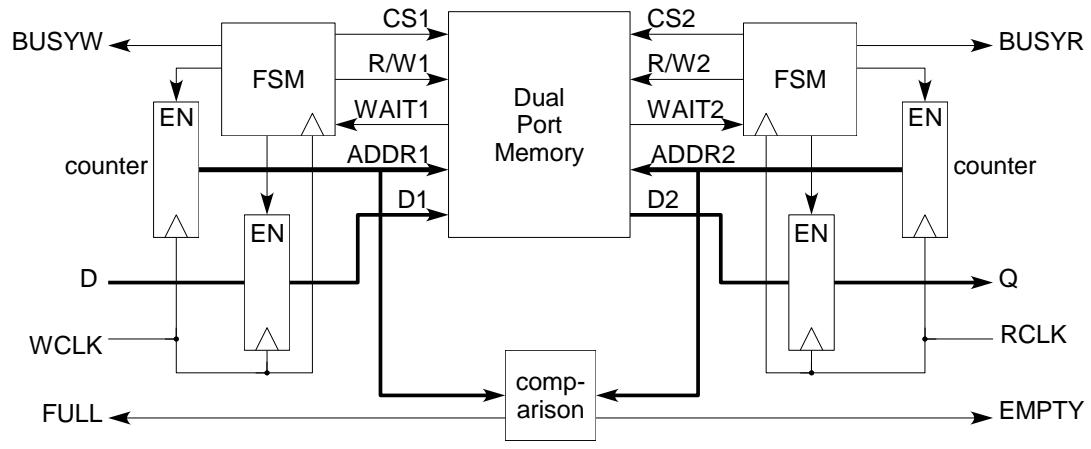
- Time taken for the data to propagate through the FIFO to the end of the queue.
- Can be used as a delay element.
 - Fixed time (clock cycles) to pass through the FIFO.



A variant of the stack is a First-In-First-Out (FIFO) memory. Also called a delay, queue, or rate buffer. Data is still stored in a FIFO in write order, but in a FIFO, we write data to the tail of a data queue and read data from the head. So, data is read in the order in which it was written. A FIFO could be implemented using a memory block or a shift register, a block of serially connected registers. A shift-register FIFO only passes data on a clock edge; therefore, for example, if you push data to a shift register FIFO of depth 8, it will take 8 clock cycles for the data to propagate through the registers before it can be popped from the output. This is called fall through time. The shift register FIFO is excellent as a delay element, where you want to store data for a fixed number of clock cycles before processing it.

Synchronous FIFO (Dual Clock)

Based around dual-port memory.



214 © Cadence Design Systems, Inc. All rights reserved.



A memory-based FIFO can be constructed around a dual port memory, with one write port and one read port. Address generation is complicated, as we need separate tails to write and header-read addresses. So, we have separate address counters for push and pop. The addresses follow each other through the memory address space, with the push address leading the pop. The addresses must be able to wrap round from maximum to minimum. We can determine the status of the FIFO by comparing the addresses. For example, if the write address points to the next free location (i.e., write then increment), then if the read is one less than the write address, the FIFO is empty. If the read and write addresses are the same, the FIFO is full. To avoid losing data, a full FIFO should not be written, and an empty FIFO should not be read. The FIFO may need additional logic and full and empty status outputs to prevent this.

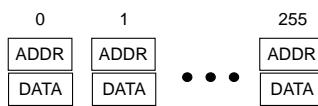
Static Cache Memory Modeling

Only a small percentage of memory may be used.

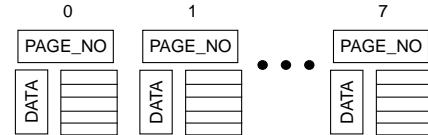
- Can be modeled as element or paged cache:
 - Element caches implement a fixed number of individual memory locations.
 - Paged cache implements fixed-size data blocks.

Maximum number of memory locations used must be known.

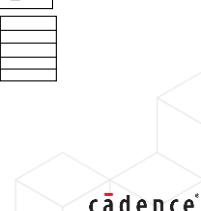
64Kx16 RAM as 256 element cache



64Kx16 memory as 8X4K paged cache



215 © Cadence Design Systems, Inc. All rights reserved.



If memory is sparsely used, design memory models as caches to only implement the number of memory locations required.

Element caches implement a fixed number of individual memory locations.

- On read or write, use a loop to read through the addresses until the required location is found.

Paged cache implements fixed-size data blocks.

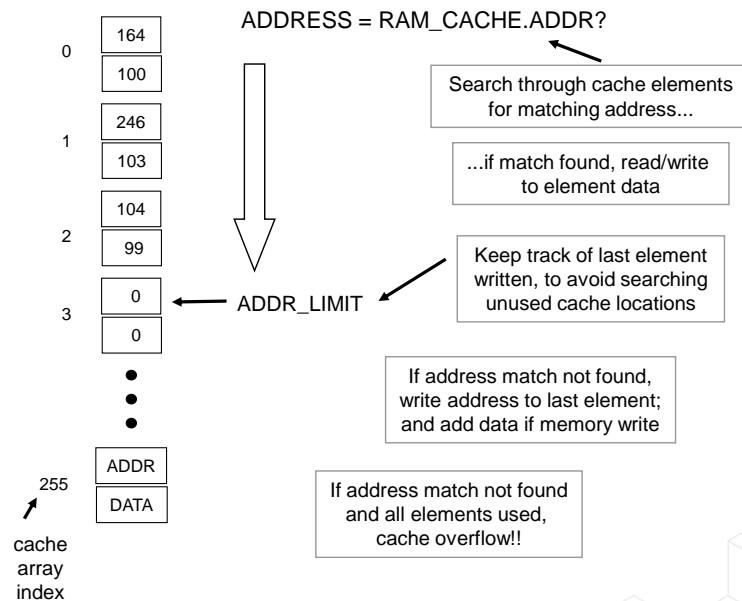
- On read or write, use a loop to read through the addresses until the required address range is found, and index data block to access the required location.

Some designs may only use a small percentage of the available memory space, and addresses used may not be contiguous. Instead of implementing the whole memory, a design could implement only the locations used as a cache. An element cache implements individual memory locations. The address and data are stored together as one entry in the cache. It may be more efficient to create a paged cache that stores fixed-size blocks of data with a page number indicating an address range and the individual address-data pairs stored in the page block. Caches have a fixed size which limits the number of elements or pages that can be stored, so cache management is a necessity. Caches are also used in processor designs to implement fast, local storage of frequently used memory locations, to reduce main memory access delays.

Element Cache Model (Static)

Questions that an implementation will need to address:

- What should happen on a cache overflow?
- What should happen for a read on an un-cached address:
 - Do we create a cache location?
 - What is the output data value?



216 © Cadence Design Systems, Inc. All rights reserved.



ADDR_LIMIT keeps track of the last cache array element written and saves time searching unused locations.

With the new address input, search from the first location to ADDR_LIMIT to find a matching address.

- If found, read or write to or from an element data field.
- If not found, then If $ADDR_LIMIT \leq$ maximum cache size, write address to ADDR_LIMIT location and write data.

What should we do for a memory read to an unwritten address?

- If $ADDR_LIMIT >$ max. cache size, we have cache overflow, and the data can be lost.

We need to know the maximum cache size in advance or use a method of dynamically growing/shrinking the cache size during simulation.

In summary, there are many different memory architectures for different design applications. Memory cannot generally be synthesized, so we need a vendor-specific memory compiler to provide the simulation, synthesis, and place and route models for the design. We can add additional logic around basic single or dual port memories to implement stack or FIFO structures.

Submodule Summary

In this submodule, you learned:

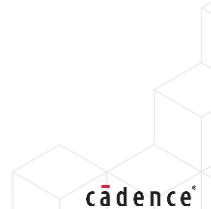
- There are different memory architectures that can be exploited in different applications.
- Memory cannot generally be directly synthesized.
- Basic memory blocks can be generated efficiently by compilers.
- Interface logic can be added to basic single/dual port memories to build more complex memory structures:
 - FIFO
 - Stack



This page does not contain notes.

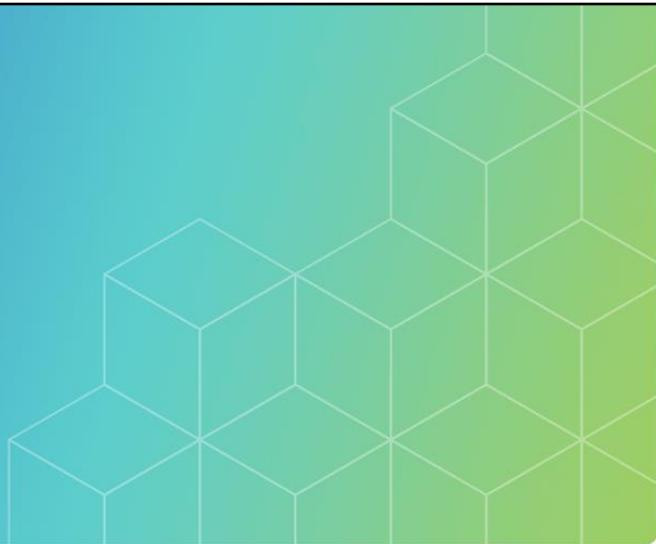
Test Your Understanding

- What are the three stages of memory access as described in the notes?
- What is “fall-through” time?
- Name three applications of a FIFO.



To test what we learned in this module, here is a quick exercise. Pause here, write down your answers, and check against the solutions.

The three stages of memory are address decode for row selection, row data propagation to the memory edge, and address decode for the data output selection. Fall-through time is the delay in a shift-register FIFO for the input data to be clocked through the register array to the outputs. Applications of a FIFO include rate-buffer, queue and delay.



Submodule 3-2

Functional Design Challenges

cadence®

In this section, we're talking about the challenges with functional design.

We've already learned a little bit about the different kinds of hard hardware constructs that we see and the different levels of abstraction at which we have to write code.

In order to describe those, this section is going to talk about some of the problems you'll find.

Submodule Objective

In this submodule, you will:

- Identify the physical factors that are present challenges to functional design.

Topics include:

- Recap on scale and complexity of IC design
- X-Propagation
- Clock domain crossing
- Reset domain crossing
- Low-power requirements – clock gating
- Introduction to low-power schemes (power intent using UPF language)



Your objective is to get started using SystemVerilog to describe the behavior of a digital design. To do that, you need to know some fundamental SystemVerilog language constructs.

Mainly they're related to physical factors that present challenges to the functional design.

Although we're coding at the lower level of abstraction register transfer level, we can't ignore the physical factors which will occur later on in the process.

So, we can see here a list of the topics we talk about.

And following this will be an introduction to low-power schemes, which use a language known as UPF.

Recap On Where We Are

We have seen how to create functional building blocks from boolean gates and flip-flops (FFs).

- For example, multiplexors, adders and so on.

An idea of scale:

- It takes 4 transistors to make a boolean NAND gate.
- It typically takes 18-20 transistors to make a FF.
- A complex processor chip might have 19 billion transistors.



For describing power, architecture, and power. We've seen how to create these functional building blocks from boolean gates and flip-flops.

These are all registers of how we describe multiplexes and others and multipliers and so on.

So, to give some idea of the scale of how this actually works in the physical world, it takes four transistors to make a boolean NAND gate.

And the brilliant Northgate, for that matter. In order to make a flip-flop. It takes something in the order of 18 to 20 flip-flops, depending on what kind of flip-flop it is.

We've already seen that's a complex process and might have anywhere approaching 20 billion or 30 billion transistors.

How Can We Fill a Chip?

By defining a huge number of boolean gates, FFs, latches, memory cells, and so on.

We would not have time to manually create schematics with FFs and boolean gates.

- It's been well over 30 years since IC design was done like that.

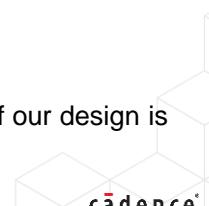
We have to write software-like code, known as RTL (Register Transfer Level) code.

- We will explain RTL in the forthcoming lectures.

EDA tools translate (synthesize) our RTL code into a schematic.

The complexity of the design requirements is truly mind-boggling.

However, stating the huge amount of RTL code required to define the desired functionality of our design is not the only concern.



That's a huge number, of course. So how can we actually fill this space? How can we create a design that contains that many transistors?

Well, we can't do it manually using schematics. Drawings, because it will take too long to do so.

The design of silicon IC used to be done like that years ago because that was the only option.

But now, because you can get so many more transistors into the same area.

Exponential increase following Moore's Law.

Now in 30 years, the increase is absolutely massive, and it's just too big. You cannot possibly create a schematic that would fill today's chips.

So, what we have to do then is write code at the registered transfer level, RTL level, and we're going to explain that in very shortly in the forthcoming lectures.

And what our tools do, our electronic design automation is what an EDA stands for.

This translates our code into a schematic effectively because underlying all of this will be a schematic created on whatever target technology we're using, whatever silicon process we're using.

So, the complexity of the design requirements is truly mind-boggling when you think about what is a graphics processor required to do or what is a general CPU required to do.

Just stating that a huge amount of RTL code is required to fulfill that functionality is not our only problem.

Remember What's Important to Optimize in an IC

PPA

- The trade-off between power, performance, and area.

We may choose to address some of these aspects in our RTL code.

We now have to consider more than just the functional requirements specification when implementing RTL.

Power

- For example, describe clock gating at the RTL level using Gray coding for FSM's state encoding.

Performance

- For example, choose to re-code, in RTL, one slow implementation of a multiplier into a faster (more gates, more parallelism and less delay) implementation.

Area

- For example, share resources, like an adder, in the time domain to reduce overall transistor count.



This is the topic of this lecture. So forecast our minds back to when we talked about what's important in an IC economically. It's always a trade-off.

Everything is a trade-off between power, performance, and area. We can address some of those aspects with how we write our RTL code. When we're writing RTL code, we're not just considering what the functional requirements are. We're also thinking about some of these PPA, which are physical aspects of the design. So, for example, how do we address power?

We could describe clock gating at the RTL level and use different kinds of encoding for state machines. When creating multipliers, there are different architectures and multipliers we can create, which will have fewer changes or fewer gates in order to have lower power consumption. This all has to be handcrafted manually. So, for performance, we might choose to recode an RTL with the different implementations of a multiplier. You can't just say multiply by B if you want a power-efficient design; you have to go manually and create different kinds of multipliers depending on what your requirements are for the area.

We can share resources like memories and others and so on in the time domain in order to reduce the overall transistor count. So all these things are pulling in different directions, and we've got to find the sweet spot in order that we get the most economical design.

What Physical Issues Might Affect Our RTL Code?

Our RTL code is defined by the desired functionality of our IC in a software-like style.

However, we cannot write RTL without being aware of physical issues.

- One doesn't have this concern when writing "normal" software.

These are related to the underlying physical technology, namely:

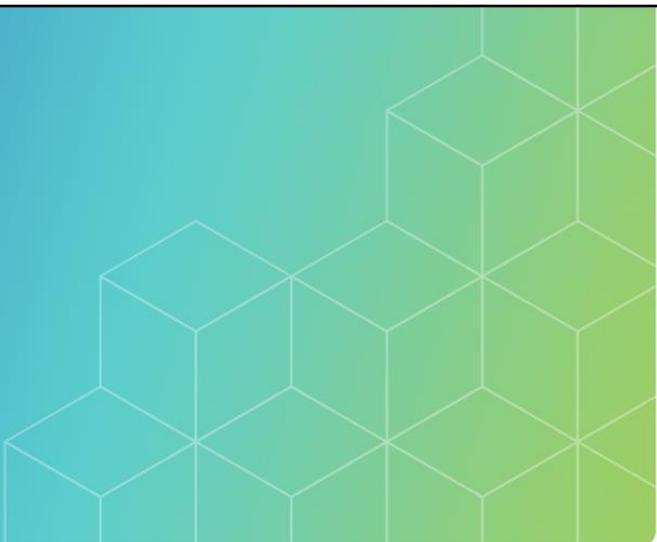
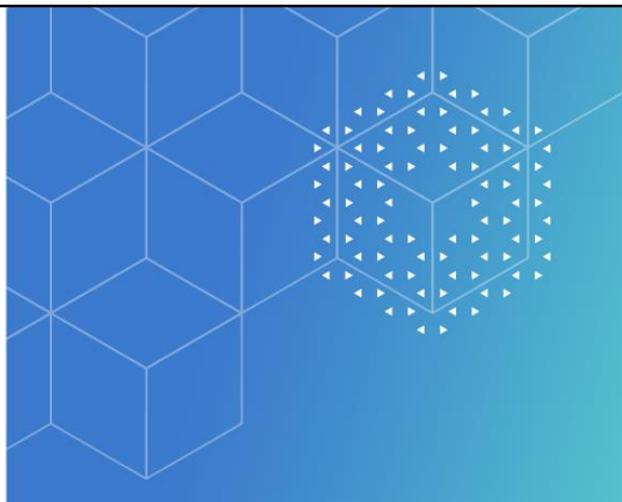
- **X-Propagation** – Unpredictable behavior due to timing and physical issues.
- **Clock Domain Crossing** – Unpredictable behavior due to timing and physical issues.
- **Reset Domain Crossing** – Unpredictable behavior due to timing and physical issues.
- **Low-Power Considerations** – Some aspects of low power we have to code into our RTL.



What physical issues might affect the code we write down? The code is, is software like it isn't software; of course, it's software-like. Because it's actually describing real-life hardware which has to care about things that software doesn't like concurrency, for example, time delays and glitches and all these kinds of things, we can't just write RTL and be unconcerned about those fiscal issues. If you're writing normal software, you don't have that consideration. You can just focus on the functional requirements.

There are some things related to the underlying physical technology. Now you have to create a chip in real life. These things are X propagation. So that is unpredictable behavior due to timing and physical issues: clock domain, crossing, reset, domain crossing, and other low-power considerations.

Some aspects of low power we can address in RTL. However, we said previously that UPF language, which describes low power intent, is another aspect of design we have to take care of, and we talk about UPF later. We're not going to describe these things now.



Submodule 3-2-1

X- Propagation

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Identify the different aspects of X-Propagation.



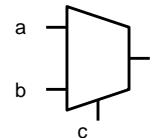
This page does not contain notes.

What Is an X-Value?

- In simulation: Value not known (could be 0 or 1).
- In synthesis: As a literal, designates value as “don’t care” (make it 0 or 1).

Where does the X value come from?

- Initial uninitialized value – a flip-flop without a reset.
- Unresolvable conflict of equal-strength 0 and 1 values.
- Unknown inputs.
- Deliberate assignment in RTL code:
 - In design code to indicate “don’t care.”
 - In verification code to indicate an error.



```
// RTL X-optimism
if (condition)
  result=a; // if c true
else
  result=b; // if other
```



- The SystemVerilog language designates the “X” literal to mean an unknown value.
- The unknown value initially appears on 4-state variables until the simulation assigns another value. During simulation, the unknown value can appear on nets simultaneously driven to equal-strength conflicting values and can appear on variables when so assigned.
- RTL simulation can mask those “X” values by converting them to another value, typically 0 or 1. This is because the synthesizable RTL branch statement has only “true” and “other than true” branches and no branch specifically for an “unknown” condition. The term for this is “X-Optimism.”

Documentation: *Elaborating Your Design Starting with X-Propagation*

The first question to ask is, what is an X value? Then we thought digital hardware was zeros and ones, but it's not real. Life is not that simple. An X value is an unknown value in simulation. The value could be a zero, or it could be one we don't really know. We can't be sure in synthesis, which is the act of taking the RTL code and building this netlist from whatever technology library we're using. This is used as I don't care, i.e., the synthesis tool can make it a zero or one just to make a more efficient design.

So, it means different things depending on which tool you're using. You might wonder, where does this come from then? How can we have a simulation where we have a value that isn't a zero or one? Well, it can come from various places, typically on an initialized value, like a flip-flop without a reset unresolvable conflict between two equal-strength drivers. You're driving a wire from two different places, and they have different values and unknown values of inputs. Deliberate assignments in RTL code for different reasons. So, in the design code, it indicates don't care.

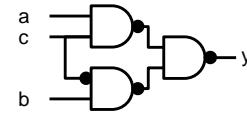
As we mentioned above, what we're synthesizing in verification code, what we're hoping is this X-propagates. And we see lots of different axes in our simulation, which tells us something's gone wrong. So how does this occur, then? If we have this condition here, if we're describing a mocks with this code here if this condition is true. The result is the value of A and the other condition. Then the result is B. If this condition gets an X value and an unknown value, the result will always be if we follow what this code says.

The X-Optimism Problem

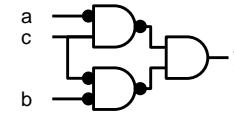
X-Optimism

- RTL simulation can convert unknown values to known values.
- User debugs problems at the gate level with difficulty, which could be more easily debugged at RTL.
- Evaluation of synthesizable RTL branch statement has only “true” and “other than true” branches – no branch specifically for “unknown.”

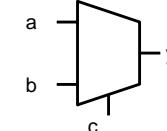
```
// RTL
if ( c )
    y = a; // c true
else
    y = b; // other
```



Implementation A



Implementation B



Implementation C

a	b	c	:	y
0	0	x	:	0
0	1	x	:	1
1	0	x	:	0
1	1	x	:	1

Known values

a	b	c	:	y
0	0	x	:	0
0	1	x	:	x
1	0	x	:	x
1	1	x	:	x

Unknown values but 3 different results!

a	b	c	:	y
0	0	x	:	x
0	1	x	:	x
1	0	x	:	x
1	1	x	:	1

a	b	c	:	y
0	0	x	:	0
0	1	x	:	x
1	0	x	:	x
1	1	x	:	x

228 © Cadence Design Systems, Inc. All rights reserved.



RTL simulations can mask unknown values by converting unknown values to known values. This X-Optimism can defer detection of design problems to a later post-synthesis simulation, where their debug is more difficult.

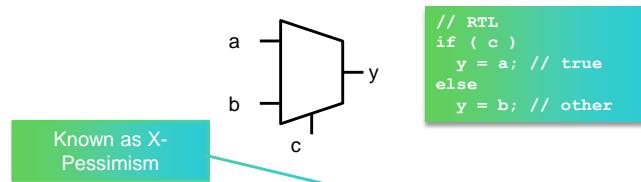
The illustration shows that the RTL model and three potential implementations all produce different output values when an unknown input value is given. The corollary *X-Pessimism* occurs when gate-level simulation produces an unknown value that the actual hardware does not produce. The first two implementations show this X-Pessimism with respect to the third. This is known as X-Optimism. So, if we have a control in our mocks, go back to the diagram here. This is the control for the mocks. If this is an x value, then the output merely follows B all the time. As we can see from this table, these are known values, which is overoptimistic because we don't really know what this value C is in real life. So, the dangers that are it can be difficult to debug problems at the gate level. This is when you've created this gate-level netlist from synthesis. It isn't easy trying to correlate that gate-level description, that level of abstraction, with your RTL code.

Now what we've got in RTL simulation, if we just follow what that code says, is that we can convert unknown values into known values. For all the output here, Y is always a known value. Suppose we get any problems at the gate level. So, the gate level is a different level of abstraction. It's after we've taken our RTL and converted it to this gate-level netlist for synthesis. It's harder to debug that way. RTL is easier to debug.

So, any evaluation of the branch here, we're evaluating that. The branch statement is only true, and something other than true, including X. There isn't a branch for what happens if the book selection is an X. Moreover, we might get more different implementations in real life.

We might have three different implementations of this mark, all of which give us a different set of results given the same inputs if C is X. This is clearly a problem. The outcome depends upon the implementation which gets chosen.

X-Optimism Simulation Solution



Default

If the condition is unknown, assign the result as stated.

Only FOX guarantees that the simulation has no *new* unknowns.

```
// Default
a b c : y
0 0 x : 0
0 1 x : 1
1 0 x : 0
1 1 x : 1
```

Forward-Only-X (FOX)

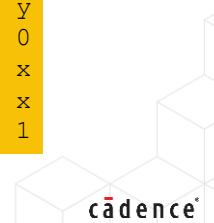
If the condition is unknown, assign the result unknown.

```
// FOX
a b c : y
0 0 x : x
0 1 x : x
1 0 x : x
1 1 x : x
```

Compute-As-Ternary (CAT)

If the condition is unknown, assign the result as the hardware would.

```
// CAT
a b c : y
0 0 x : 0
0 1 x : x
1 0 x : x
1 1 x : 1
```



The Xcelium simulator solves the X-optimism problem by providing three modes in which to evaluate the branch statement when the condition is unknown.

- The simulator, by default, executes procedural assignments exactly as stated, treating the unknown condition as other-than-true.
- The simulator in forward-only-x mode, if the condition becomes unknown, assigns the unknown value to the variables. Only this mode can guarantee that the branching construct in later gate-level simulation produces no new unknown results.
- The simulator in compute-as-ternary mode, if the condition becomes unknown, it evaluates both the true and the false paths. It assigns the variables for the merged results from both paths. This is equivalent to the operation of actual hardware.

Why X's Are Dangerous

X's might mean that the behavior we observe in RTL simulation is not the same as seen in the real hardware.

X's may propagate to outputs.

- If one were purchasing design IP, then it would probably be a contractual requirement that this cannot happen.

X in clock gating logic or low-power control can hang a whole chip.

Makes simulation less deterministic:

- Different simulators produce varying results with X.
- X-Optimism, X-Pessimism issues.
- Gate-level simulation comparison with RTL simulation is hard with X in the design.

X can be misleading for code coverage and equivalence checkers.



So, why are these dangerous, then? Why don't we want this access? Access might mean the behavior we observe in RTL is not the same as the real hardware. Exhibit X might get two outputs.

Normally, if you're selling design IP to somebody, it's almost certainly a contractual requirement that access cannot propagate to outputs. Because when you integrate different IPs together, you can't guarantee something won't go wrong, something you didn't foresee.

If you have an X in clock gating or low-power control that can kill the whole chip, you can't recover from that with software fixes or metal layer fixes, and it makes simulation less deterministic because of all these different modes we have Fox and cats, and different simulators might produce different results.

With X, we must decide whether we use X-Optimism or Pessimism. Level simulation comparisons with RTL are hard with an X in the design. In addition to this, X can mislead code coverage checkers and equivalence checkers. These are formal kinds of proofs, so basically, they're bad news all around.

We Need to Make Some Choices

Can't we just reset all FFs so we don't get Xs?

That would be too costly in the area:

- Resettable FFs require more area than Non-Resettable FFs.
- It takes more space (area) to route the reset signal to each FF.

Which mode do I choose when simulating?

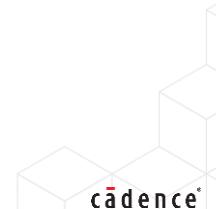
- CAT – Too Optimistic?
- FOX – Too Pessimistic?

Is there an alternative?

- Yes. We could use Formal Verification (algebraic proofs that don't require a testbench).

What's the difference regarding X-Propagation?

- In a simulation, each signal can only have one value at any given time – either 0 or 1.
- In formal, all possibilities are considered concurrently – both 0 and 1 at the same time.



We've got some choices to make here. Can't we just reset all the flip-flops so we don't get Xs in the first place? Well, the answer is no, we can't, because it costs too much in terms of area.

A flip-flop with the resets is bigger in terms of area than a flip-flop without one. You've got to root the reset wire to a reset, a flip-flop that takes more area on the chip as well. And an area was one of these trade-offs we make PPA; the E stands for the area.

In any case, what do I choose when simulating? Do I choose a cat or a fox? If there's a choice of two things, it tells you there is no kind of simple choice to make here. What's the alternative, then? Well, we could use formal verification. Formal verification uses algebraic proofs, which don't require a testbench. You know, the X-Propagation, whether it's a problem or not under all circumstances.

The difference between formal and simulation is that in simulation, each signal can only have one value at a given time, either zero or one in formal. All possibilities are considered. In formal, X means a set of values that could be zero or could be one, and simulation x is literally the discrete value one to be X. Although we're modeling digital hardware, which we expect to be zero or one, there is a different literal value, x, in formal something. If it's X, it means it can be the value zero or the value one.

What X Means in Formal

X does not exist in formal, just like in hardware.

- In the simulation, X is the discrete value 1'bX.

In formal, a signal is either 1'b0 or 1'b1.

If a signal's value is unknown, then it is a possible set of values {0, 1}.

- You can pass this through an inverter, which will be the set of values {1, 0}.
- It behaves exactly as the real hardware would.

The concrete value chosen will be the one that causes an assertion to fail or a cover to pass.

X-pessimism or X-optimism does not exist.

Cadence® JasperGold® has an X-Prop App dedicated to this exact problem.

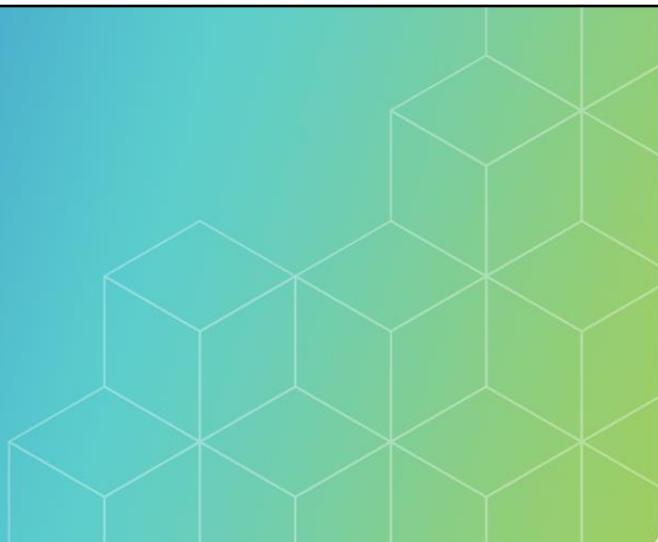
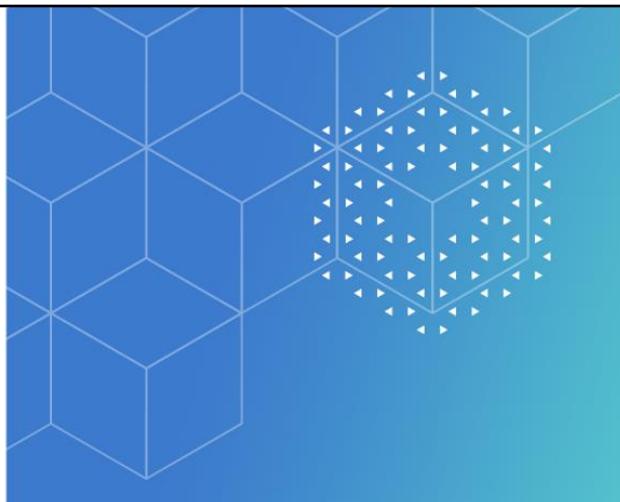
The beauty of formal is that you know whether X will cause a problem under all circumstances.

- Not just a particular set of simulations, with a particular CAT/FOX mode setting.



The cool thing about this is that if you pass that set of values through an inverter, you get the set of values one and zero. It behaves exactly like the real whole world and the concrete value you see, so it has to choose from this set what the actual value is. We find this out during a counter-example.

If an assertion fails or a cover statement passes, we get to see what the real value is. You don't get this concept of pessimism or optimism. It behaves like the hardware does. A formal tool from Cadence has an X-Prop app dedicated exactly to this problem because it's a problem that everybody has. And the beauty of that is, you know, whether X will cause a problem under all circumstances, not what you happen to simulate during all of your different tests with whatever setting you had, whether it's computers, ternary or forward, only X.



Submodule 3-2-2

Clock Domain Crossing (CDC)

cadence®

Clock domain crossing is another aspect that is affected by the physical design itself.

Submodule Objective

In this submodule, you will:

- Identify the different aspects of Clock Domain Crossing in your design and its impact.

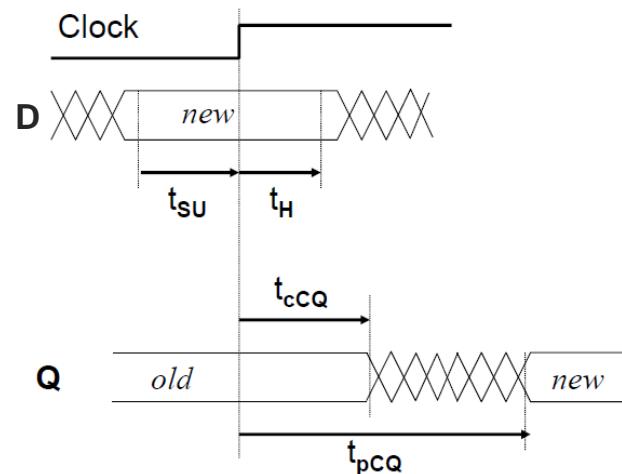


This page does not contain notes.

Timing Violations

FF Timing Specification

- Input data (D) must be stable.
 - Before active edge – setup time (t_{SU}).
 - After active clock edge – hold time (t_H).
- Propagation delay is the time from clock edge to the change in output (t_{PCQ}).



What happens when this specification is not followed?

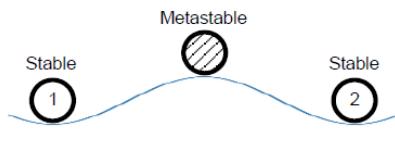
We've seen in a flip-flop that when the clock arrives, the data input can only change within a certain window. It cannot change between that line there and this line here. So the gap between the clock edge and the time prior to that where you can make the last change to date is known as setup time. And the gap between the clock edge and the earliest, at which you can then change the data in what is known as the hold time the propagation delays. The time from the clock edge to the output is actually changing.

So, when you have a data input, the clock arrives, the output will bounce up and down for a bit before it settles to a new value. What happens if we don't follow that specification?

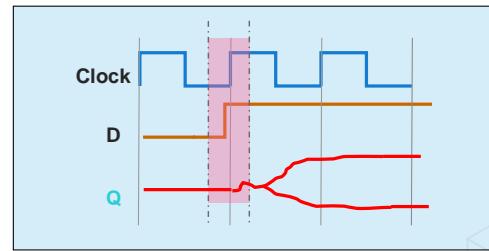
What Is Metastability?

When setup/hold conditions are violated, the output of a FF becomes unstable and unpredictable.

- Output may settle either way (1 or 0) after an unpredictable delay.
- This physical phenomenon is known as metastability.
- Seen for asynchronous inputs (clock to data relationship is unknown).
- In a gate-level simulation, a metastable value is represented as X.



236 © Cadence Design Systems, Inc. All rights reserved.



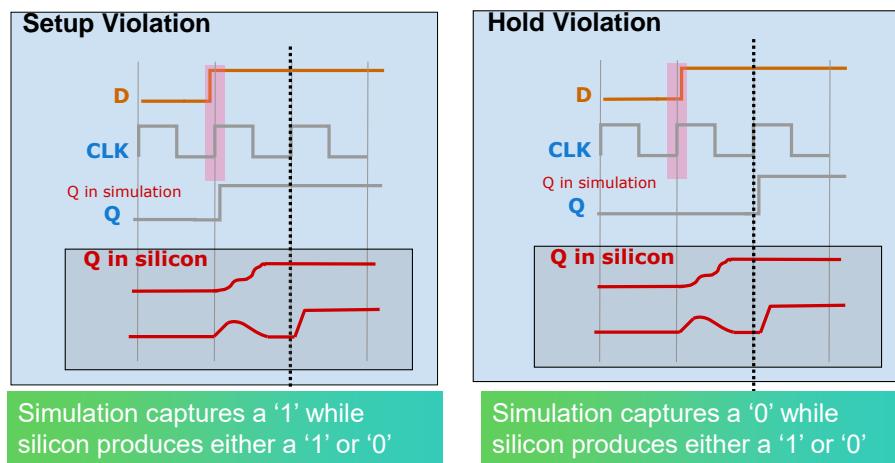
cadence®

This is known as a timing violation.

A setup or hold time is violated. The output of the flip-flop becomes unstable. We can't predict whether it will be a zero or whether it will be one, or whether it will be somewhere in between. This is a physical phenomenon. This is due to the physical aspects of the chip. This is known as metastability.

Here, these are arriving too close to this rising clock edge that the red band represents the time we shouldn't change the value. It does change, however, and the output Q drifts off either to be a one or drift to be a zero after some time. So, this is a stable value. That's where that's represented as X inside of a simulation.

Metastability Modeling



Both the setup and hold effects must be modeled in order to detect all metastability related bugs.

237 © Cadence Design Systems, Inc. All rights reserved.



How do we model these things? In the real silicone here, if we violate those set up at all times, we can see we might get different kinds of outputs.

We've got a model of those kinds of effects in simulation. If you can imagine at this next clock age if the signal is still drifting, be stable. In the sample here, one of the outcomes we might get a one a different outcome. We might get a zero, and we don't know which of these it will be. Same for a hold violation.

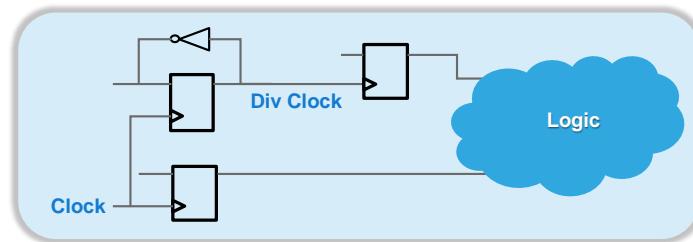
On the next clock, if we violate the whole violation at this age, on the next stage, we don't know what we're sampling necessarily. We could either sample a One or zero. And in real hardware, we don't know what we're going to sample.

What Is a Clock Domain?

A clock domain is defined as that part of the design driven by either a single clock or clocks that have constant phase relationships.

Synchronous and Asynchronous clocks

- **Synchronous:** Constant phase relationships
- **Asynchronous:** Variable phase and time relationships



238 © Cadence Design Systems, Inc. All rights reserved.



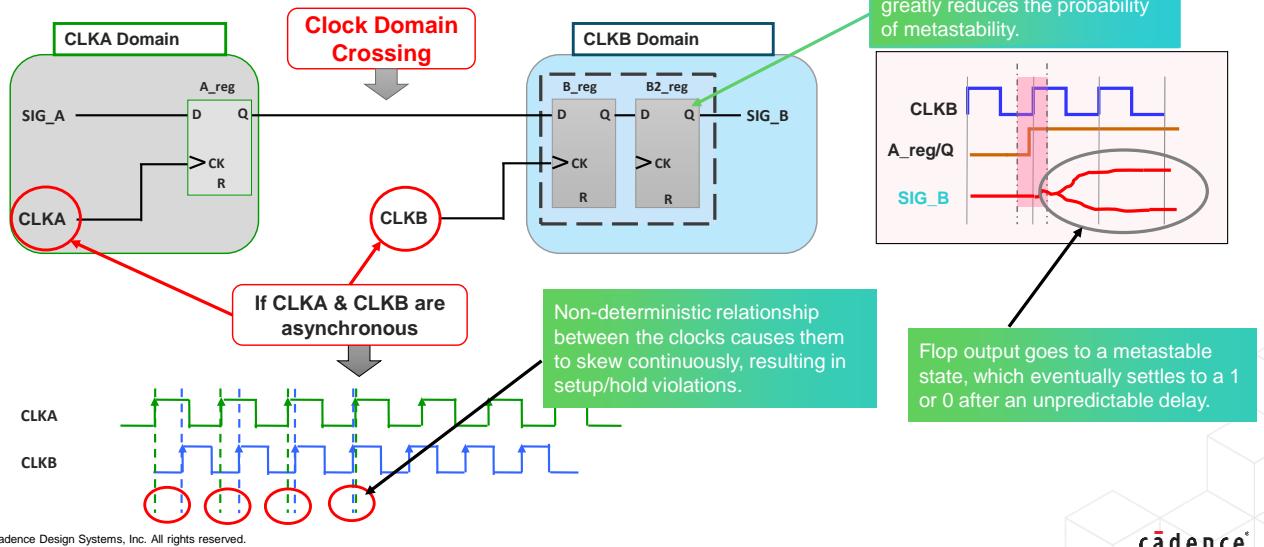
The subject we're talking about here is clock domain crossing.

What is a domain? It's part of the design driven by a single clock or clocks with a constant phase relationship.

You can get clocks that are synchronous to each other, i.e., they have a constant phase relationship, or asynchronous, where they have a variable phase relationship.

Overview: Clock Domain Crossing

Clock Domain Crossing (CDC) occurs when a signal crosses from one asynchronous clock domain to another.



The clock domain crossing is when a signal crosses from one asynchronous clock domain to another. We've got two clock domains here. We've got a clock A and a clock B.

We're driving the output of this flip-flop, every Q edge, and sampling it. This is the D ports of the flip flop. We've got a different clock here. These clocks were drifting around.

So here, these clocks are moving asynchronously to each other. Now at some point, we start narrowing that gap, and now we're changing the signals at the incorrect time. As far as the hold and set up times go and we're violating that requirement. Therefore, we will get an X value coming out of this signal B here. It's a signal B when the clocks get too close to each other. Therefore, we're violating setup and hold times. We're going to get this output going. That's stable.

So, what can do to mitigate this problem or greatly reduce the probability of stability is to put a synchronizing flip-flop here.

The likelihood of getting a signal, B, metal stability now is drastically reduced by orders of magnitude by having the synchronization circuit.

Mean Time Between Failure

- Metastability cannot be eliminated. Its effect can be minimized and reduce the probability of failure.
- The calculation of the probability of the time between synchronization failures (MTBF) is a function of multiple variables, including the clock frequencies used to generate the input signal and to clock the synchronizing flip-flop.

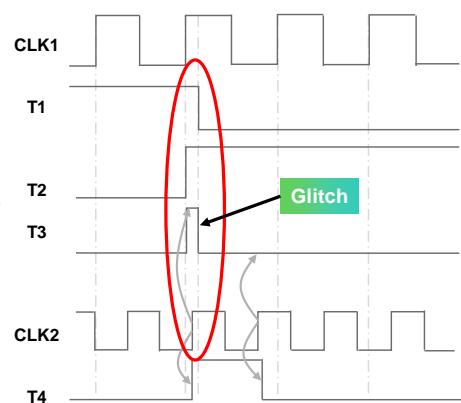
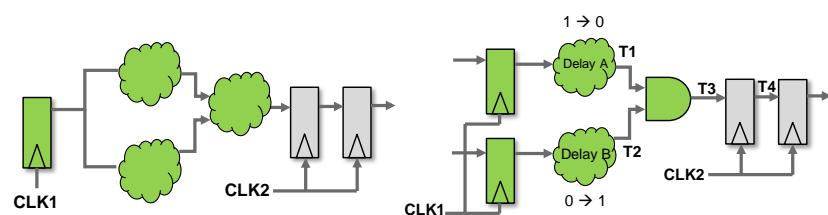
$$\text{MTBF} = \frac{1}{f_{\text{clk}} * f_{\text{data}} * X}$$

Synchronizing
clock frequency
Data changing
frequency
Other
factors

Metastability is an effect that cannot be eliminated. We can minimize it by having different kinds of hardware structures to synchronize.

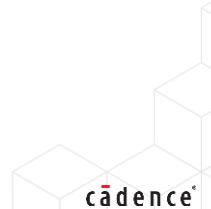
The probability is calculated via this equation here. So, it depends on the clock frequency, how often the data inputs, the flip-flop changes, and other factors as well to deal with the specific technology we're using.

Glitch on CDC Path



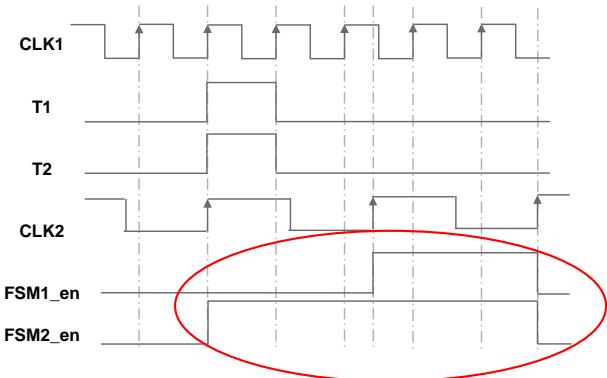
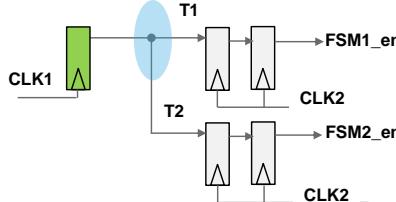
- Any logic in the crossover path can cause glitches and create functional errors downstream.
- Different delays in paths T1 and T2 causes a glitch.
- The glitch causes the flop output to go high when it was supposed to remain low.

241 © Cadence Design Systems, Inc. All rights reserved.



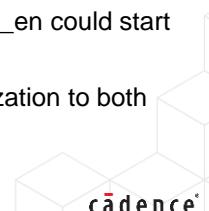
This page does not contain notes.

Divergence of CDC Signal



- A divergent logic style to multiple synchronization paths runs the risk of causing functional errors.
- Due to the propagation delay and different metastable settling times, the FSM1_en and FSM2_en could start at different times.
- This type of structure should be avoided by fanning out a single FSM enabled after synchronization to both FSMs.

242 © Cadence Design Systems, Inc. All rights reserved.

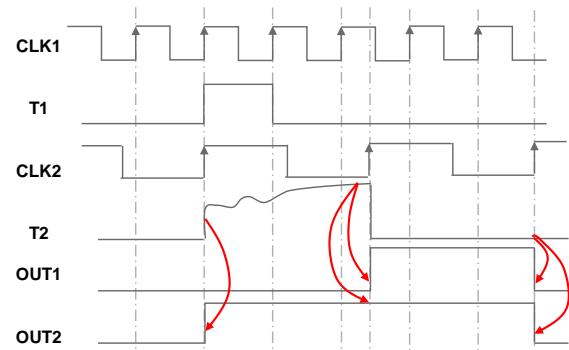
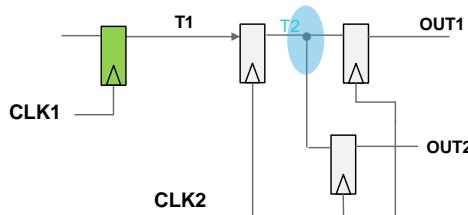


What we have here is an example of what happens if we try and diverge your signal. Here we've got an output of this flip flop from domain one, and we're splitting it off into two different synchronizes, both by clock two.

The signals from the first one enable an FSM to enable where our intent should be that these signals are both the same thing. However, due to different propagation delays and different settling times for these flip-flops, we might get a different value for the first one enabled and FSM to enable, which is what we intended, so we could avoid that kind of structure by fanning out a single FSM enables after synchronization, not before.

This is how we avoid that problem.

Divergence of Metastable Signals



OUT1 and OUT2 come out at two different clock edges as the settling and latching values of these metastable signals to the two flops are at different times.

We've got a signal from the main one where we're going through one flip-flop here, and output two is going to be stable, which means that although we're passing these through another synchronizing flip-flop here, the value that we will get from out one and out two may be different. Due to the different amounts of time it takes to settle in each different flip-flop.

We're storing the wrong value here at one or two or not storing the same value, which is our intent. So clearly, we shouldn't have taken that signal from here. We should have done it. But that signal should be taken from there instead.

Synchronization Schemes

- Metastable signals are unstable signals that need proper synchronization.
- We need to code, in RTL, an appropriate synchronizer that minimizes the possibility of an important signal becoming metastable.
- There are many ways of describing a synchronizer.
- We need to verify that our design does not have CDC problems.
- Structural analysis is basic CDC checking.
 - Advanced CDC analysis needs functional checks and metastability analysis.
 - Cadence JasperGold has an app dedicated to this problem.



In order to minimize the risk of meta stability and effect on the functionality of the design, we need proper synchronization circuits.

We need to code in RTL and appropriate synchronize that minimizes that possibility. There are many ways of describing these. We also need to verify that the design we created to mitigate this problem doesn't introduce any CDC problems or that there are domain crossings we forgot about. We didn't write synchronization.

What we have is JasperGold®. It is a formal verification tool with an app dedicated to these domain crossings to see if we get clogged domain crossing under any circumstance. Do we have an appropriate synchronization that will mitigate against all the problems we've seen about divergence and so on?

Commonly Used Synchronization Schemes

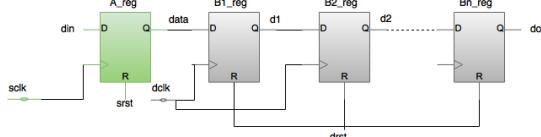


Fig: NDFF Synchronizer

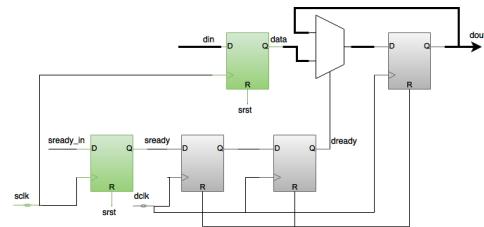


Fig: MUX Synchronizer

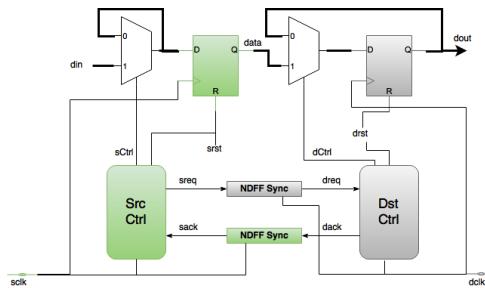


Fig: Handshake Synchronizer

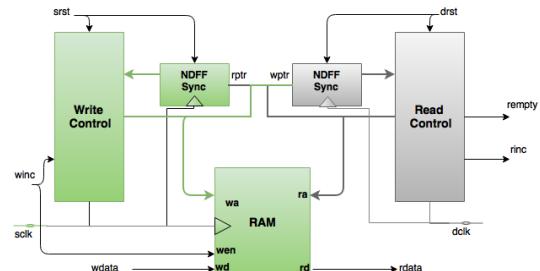


Fig: FIFO Synchronizer

245 © Cadence Design Systems, Inc. All rights reserved.

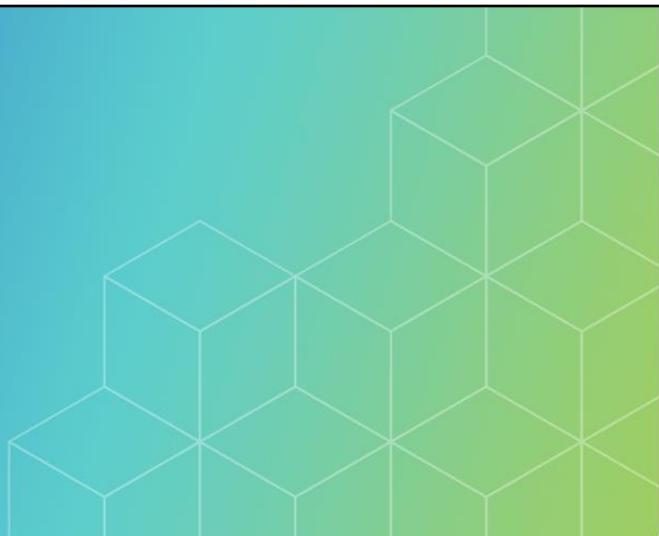
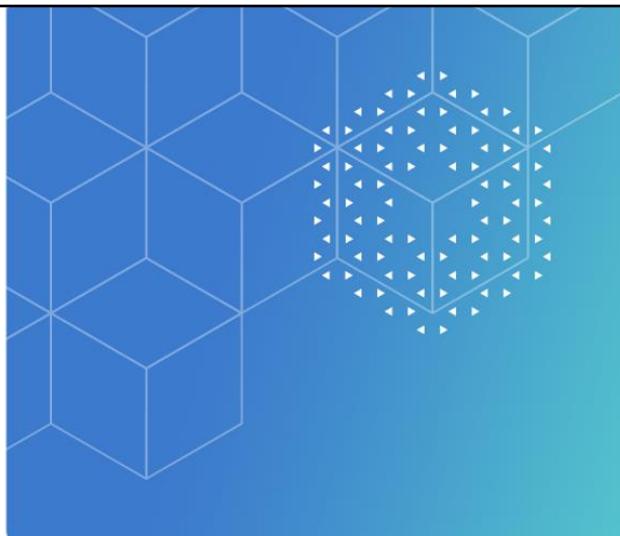
cadence®

These are the different kinds of ones you get; you can see they range from quite simple ones.

Just from this clock domain SW clock, we have two or more flip-flops. That's why it's called an NDFF synchronizer until there are more flip-flops. To do each subsequent flip-flop, we put in here a synchronization flip-flop, reducing the probability of seeing stability.

Probability is drastically reduced on the second one, and it's reduced even further by a similar factor on the third one.

We can also have other constructs that use boxes, for example, on different handshakes. We have light controllers' handshaking between these two things and can have a user memory to synchronize. We have two port memories, where we can read and write with different clocks.



Submodule 3-2-3

Reset Domain Crossing (RDC)

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Explore the concept of reset synchronization.

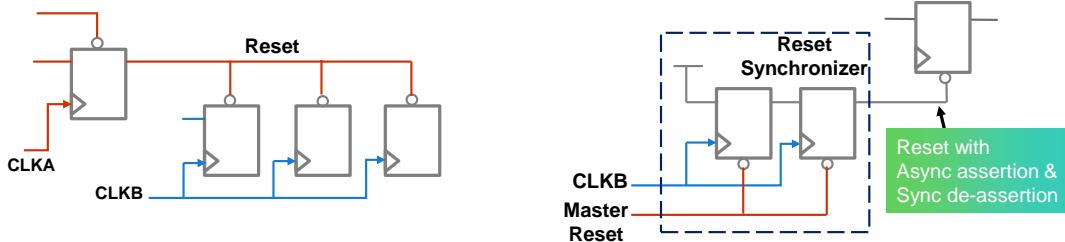


This page does not contain notes.

Reset Synchronization

If asynchronous reset is de-asserted near the active edge of the clock and violates the reset recovery time, it could cause the flip-flop to go metastable.

- A CDC verification tool, like the JasperGold CDC App, should report such scenarios.
- Solution: Asynchronous reset assertion and synchronous reset de-assertion.



248 © Cadence Design Systems, Inc. All rights reserved.

cadence®

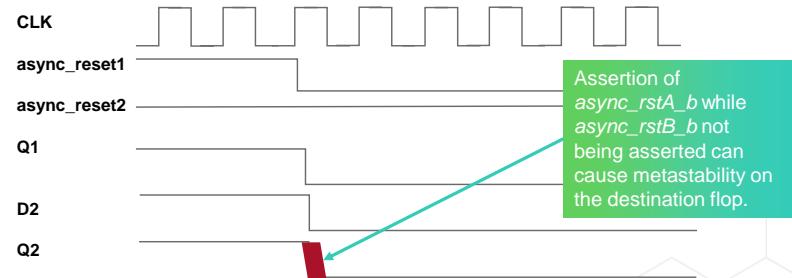
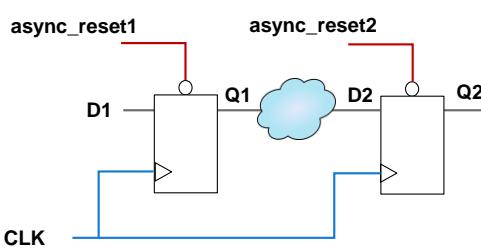
Now we have a similar problem with resets as well. This is called reset domain crossing because most modern chips do not have a single reset. They have more than one. So, if the reset is reasserted near the active edge of a clock, then we can violate the reset recovery time, which could cause the flip-flop to go to stable.

Then you may also have other physical factors like how long it takes the reset to propagate across the whole chip. We might get metastability out of these flip-flops if we have that kind of scenario. The solution is to use a CDC verification tool just for gold CDC to report such scenarios.

What we can do here is if we've got an asynchronous reset, we assert it asynchronously, and then we use a flip-flop to synchronize the reset here. The reset is asserted synchronously because we've got these synchronizing flip-flops here. There's the massive reset going in. The imports are tied high here. And we feed that into the reset of all these other flip-flops.

Reset Domain Crossing

- Just like with clocks, there are reset domains in the design.
- All flops connected to the same reset belong to one reset domain.
- Reset domain crossing can happen even within the same clock domain.
- If the async reset of the source register is different from the async reset of the destination register, even though the data path is in the same clock domain, it can lead to metastability at the destination register.
- A CDC verification tool, like the JasperGold CDC app, should report such scenarios.

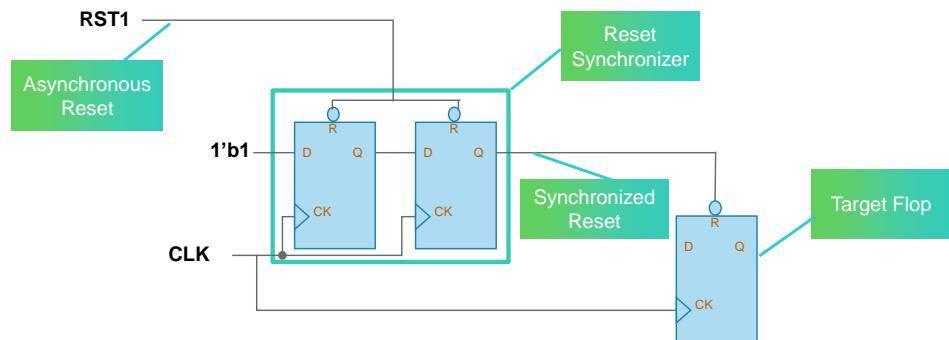


249 © Cadence Design Systems, Inc. All rights reserved.



Just like with clocks, we have different reset domains, and all flip-flops are connected to the same reset signal. Have membership of one reset domain, with reset the main crossing. This can happen even if you have the same clock for the flip-flop. We have two different reset signals here, async reset one and asynchronous set two. And we might get undesirable effects from this, like metastability, when the async reset two was released.

Reset Synchronizer



- When asynchronous reset signal crosses from one clock domain to another:
 - It should pass through a reset synchronizer.
- The output of the reset synchronizer has an asynchronous assertion, and synchronous de-assertion property:
 - Provides additional clock cycles for the target flops to recover after reset removal.

250 © Cadence Design Systems, Inc. All rights reserved.



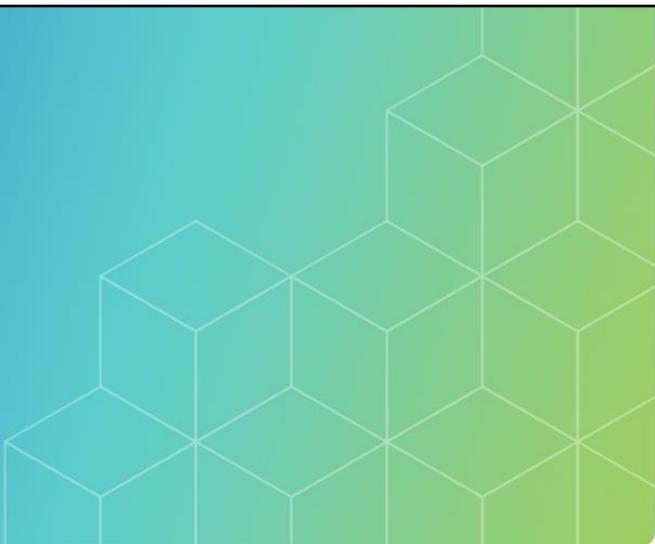
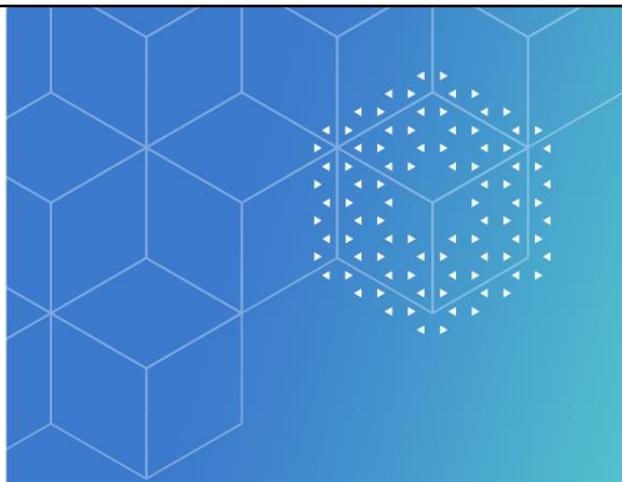
Just like clock domain crossing, we need an appropriate reset synchronizer here.

We've got a code in RTL ourselves, and we will check that we've done it correctly with some kind of CDC app that will also check the reset domain crossings.

Here's our asynchronous reset.

These flip-flops are synchronized so that we apply the resets asynchronously, and we release the reset synchronously in order to not violate setup and hold times.

Recovery times cause metastability on the output of the flip-flop.



Submodule 3-2-4

Clock Gating

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Explore the concept of clock gating.



This page does not contain notes.

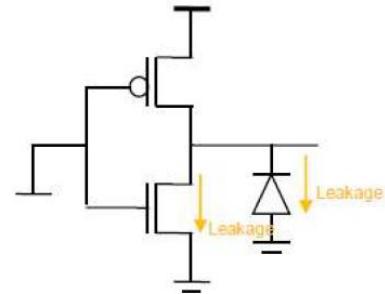
Power Consumption in a CMOS Digital IC

There are two sources of power consumption in CMOS ICs:

- Static power
- Dynamic power

Static Power – Leakage

- This is constant.
- It depends on the IC process technology we use, a function of:
 - The Supply Voltage (V_{dd})
 - The Switching Threshold (V_{th})
 - The transistor dimensions
- We can't do anything about this in our RTL code.



253 © Cadence Design Systems, Inc. All rights reserved.

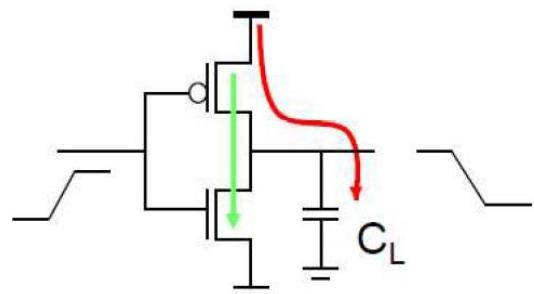
Clock gating is a way of reducing power consumption.

In a CMOS chip, which all modern chips are, there's two sources of power consumption. One is static power; one is dynamic static power. You can't really do anything about this in your RTL code. This is based upon the technology, and it's constant, hence its name. It's a function of the supply voltage, the switching threshold, the kind of technology you've got, and the transistor dimensions. So, there's nothing you can do about that. That's all part of the process.

Dynamic Power Consumption in a CMOS Digital IC

Dynamic Power – Switching

- A sum of:
 - Switching Power (charging/discharging capacitance of nets and transistors) – **the red line**.
- Short circuit power – **the green line**.
 - During switching, both P and N (CMOS) transistors are on for a very short time.
 - That is, the power supply is short-circuited.



We can do something about Dynamic Power in our RTL code.

Our RTL determines when switching occurs.



What happens when you do this to a car battery?

254 © Cadence Design Systems, Inc. All rights reserved.



What you can do something about, however, is dynamic power because this is due to switching.

The switching power is the sum of the charging and discharging of the capacitance of the load, the load being the inputs to other transistors on the wire, the interconnects. This green line shows you the short circuit power. CMOS is a complementary transistor. When one is on, the other is off, and vice versa. Now, during the switchover, when you're switching something from on to off, both transistors are on for a very brief period.

What happens if you do that to a car battery? Well, you draw like infinite current, in effect, until everything just burns down and melts.

We can do something about that in our code because we can determine how often the switching occurs.

What Can We Do in RTL to Reduce Power?

- Use RTL coding techniques, for example, manual resource sharing and Gray coding for FSM state encoding.
- However, these techniques might be time-consuming to implement and not have a significant enough effect to make it worth the effort.
- Reducing the switching activity has a much greater effect.
- We need to ensure that switching only occurs when absolutely necessary.
- Remember, we said in a previous lecture that a complex processor at a 7nm node can only have 50% of the transistors active at any given time to keep below the power budget.
- To reduce the switching activity, we use a technique called **Clock Gating**.



We can use techniques we've discussed already, like manual resource sharing, Gray coding for state machines, etc. But these might be time-consuming to implement and verify and not really have enough effect to make it worth the bother. What has a much greater effect is, reducing the switching activity so that we only have parts of the circuit switching when they need to.

Remember, we said previously that for a new process node like seven nanometers, you might only be allowed to switch 50 percent of the transistors at any given time if you keep below the power budget and stop the chip kind of overheating.

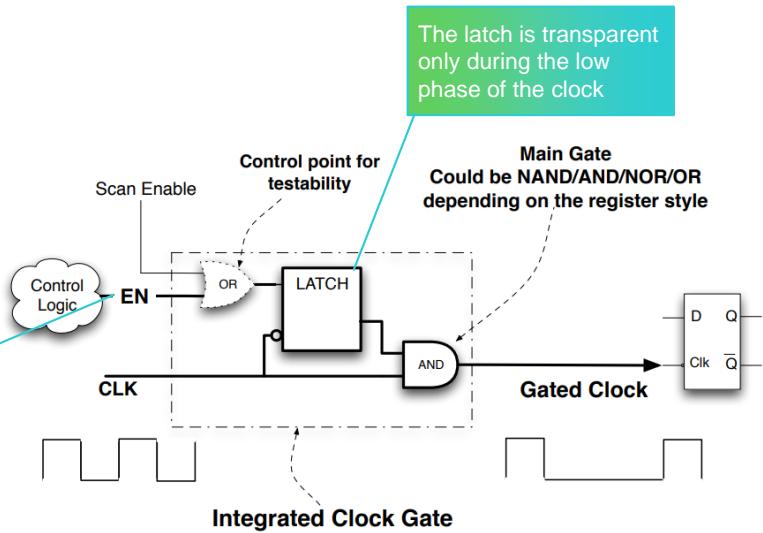
We can reduce switching activities by a technique called clock gating. Clock gating is a good technique but it creates another verification headache for us.

What Is Clock Gating?

A technique used in synchronous circuits to reduce dynamic power by removing the clock when it is not essential to design intent functionality.

- Effectively, we prune the clock tree to save power.
- However, remember the PPA tradeoff?
- This is at the expense of more gates.

Design Block
Enable Signal
generated from our
Control Logic



256 © Cadence Design Systems, Inc. All rights reserved.

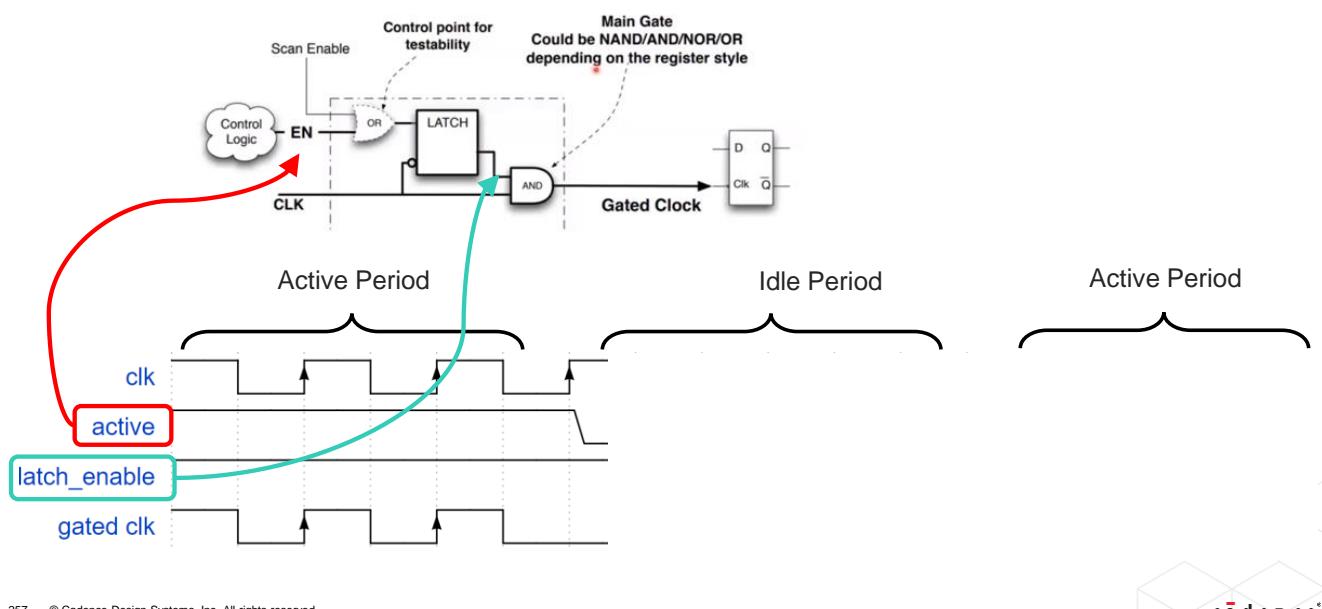


Clock gating is a technique to reduce dynamic power by effectively pruning the clock tree.

Remove the clock when it's not essential to the design functionality. This is at the expense of more gates; however, we make a higher performance area in the tradeoff. We have more gates to consume less power. Here's an example of clock gating. The clock to our chip isn't coming from one of the input pins. It's coming from all this different logic here.

We can see some control logic here, and we've got an and gate here with the clock, which is coming from the input pin and ending with all these different signals. The control logic for our clock gating and other things like scan enable is used for manufacture testing. It's a control point for testability. We've got all this extra circuitry to reduce the number of times we switch this flip-flop.

Example: Waveforms for a Gated Clock



257 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Why Is Clock Gating Such a Hard Problem?

The need for power efficiency requires clock gating to be manually defined at the RTL level.

Because doing it in RTL is more efficient than:

- Software-level clock gating.
- Automated clock gating applied by synthesis tools.

Because we code the clock gating manually in RTL:

- It's highly error prone.
- It makes verification much more difficult.

To obtain a reasonable amount of coverage of clock gating activity using simulation-based flows may be unfeasible in acceptable timescales.

The exhaustive nature of formal verification fits well into this domain.

258 © Cadence Design Systems, Inc. All rights reserved.



Clock gating is a complex problem. You can't do this automatically.

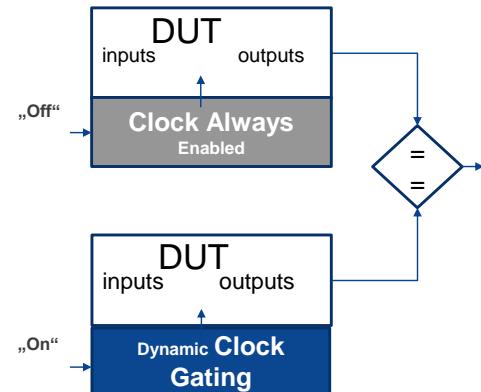
There are things in the software that can apply such techniques, so you can get synthesis tools that use Clock gating automatically or do this at some kind of software level. Still, you're never going to be the efficiency of a human. This is why people do it because you get the best power efficiency. This institution has its own problems because it's a person doing it. It's highly error-prone, and it makes verification much harder now.

We now must consider all the different stages or states the clock gating could be at.

Doing this in a simulation is unfeasible in any kind of acceptable timescale, being as we've got this deadline the tapeout to meet. This problem is well suited to a formal because its formal and exhaustive nature allows it to determine that the clock gating works correctly under all conditions.

What Happens When We Change a Design to Use Gated Clocks?

- We may not have been aware that we had a low power requirement when we first created our design.
- Clock gating is intended only to reduce power consumption, not change the functionality of the design.
- This greatly complicates our verification requirements.
- We can solve this with a formal technique known as “Equivalence Checking.”
- We ask a formal tool to prove to us that the original design is the same as the design modified to have clock gating.
- Cadence JasperGold has an Equivalence Checking App called SEC (Sequential Equivalence Checking) to verify that clock gating does not affect the original functionality.



259 © Cadence Design Systems, Inc. All rights reserved.



What happens if we change your design to use gated clocks? We had a previous design that didn't consider clock gating. Now we have decided we need clock gating because we're using this newer technology where we can't switch everything at the same time. We introduced clock gating to reduce power consumption, not to change the functionality of the design. So how can we deal with this in formal?

We can do this with something known as equivalence checking so we can take the design that didn't have a clock gating gated clock. On a design that does have a click-gated clock and check that the outputs are always identical, i.e., we didn't change the functionality by introducing clock gating. This app is known as sequential equivalence checking in just four gold. That's one of its primary uses for inserting clock gating didn't change the functionality of the design.

Can We Define All Low-Power Requirements in RTL?

No, we can't.

Other low-power techniques related to physical effects have to be defined using a different language.

Because RTL has no way of expressing those techniques.

The Power Intent Language is called UPF (Unified Power Format).

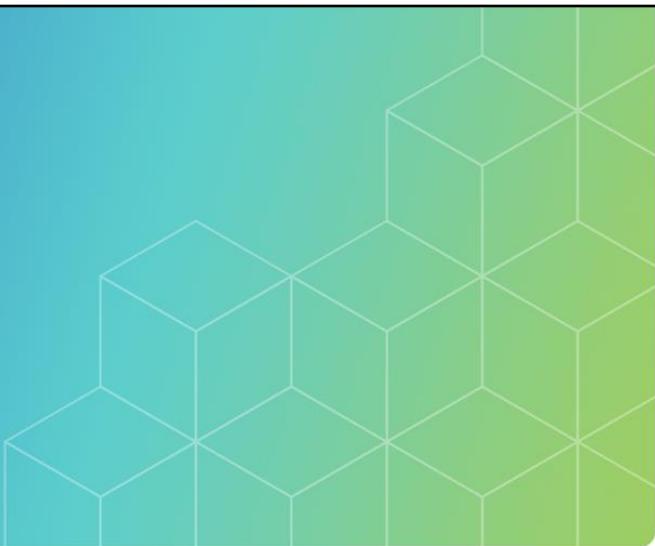
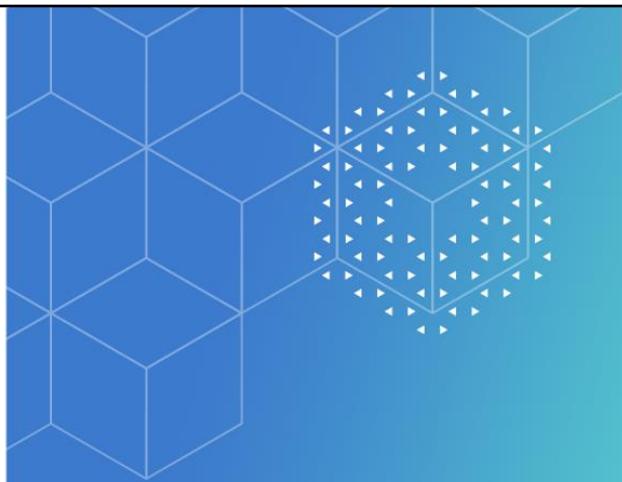
UPF is an IEEE standard – IEEE 1801.

This is the subject of the next lecture.



Can we define all our power requirements in RTL? Can we address all those low-power requirements?
No, we can't.

We need other low-power techniques related to the physical effects, and we define these using a different language. This is the thing like isolation cells, level shifting cells in terms of shifting the voltage levels. We can't describe that in RTL because it doesn't have the semantics to do it. So, we need another language known as Unified Power Format (UPF), which specifies the power intent. This is not really a standard. Watch IEEE 1801.



Submodule 3-3

Low-Power Concepts

cadence®

This page does not contain notes.

Submodule Objectives

In this submodule, you will:

- List power reduction techniques.
- Analyze low-power concepts.

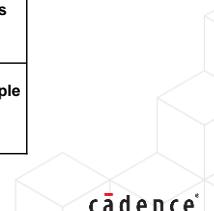


This page does not contain notes.

Power Reduction Techniques Summary

Power Reduction Technique	Leakage Power	Dynamic Power	Timing	Area Penalty	Methodology Impact	Methodology Change
Low-Power Optimization	10%	10%	0%	10%	None	None
Multi-Vt	6X	0%	0%	0%	Low	Multi-Vt library needed
Clock Gating	0%	20%	0%	<2%	Low	Clock-gating cells needed and extra overhead in STA
Multi-Supply Voltage (MSV)	2X	40-50%	0%	<10%	Medium	Micro-architecture and methodology needs to be domain aware; need voltage regulators and level shifters; verification and analysis challenge
Power Shut-Off (PSO)	10-50X	0%	4-8%	5-15%	Medium-High	Insertion of switch cells; retention flops; wake-up and shut-down time analysis; power shut off and restore verification
Dynamic Voltage Frequency Scaling (DVFS)	2-3X	40-70%	0%	<10%	High	Deterministic scheduling; multi-mode optimization and analysis flow needed; clock synchronization
Substrate Biasing	10X	0%	10%	<10%	High	Maintain well separation; multiple power rail distribution; static timing analysis

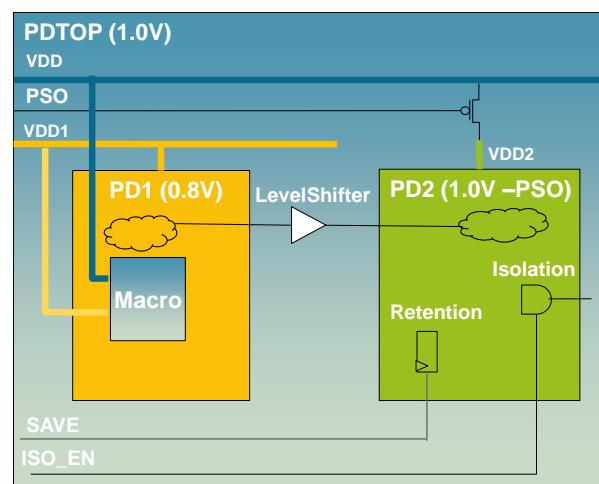
263 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Low-Power Concepts

Power Domain	<ul style="list-style-type: none"> A user-defined grouping of instances that share the same supply nets.
Level Shifters	<ul style="list-style-type: none"> Handle interfaces that have different voltage levels.
Power Switches	<ul style="list-style-type: none"> Switch definition that turns on/off power to a domain.
Isolation	<ul style="list-style-type: none"> Isolates a power shutoff domain from functional logic. Ensures electrically and functionally correct interfaces during shutoff.
Retention	<ul style="list-style-type: none"> Ability to specify registers as being maintained through power shutoff. Many styles of control are available.
Macro Model	<ul style="list-style-type: none"> Hard IPs with multiple supplies that have been blackboxed e.g., memory models.



**Three Power Domains: PDTOP (VDD + VSS)
PD1 (VDD1 + VSS), PD2 (VDD2 + VSS)**

The main aspects of power intent for low-power design are Power Domain, Level Shifters, Power Switches, Isolation, Retention, and Macro Model.

What Are the Low-Power Library Cells Required for Implementing the PSO Technique?

PSO design requires the following low-power cells:

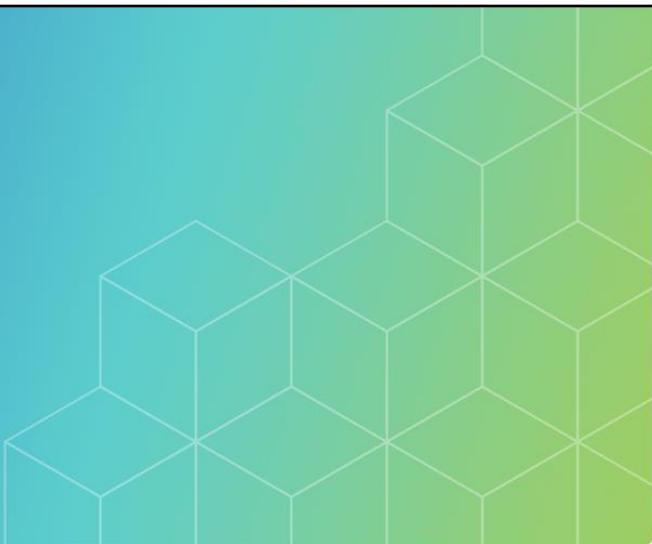
- Power switches to turn on/off the power supply to a domain.
- Isolation cell is used to isolate an active (ON) domain from a signal from an inactive (OFF) domain.
- Retention cell is a special D flip-flop that has control to save a state when power is turned OFF so the unit can recover when power is restored.
- AON buffers for signals that feed through the shutoff domain.



Answer

PSO design requires the following low-power cells:

- Power switches to turn on/off power supply to a domain.
- Isolation cell is used to isolate an active (ON) domain from a signal from an inactive (OFF) domain.
- Retention cell is a special D flip-flop that has control to save a state when power is turned OFF so the unit can recover when power is restored.
- AON buffers for signals that feedthrough shutoff domain.



Submodule 3-3-1

Introduction to Low-Power Simulation

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Identify implicit low-power simulation behaviors.

Topics include:

- Power-shutoff simulation behavior
 - Port isolation, state retention, state loss
- The generic power control sequence
 - For balloon latch technologies: edge-based or level-based save and restore
 - For master/slave-alive latch technologies: a retention condition

Before studying this module, you need to be familiar with Xcelium™ digital simulation and debug.

267 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to identify implicit low-power simulation behaviors, that is, port isolation, state retention, and power shutoff.

This training module discusses:

- Power-shutoff simulation behavior, including port isolation, state retention, and state loss; and
- The generic power control sequence, which for balloon latch technologies, can use edge-based or level-based save and restore signals, and for master/slave-alive latch technologies, uses a retention condition.

Glossary of Low-Power Simulation Fundamental Terms

Low-Power Simulation (LPS)	Power-aware digital simulation.
Power Design	Unique power structure.
Power Domain	Collection of design elements sharing a power distribution network.
Power Shutoff (PSO)	Feature to switch domain power on and off.
Isolation	Defines logic signal level while otherwise not driven due to power shutoff.
Retention	Preserves state values during shutoff.
State Retention Power Gating (SRPG)	Retaining states through power shutoff.
Balloon Latch	Dedicated low-power state-retention latch.



This training module introduces terms that may be new to you.

- **Low-Power Simulation (LPS)** is power-aware digital simulation.
- **A Power Design** is a unique power structure that can be associated with either a top design or with one or more logic modules.
- **A Power Domain** is a collection of instances, pins and ports that can share the same power distribution network.
- **Power Shutoff (PSO)** is a power reduction technique that switches off a power domain while it is not active.
- **Isolation** is a technique that defines logic signal values while the signals are otherwise not driven due to power shutoff.
- **Retention** is a technique that preserves sequential element states that would otherwise be lost due to power shutoff.
- **State Retention Power Gating** is a technique to retain design states through power shutoff.
- **A Balloon Latch** is a dedicated always-on low-power latch added to a sequential element to preserve the element value while shutoff.

What Is Low-Power Simulation?

Low-power simulation is the simulator mimicking the target hardware's reduced-power behavior.

You simulate reduced-power behavior to confirm design reaction to power-change events.

For example, for power shutoff, the simulator mimics the behaviors:

- Port isolation – forces a known logic level onto a powered-on input not driven due to power shutoff.
- State retention – saves state values through power shutoff and then restores them.
- State loss – forces the unknown logic level onto elements driven by the powered-down domain.

Power-Down: =>



Power-Up: =>



269 © Cadence Design Systems, Inc. All rights reserved.



Digital simulation performs three behaviors related to power domain shutoff.

For the power-down sequence, it:

1. Forces a specified value onto the staying-on receivers of nets driven by the domain about to be shut off, to prevent propagating unknown and, thus, potentially destructive values from a power domain that is shut off to a power domain that remains on.
2. Saves the value of selected sequential elements in the domain about to be shut off; and
3. Mimics state loss in the shut-off domain by corrupting all domain inputs that have loads in the domain, and all domain state elements and driven nets that are not part of an always-on cell.

For the power-up sequence, it:

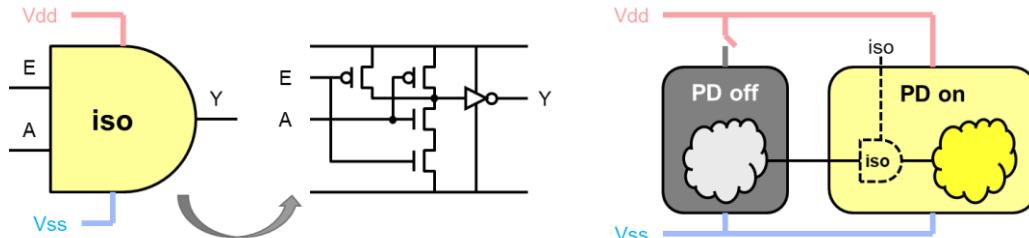
1. Releases the corrupting forces;
2. Restores the saved values of the selected sequential elements; and
3. Releases the isolation forces.

What Is Port Isolation in a Low-Power Simulation Context?

Port isolation forces a logic level onto an input pin not driven due to power shutoff.

You isolate a powered-on cell input driven by powered-off cell output to prevent rogue input values from affecting the powered-on cell functionality and possibly even damaging it.

- For gates, simulation executes behaviors specified by the technology library isolation cell simulation model.
- For RTL, simulation forces specified values onto undriven inputs of cells in the powered-on power domains.



270 © Cadence Design Systems, Inc. All rights reserved.

cadence®

A power domain is usually connected to other power domains. When shut off, it could propagate unknown and thus potentially destructive levels to power domains that remain on. Low-power design adds isolation logic between power domains that can shut off and power domains that can remain on, to prevent propagating unknown states from a power domain that is shut off to a power domain that remains on.

The post-synthesis gate-level netlist includes actual isolation cells to provide the isolation behavior.

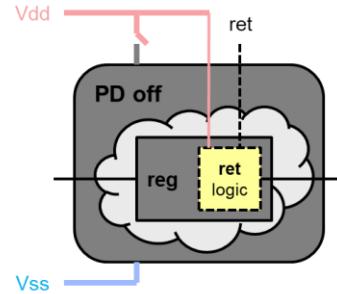
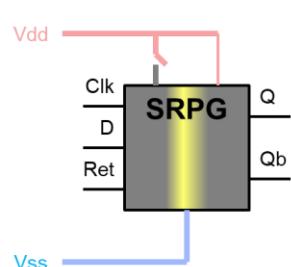
At the register-transfer level, the simulator implicitly models this pin isolation by simulating the effect of inserting an appropriate isolation cell. It inserts the virtual isolation cell outside the powered-off domain as a driver of the net connected to the powered-on domain. The isolation logic drives a specified logic value to the pin in the powered-on domain.

What Is State Retention in a Low-Power Simulation Context?

State retention saves state values through power shutoff and then restores them.

You retain critical (not all) states through power shutoff to resume operation after power up.

- For gates, simulation executes behaviors specified by the technology library retention cell simulation model.
- For RTL, simulation retains and restores states of specified state elements.



271 © Cadence Design Systems, Inc. All rights reserved.

When a power domain is powered down, the states of key sequential elements in the power domain, such as some latches, flip-flops, and registers, must be saved and retained for the entire shutoff period. When the power domain is powered back up, the saved states must be restored to the sequential elements. To ensure that a powered-down domain resumes normal operation after power up, state retention cells replace these key sequential elements. Power-efficient designs apply state retention to only the key sequential elements that must be retained.

The post-synthesis gate-level netlist includes actual state retention cells to provide the state retention behavior.

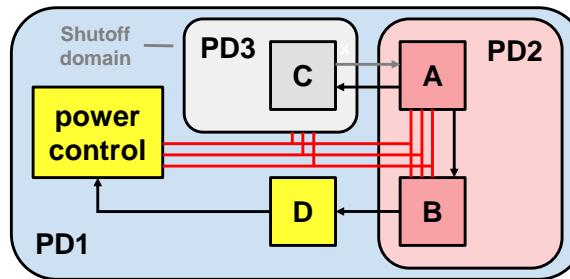
At the register-transfer level, the simulator implicitly models state retention by simulating the effect of inserting appropriate state retention cells. It saves the state of the cell upon the save event and restores the state of the cell upon the restore event. If the save or restore signals meanwhile become unknown, then the simulator makes the saved value also unknown.

What Is State Loss in a Low-Power Simulation Context?

State loss corrupts state values while power is shut off.

Outputs of unpowered cells float to undetermined values. The simulation must mimic this behavior.

- For gates, simulation executes behaviors specified by the technology library retention cell simulation model.
- For RTL, simulation corrupts by default all domain inputs that have loads in the domain and corrupts all nets, signals, and variables driven by the powered-down domain except those specified not to be corrupted.



272 © Cadence Design Systems, Inc. All rights reserved.



At the physical level, when a power domain shutoff condition becomes true, appropriate switches open, resulting in a loss of voltage to the domain. With no proper biasing, the local voltages float, and the output of every cell in the shutoff region is undetermined.

Simulation models this behavior by corrupting all[†] domain inputs that have loads in the domain, and all domain state elements and driven nets that are not part of an always-on cell. It forces the output of the corrupted elements to a corruption value. All signals within the shut off region transition to the corruption value, and all sequential elements (registers) and variables within the shutoff region lose their state and transition to the corruption value.

The simulator also powers off the associated power domains if the shutoff condition becomes unknown.

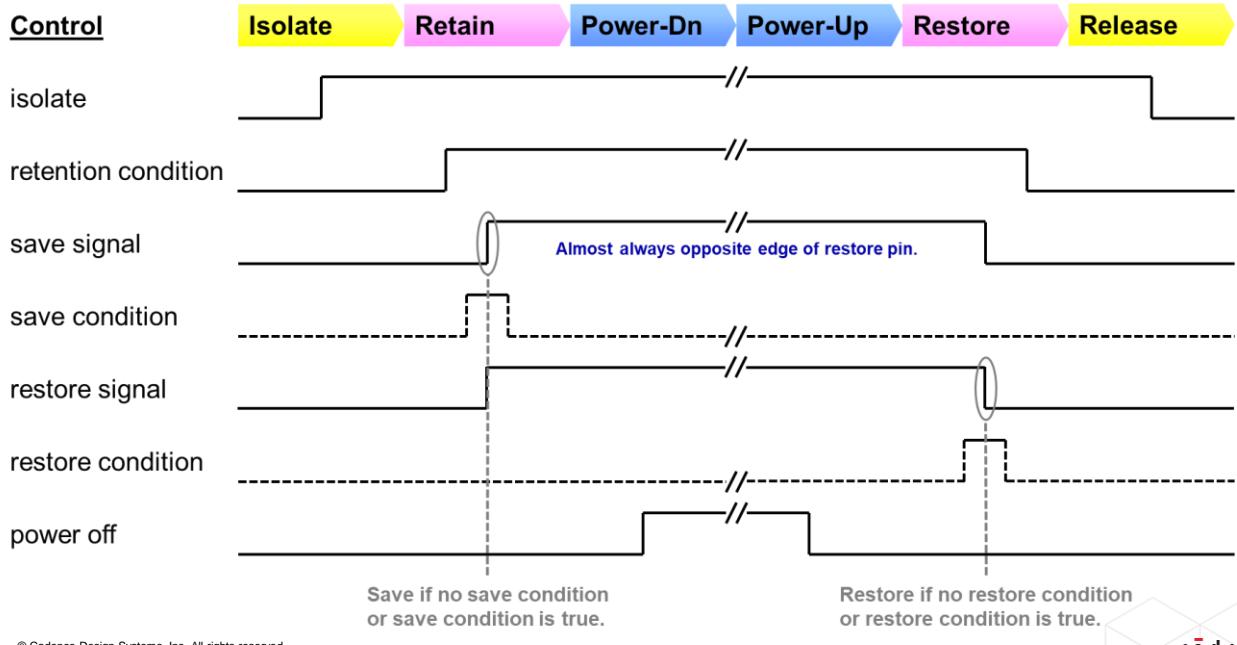
Upon shutting off a SystemVerilog power domain, the simulator assigns the unknown logic value (X) to all variables within the shutoff domain and drives all nets within the shutoff domain that are not part of an always-on cell to the unknown logic value.

Upon shutting off a VHDL power domain, the simulator assigns the unknown logic value (X) to all signals and variables of standard logic types within the shutoff domain. To signals and variables of enumerated types, it, by default, assigns the leftmost value.

When a power domain is again powered on, the simulator releases the forced corruption and resumes normal simulation activity. Power-up behavior can generate glitches in domain signal values similar to those that occur when real hardware powers up. The simulator by default does not restore any previously forced values, but you can optionally change this behavior.

[†] An optional driver-based algorithm more accurately chooses the loads in the domain to corrupt.

Generic Power Cycle Sequence for Balloon-Latch: Edge-Based



273 © Cadence Design Systems, Inc. All rights reserved.

cadence®

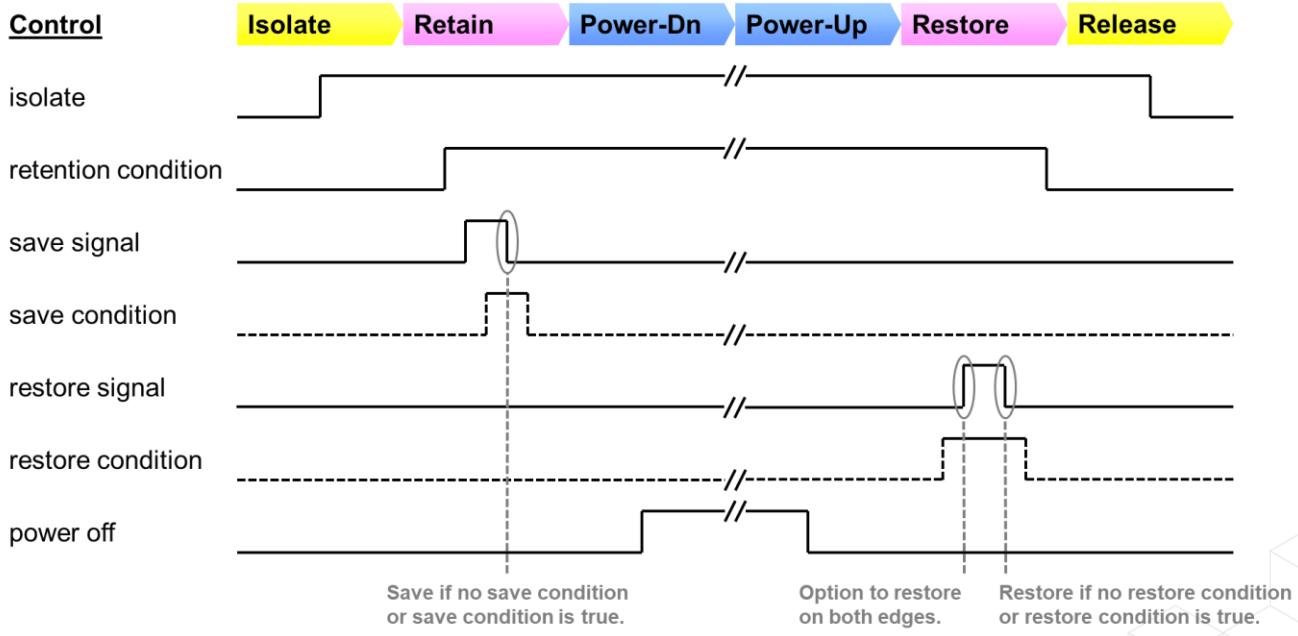
Implicit simulation behaviors support multiple retention technologies.

This is the generic balloon latch edge-based state retention technology.

- Isolation introduction is independent of the retention technology.
- State retention occurs on the save signal active edge.
 - The save condition is optional. You can use it to guard the save edge.
- The optional retention condition, if false, corrupts the saved value.
- Power-down and power-up are independent of the retention technology and do not affect balloon latch edge-based state retention.
- State restoration occurs on the restore signal active edge.
 - The restore condition is optional. You can use it to guard the restore edge.
- Isolation removal is independent of the retention technology.
- The save and restore signals are almost always opposite edges of one cell pin.

You describe all these behaviors by entering commands into your power intent document.

Generic Power Cycle Sequence for Balloon-Latch: Level-Based



274 © Cadence Design Systems, Inc. All rights reserved.



Implicit simulation behaviors support multiple retention technologies.

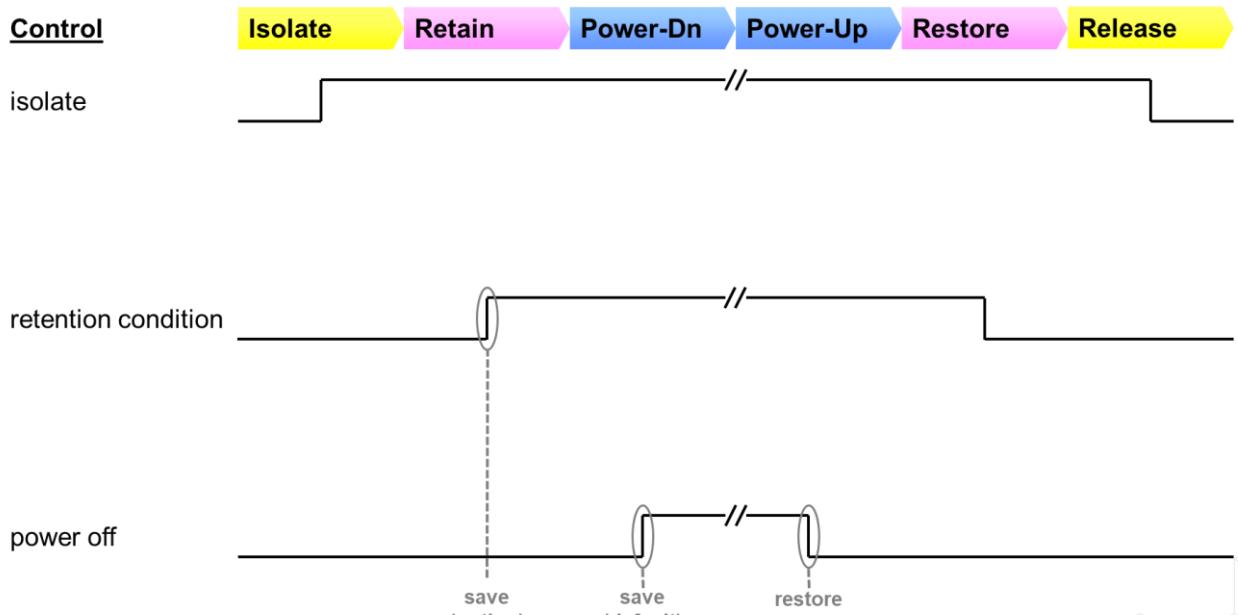
This is the generic balloon latch level-based state retention technology.

- Isolation introduction is independent of the retention technology.
- State retention occurs on the save signal trailing edge.
 - The save condition is optional. You can use it to guard the save edge.
- The optional retention condition, if false, corrupts the saved value.
- Power-down and power-up are independent of the retention technology and do not affect balloon latch level-based state retention.
- State restoration occurs on the restored signal trailing edge.
 - The restore condition is optional. You can use it to guard the restore edge.
 - You can optionally specify to perform state restoration on both restore signal edges.
- Isolation removal is independent of the retention technology.

You describe all these behaviors by entering commands into your power intent document.

For Xcelium simulation, the elaborator option `-lps_restore_level` specifies restoring the state on both edges of the level-sensitive restore signal.

Generic Power Cycle Sequence for Master-Slave Latch



16
© Cadence Design Systems, Inc. All rights reserved.



Implicit simulation behaviors support multiple retention technologies.

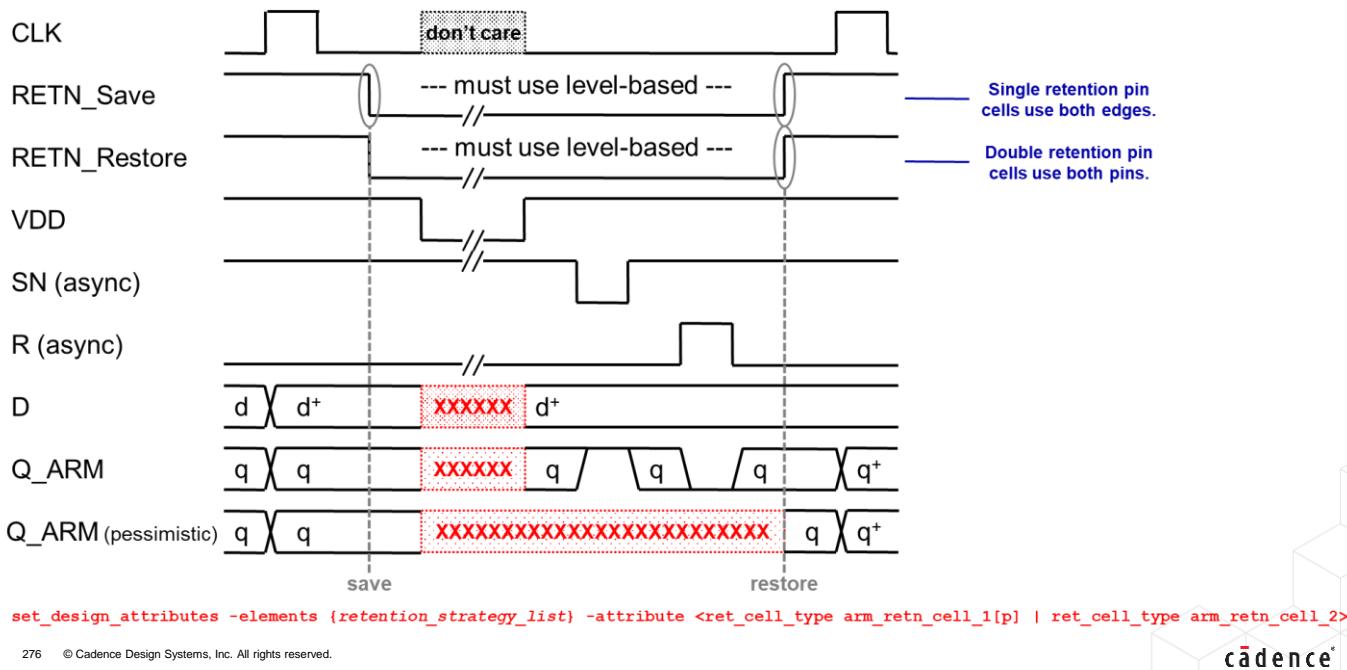
This is the generic master/slave-alive state retention technology.

- Isolation introduction is independent of the retention technology.
- State retention optionally occurs upon the retention condition becoming true.
- Power-down is independent of the retention technology. The slave latch, by default, stays alive, and so retains its state.
- State restoration occurs upon power-up.
- Isolation removal is independent of the retention technology.

You describe all these behaviors by entering commands into your power intent document.

For Xcelium™ simulation, the elaborator option `-lps_rtn_save_lock` specifies to instead save the state at the retention condition leading edge.

Non-Standard Level-Based Variation for ARM® State Retention



Some ARM state retention cells exhibit a slightly different behavior from the IEEE Std. 1801-2015 retention strategies cannot describe. They restore state continually when the clock is low, and power is restored, and up to the end of the restoration interval. Meanwhile, the set and reset pins can combinatorially affect the current output but not the saved state. Xcelium simulation provides an RTL solution for SystemVerilog and VHDL that supports these ARM state retention cells. For these cells, it supports only level-sensitive retention rules and issues an error upon encountering an edge-sensitive retention rule. To designate the retention strategies to which this behavior applies, set their retention-cell-type (ret_cell_type) attribute, for single-pin save and restore, to the value arm-retention-cell-1 (arm_retn_cell_1), and for double-pin save and restore, to the value arm-retention-cell-2 (arm_retn_cell_2). The single-pin save and restore, but not the double-pin save and restore, also has a pessimistic cell model that holds the cell output unknown until the end of the restoration interval.

```
set_design_attributes -elements {MY_RR_RULE} -attribute ret_cell_type arm_retn_cell_1
set_design_attributes -elements {MY_RR_RULE} -attribute ret_cell_type arm_retn_cell_1p
set_design_attributes -elements {MY_RR_RULE} -attribute ret_cell_type arm_retn_cell_2
```

Test Your Understanding

- What basic power behaviors can a simulator add to an RTL simulation?
 - To an RTL simulation, the simulator adds implicit port isolation and state retention, corruption, and restoration behaviors.
- What control signals does a power controller typically generate?
 - A power controller typically generates isolation, retention, and shutoff signals.
- Explain how you differently use edge-based and level-based balloon-latch state retention.
 - Edge-based balloon-latch retention cells almost always have only one save-restore pin and use opposite save-restore active edges. Level-based balloon-latch retention cells more frequently have separate save-restore pins and, by default, operate on their inactive edges.



This page does not contain notes.

Submodule Summary

In this module, you:

- Identified implicit low-power simulation behaviors.

This training module discussed:

- Power-shutoff simulation behavior.
 - Port isolation, state retention, state loss.
- The generic power control sequence.
 - For balloon-latch technologies: edge-based or level-based save/restore.
 - For master-slave latch technologies: a retention condition.



You should now be able to identify implicit low-power simulation behaviors.

This training module discussed:

- Power-shutoff simulation behavior, including port isolation, state retention, and state loss; and
- The generic power control sequence, which for balloon latch technologies, can use edge-based or level-based save and restore signals, and for master/slave-alive latch technologies, uses a retention condition.

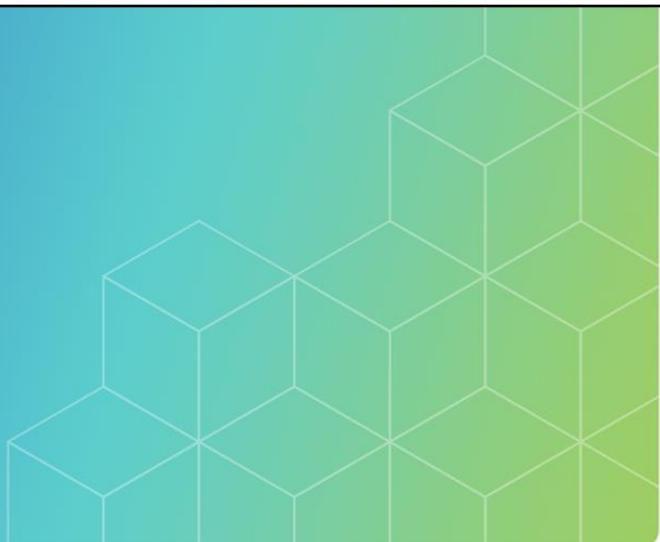
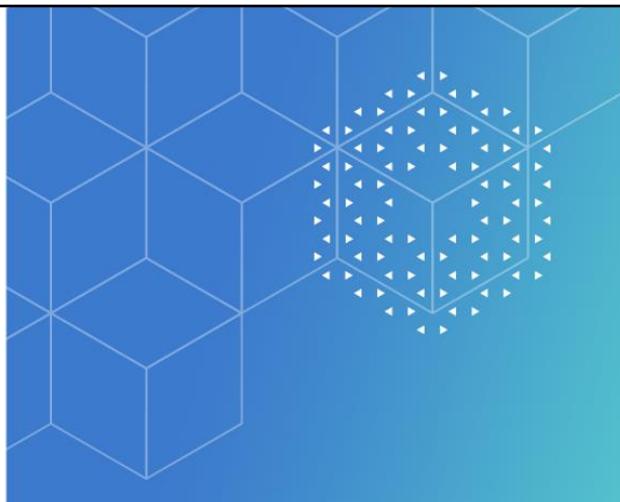
Module Summary

In this module, you:

- Identified the basic design flow of spec input, coding and output to synthesis.
- Listed the HDL history.
- Analyzed why we were still using RTL and identify the levels of abstraction.
- Identified register versus combinational logic with examples.
- Tested their memory on the memory state, latch FF memory cell.
- Defined the setup, hold and propagation delay.
- Identified and appreciated the different realistic design challenges such as Low-power, Size and area minimization, clock domains and crossing, etc.



This page does not contain notes.



Module 4

SystemVerilog Fundamentals for Design

cadence®

Welcome to the SystemVerilog Fundamentals for Design training module. This training course focuses on SystemVerilog constructs for RTL Design.

Module Objective

In this module, you will:

- Utilize simple SystemVerilog constructs for design.

Topics include:

- Design modules
- Standard data types
- Operators
- Procedural statements
- Blocking and nonblocking assignments
- User-defined data types
- Packages
- RTL Coding for Synthesis (synthesis review)
- Designing Finite State Machines (FSMs)



In this module we will utilize system Verilog constructs for writing RTL. We will go through all the mentioned topics to understand how these constructs work.

SystemVerilog Fundamentals for Design

SystemVerilog is the most commonly used HDL.

- A substantial upgrade on Verilog...
- ... which has a long history dating back to 1985

SystemVerilog contains:

- A huge number of constructs.
- Wide variety of options, alternative syntax, etc.
- Many redundant features (e.g., transistor level modeling).

SystemVerilog



Verilog

This course *only* covers the fundamentals for design.

- Specifically, the most useful and frequently used features.
- Best design practices where there are options.

SystemVerilog Fundamentals for Verification covers verification and testbench design constructs.



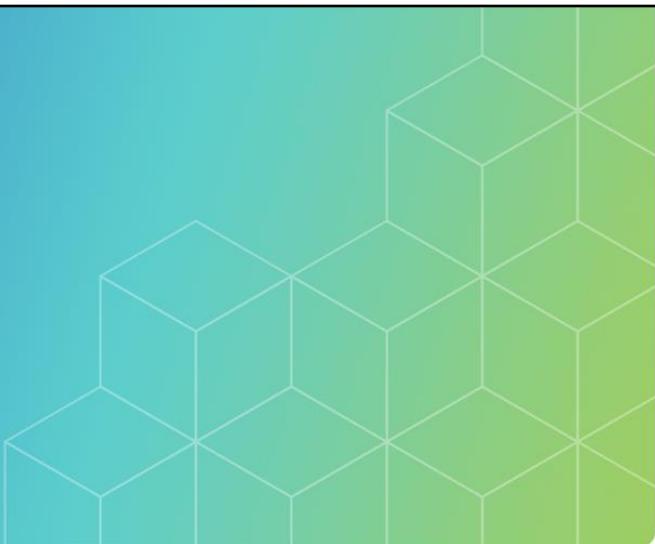
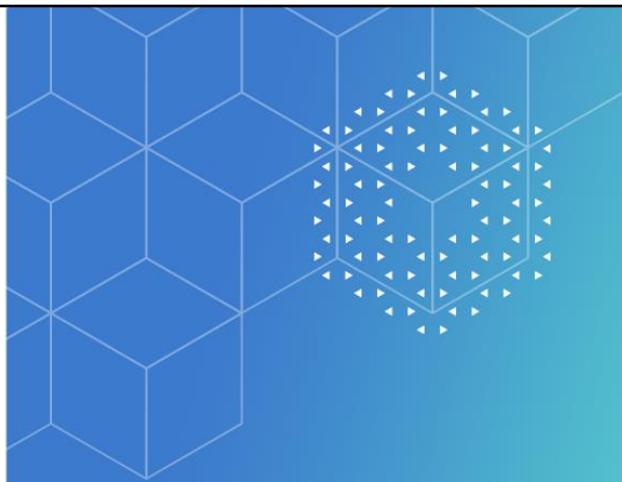
282 © Cadence Design Systems, Inc. All rights reserved.

SystemVerilog (SV) is an HDL with a complex history. SV is a substantial upgrade on the Verilog HDL. In fact, Verilog no longer exists as a separately defined language, it is a subset of SV. Verilog dates back to 1985, where it was originally designed as a switch-level modelling language for transistor circuit simulation.

Given the history of Verilog, and the requirement to make SV backwards compatible with Verilog (as far as possible) there is much duplication, alternative syntax, options and redundant features in SV. For example, there are usually many ways in the language to define even the most basic of language constructs such as a module declaration.

This course covers only the design features of SV, and we only cover the most useful and frequently used features. Where there are language syntax options, we only show the best design practice rather than all the possible options. Therefore, you may see SV code written differently than the examples in this course.

The separate *SystemVerilog Fundamentals for Verification* course covers verification and testbench design.



Submodule 4-1

Design Modules

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Use basic SystemVerilog constructs to describe a simple design.

Topics include:

- Describing design modules
- Rules for identifiers
- logic and bit data types
- Representing hierarchy
- Synchronizing module behaviors
- Communicating between behaviors
- Rules for comments and white space
- Compiling a design



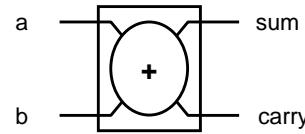
Your objective is to get started using SystemVerilog to describe the behavior of a digital design. To do that, you need to know some fundamental SystemVerilog language constructs.

Describing Design Modules

Start with the **module** keyword and identifier.

Define the module port list.

- Various syntax options.
- Simplest is ANSI-C format:
 - <direction> <type> <identifier(s)>



Describe the module behavior.

- assign outputs from an expression of inputs.

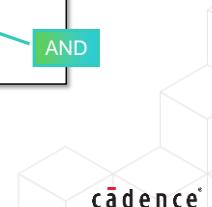
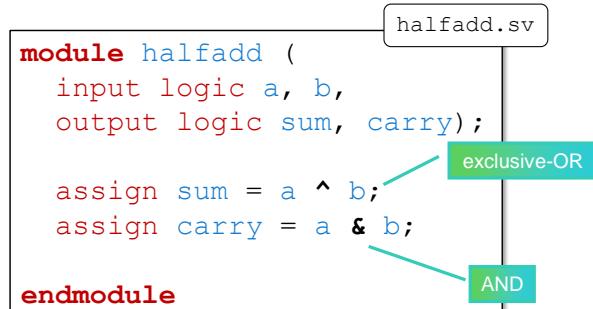
End with the **endmodule** keyword.

Save in a file with the **.sv** extension.

Note

- **Identifiers** are case-sensitive.
- **Keywords** are always lowercase.

285 © Cadence Design Systems, Inc. All rights reserved.



The basic building block of the design hierarchy is the module declaration. A module can represent the complete system, its major sub-blocks such as the CPU, further sub-blocks such as the ALU, and physical blocks such as an ASIC cell.

Each module declaration starts with the **module** keyword followed by a unique definition name. Most module declarations then follow with a list of port declarations. A module declaration then declares additional module items and concludes with the **endmodule** keyword.

Here is a module describing a half adder block. The module definition name is **halfadd**. There are many options to define the module port list, but the simplest is the ANSI-C format to declare the direction, type, and name of each port in a single location.

The module describes the half adder functionality by making two continuous assignment statements to the module output ports. The statements assign the results of Boolean expressions. Continuous assignments react to changes in their inputs. Upon any change of the **a** or **b** input ports, the simulator automatically recalculates the assignment expression and drives the new values to the output ports.

You should save the module in a file with the **.sv** extension so the compiler knows this file contains SystemVerilog code.

SystemVerilog is case-sensitive. All keywords are lowercase, and, by convention, most variable declarations are also lowercase.

Rules for Naming Identifiers

Identifiers start with a letter or an underscore (_).

- Followed by letters, digits, \$ or _

SystemVerilog does not restrict name length.

- Although tools or methodologies might.

Identifiers are case-sensitive.

- ABC, Abc, abc are all different legal names.

All keywords are lowercase.

Legal	unit_32 bus_16_bits abc\$
-------	---------------------------------

Not Legal	unit-32 16_bit_bus \$abc
-----------	--------------------------------

Convention

- Write everything in lowercase.
 - Exception: certain user-defined type values.

Escaped identifiers allow illegal names.

- Rarely used.

Escaped	\unit-32 \16_bit_bus \\$abc
---------	-----------------------------------

286 © Cadence Design Systems, Inc. All rights reserved.



You must start an identifier with an alphabetical character (a–z, A–Z) or an underscore (_). Your later characters can include any alphanumeric character, the dollar sign (\$), and the underscore.

The SystemVerilog language is, for the most part, sensitive to character case. Numbers and radices can be in either case. Keywords and the built-in system tasks and system functions are all lowercase.

By convention, everything in SystemVerilog is usually written in lowercase, but your company may have guidelines that differ from this convention. An exception to this rule is for certain user-defined type values which use uppercase to avoid name clashes with existing declarations.

An escaped identifier accepts any printable ASCII character in any position. You escape an identifier using a backslash (\) character prefix and a whitespace (space, tab, newline) character suffix. The backslash and white space are not part of the identifier.

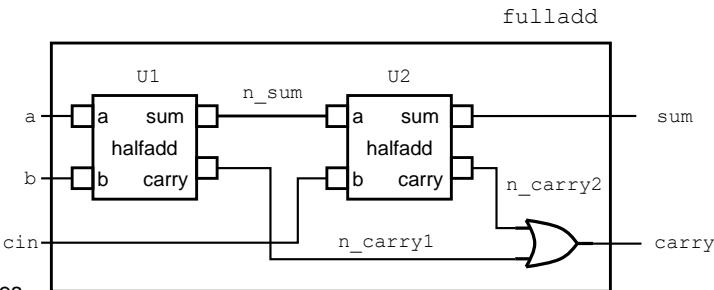
Escaped identifiers are used to fix tool compatibility issues with SystemVerilog but are rarely if ever, used anymore.

Representing Hierarchy

A full adder is 2 half adders + or operator.

To create the hierarchy:

- Declare local variables.
- Instantiate module(s):
 - Give each instance a unique name.
 - Connect instance ports to local ports and variables.
 - `.port(variable)`



```
module fulladd (input logic a, b, cin,
                  output logic sum, carry);

  logic n_sum, n_carry1, n_carry2; Local variables

  halfadd U1 (.a(a), .b(b), .sum(n_sum), .carry(n_carry1));
  halfadd U2 (.a(n_sum), .b(cin), .sum(sum), .carry(n_carry2));

  assign carry = n_carry1 | n_carry2; Port mapping

endmodule OR operator
```

287 © Cadence Design Systems, Inc. All rights reserved.

cadence®

You create hierarchy by declaring ports, variables, and module instances and connecting ports of the module instances to the locally declared ports and variables.

A full adder can be defined as two instances of the half adder and an `or` operator. One half adder instance (U1) adds the `a` and `b` inputs. The other instance (U2) adds the `sum` output of U1 to the `carry` input port to generate the `sum` output of the full adder. Finally, the full adder `carry` output is the `or` of the carry outputs of U1 and U2.

The full adder requires 3 internal connections. The `sum` from U1 to U2, and the two internal carries. These must be declared before they can be used.

The full adder then makes the two instantiations of the half adder module. Each instance is given an identifier that is unique within the current module (U1 and U2). We then map the full adder ports and local variables to the input and output ports of the half adder instances using a port map. There are several options, but the easiest is named connection, which uses the form `.port_name(variable)`. Where `port_name` is a port of the half adder module and `variable` is a local variable or port of the full adder module.

Connecting Hierarchy: Ordered Port Connection

Port mapping can be made by position.

- Local port, variable mapped in order of port declaration.

Very easy to map in the wrong order.

- Not recommended.

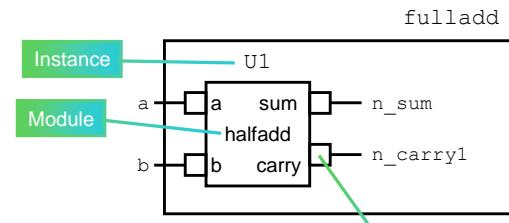
```
module fulladd (input logic a, b, cin,
                  output logic sum, carry);

    logic n_sum, n_carry1, n_carry2;

    halfadd U1 ( a, b, n_sum, n_carry1 );
    halfadd U2 ( n_sum, cin, sum, n_carry2 );
    ...

```

```
module halfadd (input logic a, b,
                  output logic sum, carry);
    assign sum = a ^ b;
    assign carry = a & b;
endmodule
```



Variable `n_carry1` of module `fulladd` mapped to output `carry` of instance `U1` of module `halfadd`

288 © Cadence Design Systems, Inc. All rights reserved.



A different option for port list connections is ordered port connection.

In ordered port connection, you simply list the full adder local variables or ports to be connected to the half adder instance ports in the same order that the half adder module declared the ports.

You can omit any port connection but need to retain the comma as a placeholder to maintain the order of connections.

Leaving an output port unconnected is common. Leaving an input port unconnected is usually an error. Leaving an input port unconnected feeds a high-impedance value into the module, which is treated as an unknown logic value.

With ordered port connections, it is very easy to map in the wrong order, and such mistakes are not easily visible as the original declaration of the port order is in a different module in a different file. Therefore, ordered port connections are likely to create incorrect connections. The more verbose named port connection solves these problems.

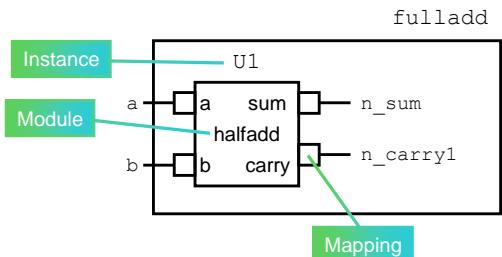
Connecting Hierarchy: Named Port Connection

Named port connection is much safer.

- `.port(variable)`

Where port and variable names match, there is a shortcut.

- `.sum = .sum(sum)`



```
module fulladd (input logic a, b, cin,
                 output logic sum, carry);

    logic n_sum, n_carry1, n_carry2;

    halfadd U1 (.a, .b, .sum(n_sum), .carry(n_carry1));
    halfadd U2 (.a(n_sum), .b(cin), .sum, .carry(n_carry2));
...
```

```
module halfadd (input logic a, b,
                 output logic sum, carry)
  assign sum = a ^ b;
  assign carry = a & b;
endmodule
```

Variable `n_carry1` of module `fulladd` mapped to output `carry` of instance `U1` of module `halfadd`

289 © Cadence Design Systems, Inc. All rights reserved.



Named port connection clearly identifies which port of the half adder instance is connected to which port of local variable of the full adder module.

Readability is better and incorrect connections are far less likely to be made.

However, the named port connection is very verbose. SystemVerilog allows a shortcut where the instance port name and the variable or port connection have the same identifier. In this case, you can simply write `.name`, which expands to `.name(name)`. This is called a “dot-name” connection.

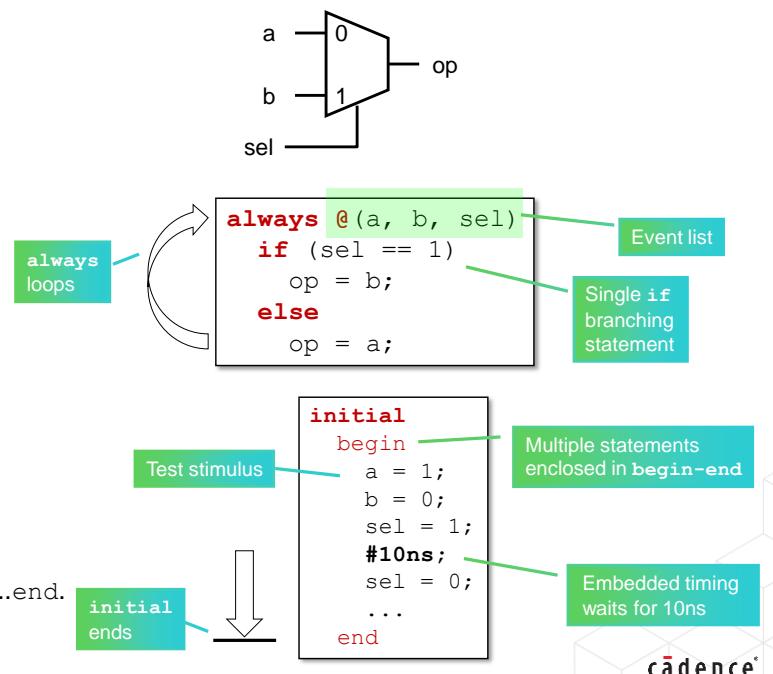
Procedural Blocks

Procedural blocks define complex behavior.

- For example, conditional and repetitive combinational logic.
- Registered logic.

Two general procedural blocks:

- always**
 - Executes at the start of simulation.
 - When at end, loops back to beginning.
 - Further execution controlled by the event list.
- initial**
 - Executes at start of simulation.
 - Embedded timing pauses execution.
 - When at end, terminates.
 - Testbench construct.
- Multiple procedural statements need `begin...end`.



290 © Cadence Design Systems, Inc. All rights reserved.

cadence

Previous examples described the half adder behavior by making continuous assignment statements.

To describe complex behavior, SystemVerilog provides procedural blocks which contain procedural statements. Procedural statements can define much more complex behavior such as conditional or looped functionality. You must also use procedural blocks to infer registered logic.

Multiple procedural statements within a block must be grouped within `begin` and `end` keywords. A single statement within a block does not require a `begin` and `end`.

SystemVerilog provides two general procedural blocks:

- always** starts executing at the start of simulation. Upon executing the last statement, execution loops back to the beginning of the construct. `always` typically has an event list that controls further execution. Any change in value for a variable in the event list retriggers the `always` execution. `always` may also (but rarely) contain embedded timing to suspend and resume execution.
- initial** starts executing at the start of simulation. Embedded timing within the block suspends and resumes execution. Upon executing the last statement, the `initial` terminates. `initial` is a testbench construct for applying a set number of simulation data.

Within a procedural block, the statements execute sequentially in their order of appearance.

A module can contain any number of procedural blocks, and no execution order is implied between them. Multiple procedural blocks triggered by a change in a variable value can execute in any order.

Synchronizing Block Execution

The @ event expression controls block execution.

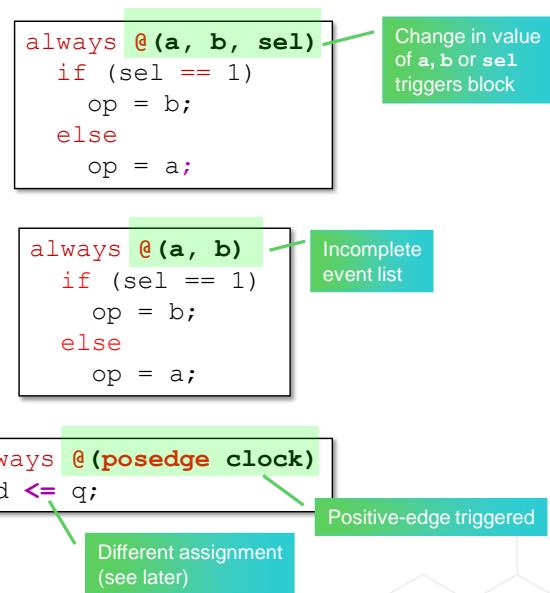
Change in value of any variable in expression triggers block.

Incorrect event expressions are an issue.

- For example, in combinational logic, event list must be *complete*.
 - Must contain all variables read in block.
- Otherwise, RTL and gate-level behavior differ.

Expressions can be edge-sensitive.

- Trigger on a specific transition.
- Use `posedge`, `negedge` or (rarely) `edge`.
- Essential for sequential (register) logic.



SystemVerilog provides procedural timing controls for triggering the execution of procedural blocks. The most common of these is event control. An event control starts with @ and then follows with either a wildcard (*) character, a single event identifier, or a parenthesized event expression. The event expression can be a list of event expressions separated by a comma (or historically by the keyword `or`).

Event expressions can be made sensitive to a specific transition, rather than any value change, by using `posedge` or `negedge` keywords. More on this later.

Specialist RTL Procedural Blocks

RTL code uses special always blocks

- Do **not** use initial or always

always_comb

- Combinational logic
- Implicit, complete event list
- Uses = assignment

always_ff

- Registered logic
- Requires an edge-triggered event list
 - Clock and reset signals only
- Uses <= assignment

Detailed descriptions later

292 © Cadence Design Systems, Inc. All rights reserved.

An always block can synthesize to combinational, latched, or sequential logic, depending upon the sensitivity list and your coding style. The simulator does not know what you intend, so cannot verify that your block matches your intentions.

SystemVerilog adds implementation-specific procedural blocks (always_comb, always_latch, always_ff).

The always combinational (always_comb) procedural block models combinational logic

- It infers a sensitivity list that includes every variable read by the procedural block. So, it is automatically complete.
- It cannot contain any embedded timing or event control.
- It uses blocking assignment (=). More on this later.

The always flip-flop (always_ff) procedural block model registered (sequential) logic

- It cannot contain any additional embedded timing or event control.
- It uses nonblocking assignment (<=) to infer registered logic. More on this later.

Both blocks prohibit their assigned variables from being driven anywhere else in the design. With these blocks, it is illegal for a variable to be written to by more than one driver, even if these drivers are in different modules or interfaces. This fixes a common issue in SystemVerilog design.

```
always_comb
  if (sel == 1)
    op = b;
  else
    op = a;
```

All variables read in block automatically added to event list

```
always_ff @ (posedge clock)
  d <= q;
```

Positive-edge triggered

Different assignment
(see later)



Rules for Comments and White Space

```
// A one-line comment starts with // and ends with newline character
/* A block comment starts anywhere with /*
   and ends anywhere with */

module muxadd (
    input logic a, b, sel,           // module inputs
    output logic sum, carry, y);

// SystemVerilog is a free-format language
// White space is needed only to separate some language tokens
// Use additional white space to enhance readability
assign sum    = a ^ b;
assign carry  = a & b;

// Also use indentation (2 space is best) to enhance readability
always @ (a, b, sel)
    if (sel == 1)
        y = b;
    else
        y = a;
...
```

Comments should be meaningful,
not stating the obvious

293 © Cadence Design Systems, Inc. All rights reserved.

cadence®

SystemVerilog is a free-format language. You can use white space to organize the code to enhance readability. SystemVerilog ignores these characters except where needed to separate other language tokens.

SystemVerilog has single-line comments and block comments:

- Single-line comments start with two consecutive forward slash characters (//) and terminate at the end of the line. The comment can include the whole line or any final part of it.
- A block comment starts with a forward slash character and following asterisk character /*) and terminates with the reverse – an asterisk character followed by a slash character (*). As a block comment ignores newline characters, you can place a block comment in a part of a line and across multiple lines. You do have to be careful that you do not use those character sequences within a block comment, as you cannot nest block comments.

You should, in general, not clutter your source with unneeded comments. Describe your module in a comment header at the top of the file and otherwise use comments only where absolutely necessary to clarify obscure fragments of your code.

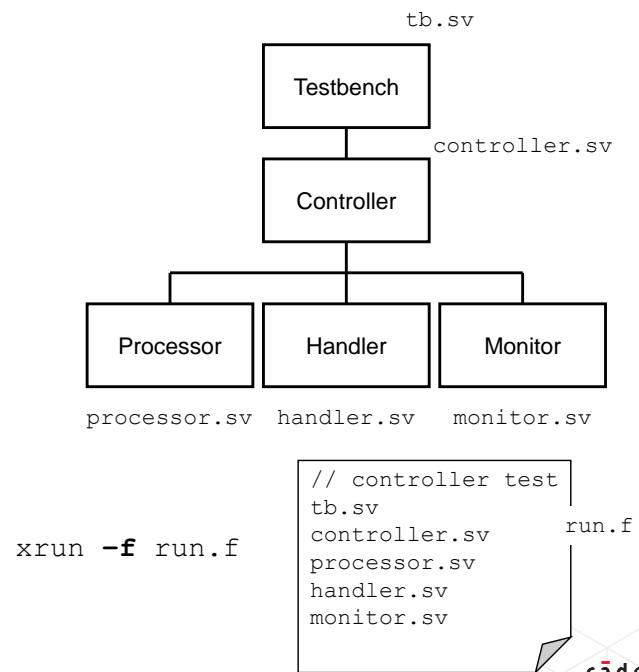
Use indentation and line breaks to make your code readable. You should indent your code a consistent two spaces for each indent. One space is easily missed, and more than two consume too much white space and eventually make many of your statements wrap over to a second line. You should, in general, place no more than one executable statement on a line.

Compiling the Design

Module compilation order is generally not important.

- One common exception is packages.
 - Contains declarations shared over multiple modules.
 - Must be compiled before modules that use the package.
 - More information on packages later.

Hint: use a compile file list.



294 © Cadence Design Systems, Inc. All rights reserved.

This diagram shows a complete hierarchical design and testbench.

- The behavioral testbench sits at the top level of the hierarchy. The testbench instantiates the design module and contains test code.
- The structural design module instantiates the three primary design blocks and contains no design code of its own.

You can normally compile these modules in any order. The files are individually compiled, and then an elaboration step links together the compiled module descriptions to build the design and testbench hierarchy for simulation.

A realistic compilation of a design will need to compile many different files with several simulator options. You could type a file list every time on the command line, but it's much more convenient to place the compile file names and simulator options in a separate file, called a run file, and reference this file with the `-f` option. Your run file can contain embedded comments using SystemVerilog syntax, and additional simulator options can be added to the command line for added flexibility. For example, it is common to place omit the options for a GUI simulator run from the run file, so the user can switch between batch mode and GUI simulation simply by adding the options directly to the command line.

Submodule Summary

You should now be able to use basic SystemVerilog constructs to describe a simple design.

This module briefly introduced:

- Describing design modules (the `module` keyword).
- Representing hierarchy (instantiation and port connection).
- Describing module behavior (procedural blocks).
 - `always` and `initial` for testbenches.
 - `always_comb` and `always_ff` for RTL.
- Rules for identifiers, comments, white space.
- Compiling a design.



This module examined the fundamental language constructs and how you use them to describe a design.

Test Your Understanding

1. What is the basic building block of a SystemVerilog design?
2. What is the fundamental difference between `always` and `initial`?
3. Which of the following identifiers for an active low reset are legal?

```
_reset  
reset~  
reset!  
n-reset  
Reset  
re$et  
/reset
```



Please pause here for a moment to consider these questions. Refer to the module contents as needed. When you are ready, compare your answers to those on the next slide.

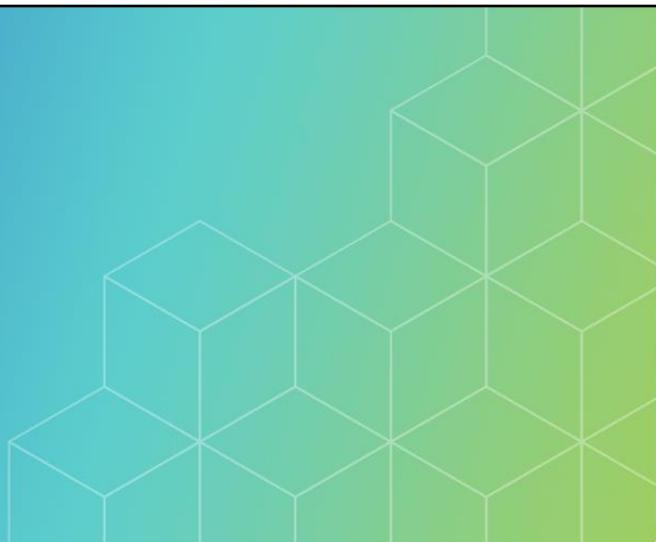
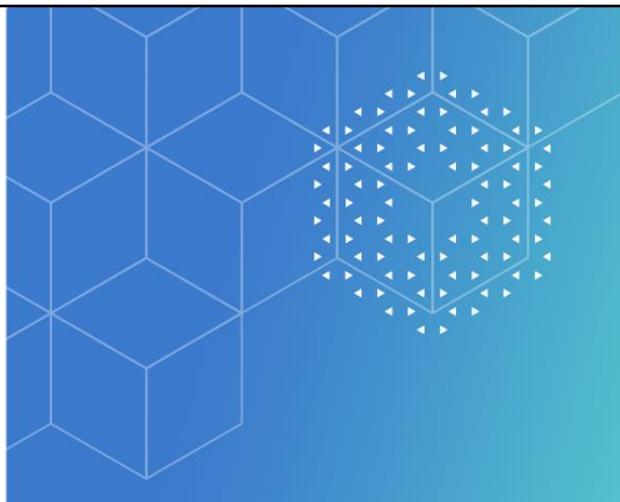
Solutions: Test Your Understanding

1. What is the basic building block of a SystemVerilog design?
 - module
2. What is the fundamental difference between `always` and `initial`?
 - When an `always` execution reaches the end, it loops back to the beginning, whereas when an `initial` execution reaches the end, it terminates.
3. Which of the following identifiers for an active low reset are legal?

<code>_reset</code>	legal
<code>reset~</code>	not legal ~ character not allowed in identifier
<code>reset!</code>	not legal ! character not allowed in identifier
<code>n-reset</code>	not legal – character not allowed in identifier
<code>Reset</code>	legal, although remember SystemVerilog is case sensitive
<code>re\$et</code>	legal
<code>/reset</code>	not legal / character not allowed in identifier. Escaped names use \



Here are the answers to the questions.



Submodule 4-2

Standard SystemVerilog Types

cadence®

This submodule examines the SystemVerilog value set and data types. It explores nets, variables, and a form of constant called a parameter.

Submodule Objective

In this submodule, you will:

- Use SystemVerilog data types correctly.

Topics include:

- Logic values
- Data types
 - Variables
 - Nets
- Declaring vectors
 - Truncation and padding
- Defining literal values
- Constants



Your objective is to appropriately choose and effectively use the SystemVerilog data types. To do that, you need to know what values can be represented, and how to represent those values using literals, constants, variables, and nets, and how to create aggregates.

Value Sets

Value	Associated Informal Terms
0	Zero, Low, False
1	One, High, True
Z	High Impedance, Tri-State, Undriven,
X	Uninitialized, Unknown (bus contention)

logic

bit

Limited use
in RTL

4-state `logic` variables initialize to `x` at the start of simulation.

- Helps detect initialization/reset issues.
 - Variables that remain at `x` have not been reset correctly.

2-state `bit` variables initialize to 0.

- Hides initialization issues.
- Used for RTL *only* in very limited situations.

300 © Cadence Design Systems, Inc. All rights reserved.



The SystemVerilog logic value set consists of the four basic values:

- 0 – to represent a logic zero, low, or false condition;
- 1 – to represent a logic one, high, or true condition;
- z – to represent a high-impedance state; and
- x – to represent an unknown logic value.

All `logic` variables are initialized to `x` at the start of the simulation. A variable that is never assigned a value will remain at `x` throughout the simulation. This helps detect initialization or reset issues in your design.

A high-impedance value `z` is usually due to drivers being disabled. In real hardware, this situation either has a short duration or does not occur because special logic pulls the net to a high or low logic state. The exception is for variables that are declared but never assigned. These are initialized to `z`.

An unknown value during simulation is usually due to a clash between drivers driving different values. In real hardware, this situation will either not exist or have an extremely short duration.

An alternative to `logic` is `bit`, which only has the 0 and 1 values. All `bit` variables are initialized to 0 at the start of the simulation. As this is a known value, using `bit` makes it much harder to detect initialization and reset issues in your design. Therefore, `bit` types have very limited use in RTL.

RTL Data Types

SystemVerilog provides three “data types” for RTL design.

- Variables
 - var
 - General purpose RTL use.
- Nets
 - E.g., wire
 - Include resolution tables to resolve multiple drivers.
 - Used for multiply-driven connections *only*.
 - E.g., tri-states and bidirectionals.
- Constants
 - True constants (`localparam`).
 - Instance-specific constants (`parameter`).
 - Can be overridden for an instance basis giving greater flexibility.



SystemVerilog has three “classes” of value objects and only a very few types in each class:

- Variables for general purpose RTL connections
 - `var`
- Nets to represent connections with multiple drivers, such as bidirectional and tristate buses. nets have resolution tables to resolve multiple drivers.
 - `wire` is the most common net type.
- Constant forms
 - `localparam` is a true constants which cannot change.
 - `parameter` is an instance-specific constant, which can be modified for specific instances allowing generic, customization code blocks to be created.

Variable Rules

All `logic` types are variables by default.

- The variable keyword `var` is rarely required.

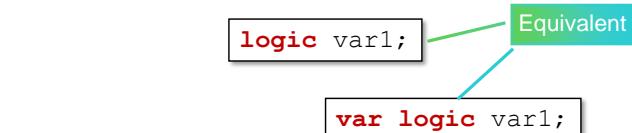
Variables can only have a single driver.

One of the following:

- An `assign` statement.
- A module output port.
 - In an instantiation port map.
- An `always_comb` block.
- An `always_ff` block.

Multiple drivers on a variable give compilation errors.

- If you *need* multiple drivers, declare a net type.
 - E.g., `wire`



Error

```
module test;
  logic in1, in2, in3;
  logic op;
  mone u1 (op, in1, in2);
  mtwo u2 (op, in1, in3);
endmodule
```

Multiple drivers on `logic op` illegal

✓

```
module test;
  logic in1, in2, in3;
  wire op;
  mone u1 (op, in1, in2);
  mtwo u2 (op, in1, in3);
endmodule
```

302 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The full declaration of a signal uses the form:

<data type> <value set> <identifier>;

E.g., `var logic var1;`

However, `logic` types are variables by default, so, by convention, the `var` keyword is not used in declarations. There are some corner syntax cases where it is required, but these are very rare.

Variables can only have a single driver. We can drive a variable from an `assign` statement, from a module output port in an instantiation port map, or from an `always_comb` or `always_ff` procedural block. You can only use one of these options to drive a specific variable. It is a compilation error to have multiple drivers on a variable. For multiply-driven buses, you must use a net datatype like `wire` (see later).

There is an exception to the rule in that you can assign a variable from multiple `always` or `initial` blocks. This is for backward compatibility with Verilog. However, if you exclusively use `always_comb` or `always_ff` in RTL code, these specialist blocks enforce the “single driver per variable” rule.

Having multiple accidental drivers on a connection is a common problem in RTL design which can be difficult to debug. However, by only using variables and `always_ff` or `always_comb` in RTL code, we can guarantee the compiler will generate errors for multiply driven variables.

Net Data Types

Nets are used where multiple drivers are required.

- E.g., tristate, bidirectional.
- Always 4-state.

Include resolution tables to resolve multiple drivers.

Many different net types.

- `wire` is the only one in common use.

Net types can only be driven by:

- `assign` statements.
 - Module output or inout ports.
- Cannot* be driven from a procedural block.
- `initial` or *any* form of `always`.

Multiple drivers require
`wire`

```
logic ena1, ena2, data1, data2;
wire dataout;

// if ena1 true, drive data1 else drive Z
assign dataout = ena1 ? data1 : 1'bz;
assign dataout = ena2 ? data2 : 1'bz;
```

`assign` if statement

Output port driver OK

Assign OK

Net cannot be driven
from procedural block

`wire op;`

mone u1 (`op`, `in1`, `in2`);

assign `op` = `a` ^ `b`;

always_comb
`op` = `a` & `b`;

Error

303 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Net types are only used in SystemVerilog for multiply-driven connections such as tristate or bidirectional buses. Net types are always of type `logic` as they require the `x` and `z` values to resolve multiple, conflicting drivers.

A tristate driver is inferred by conditionally assigning a `wire` to the hi-impedance value `z`.

Net types have resolution tables to resolve the value of a bus from multiple drivers. There are many different net types (many of them historical and no longer used). `wire` is the only net type in common use for RTL.

Net types have a major restriction in that they cannot be driven from any procedural block. They can only be driven from `assign` statements or from module output or inout ports.

As we cannot drive wires from procedural blocks, we cannot use the procedural `if` statement to conditionally assign the wire to `z`. We must have a “short-cut” version of the `if` statement. The syntax for this is as follows:

```
result = <condition> ? <true value> : <false value>;
```

For example, if `condition` is true (1), `result` is assigned from the value or expression before the colon character. If `condition` is 0 or unknown, `result` is assigned from the value or expression after the colon character.

Net Data Type Resolution

Only a net can resolve the value of multiple drivers
wire resolution:

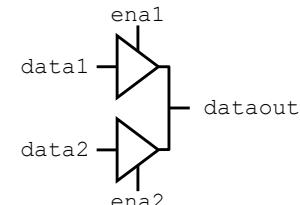
- Simultaneous drive of 0 and 1 results in unknown (X).
- Simultaneous drive of 0 and Z results in 0.
 - For modelling tri-states.

Other resolutions exist, but rarely used.

- E.g., `wand` models wired-AND logic.

```
logic ena1, ena2, data1, data2;
wire dataout;

assign dataout = ena1 ? data1 : 1'bz;
assign dataout = ena2 ? data2 : 1'bz;
```



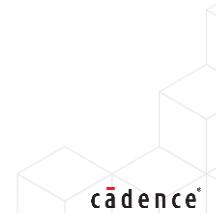
		data2			
		0	1	Z	X
data1	0	0	0	0	0
	1	0	1	1	X
	Z	0	1	Z	X
	X	0	X	X	X

wand resolution

		data2			
		0	1	Z	X
data1	0	0	X	0	X
	1	X	1	1	X
	Z	0	1	Z	X
	X	X	X	X	X

wire resolution

304 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog resolves the value of a net driven by multiple drivers. Assuming that the drivers all have the same strength, the conflict of 0 and 1 values on a `wire` net results in the unknown value X.

SystemVerilog inherited many different net types from Verilog, reflecting Verilog's original as a switch-level modeling language. However, only `wire` is generally used, and, in SystemVerilog, only for multiply-driven connections.

Different net types allow different resolution tables.

E.g., wired-logic nets exist to model technology-dependent logic conflict resolution:

- The wired-AND net type to model open collector logic; and
- The wired-OR net type to model emitter-coupled logic.

The conflict of 0 and 1 values on a wired-and net results in a 0 value.

Default and Implicit Types

Declarations without a data type default to `wire`.

- `wire` allows multiple drivers.
- Lose the “single driver” compilation check.

You should fully declare all variables.

```
module halfadd (
    input logic a, b,
    output sum, carry);
    ...

```

Variables

No data type –
sum and carry
default to wire

In some constructs, undeclared identifiers are allowed.

- Called implicit declarations.
- Declared as a single bit `wire`.

Check identifiers carefully.

- Typos can lead to broken connections.

```
module fulladd (input logic a, b, cin,
                 output logic sum, carry);
    logic n_sum, n_carry1;
    halfadd U1 (.a(a), .b(b),
                .sum(n_sum),
                .carry(n_carry1));
    ...

```

Typo in variable name
implicitly declares wire

305 © Cadence Design Systems, Inc. All rights reserved.



There are implicit and default declaration rules for backward compatibility with Verilog.

If you do not define the data type, it defaults to `wire`. As `wire` is allowed to have multiple drivers, and multiple accidental drivers are a common issue in SystemVerilog design, you should avoid default `wire` declarations by fully declaring all variables using the `logic` keyword.

In certain language constructs, undeclared identifiers are allowed. Port maps for module instantiations is one example. If you use an identifier without a declaration, it is implicitly declared as a single-bit `wire`. This, again, loses the multiple-driver detection of `logic` variables. Even worse, a typo in an identifier name will not generate a compiler error but will implicitly declare a `wire`, which can lead to broken connections between modules.

Remember, an undriven `wire` defaults to the value `z`, so this feature can help detect broken connections.

Declaring Vectors

A vector is declared an array.

- Collection of individual bits.
- Defined with a range specification.
- Indexed with an integer.

Define the range when declaring the variable:

- Bounds can be descending or ascending.
 - [msb : lsb] or [lsb : msb]
- Bounds can be negative, zero, or positive.
- Bounds can be expressions...
 - ...but must be constant and known at start of simulation.

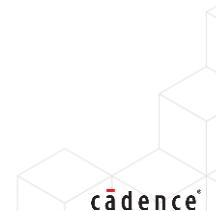
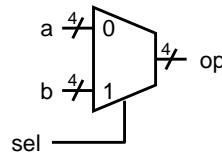
By convention, bounds are descending to 0.

- E.g., [3 : 0]

```
module mux4 (
  input logic [3:0] a, b,
  input logic sel,
  output logic [3:0] op
);

  always_comb
    if (sel == 1)
      op = b;
    else
      op = a;

endmodule
```



306 © Cadence Design Systems, Inc. All rights reserved.

Ports, nets, and variables are a single bit unless you declare them with a range. A scalar variable is a one bit variable. The reg, logic, and bit data types default to one-bit scalars. A vector is an array of consecutive bits. The standard refers to vector as a packed array. The reg, or, logic, or bit data types can be used to represent a vector of any size. You can specify a range for declaring vectors inside square brackets.

The range provides addresses for the individual bits. The only restriction on the range bounds is that they must be constant expressions. Either or both bound, can be negative, zero, or positive, and the range can be ascending or descending.

By convention, most bounds are declared in ascending direction, with a left bound representing the Most Significant Bit (msb), and the right bound Least Significant Bit (lsb) being 0.

Using Vector Ranges

A vector can be sliced with a range of one or more contiguous bits.

- Without a range, the whole vector is selected.

Elements are assigned in order of declaration.

- Unselected elements are unchanged.

Slice must be in the same direction as the declaration.

```
logic [3:0] idec;
logic [0:3] oasc;

assign oreg = ireg;
// osc[0] = idec[3]
// osc[1] = idec[2]
// osc[2] = idec[1]
// osc[3] = idec[0]

assign oreg[2:0] = ireg[2:0];
```

Assigned in order
of declaration



Error

```
logic [3:0] inp;
logic [3:0] outp;

assign outp = inp;
// outp[3] = inp[3]
// outp[2] = inp[2]
// outp[1] = inp[1]
// outp[0] = inp[0]

assign outp[3] = inp[0];
// outp[3] = inp[0]

assign outp[3:0] = inp[1:0];
// outp[3] = inp[1]
// outp[2] = inp[0]
```

Without range,
whole vector selected

Assigned in order
of declaration

Individual element

Slice

Cannot slice ascending vector
with descending bounds

307 © Cadence Design Systems, Inc. All rights reserved.



The vector range provides addresses for the individual bits. You can address a vector bit by providing an index, and a vector slice, by providing a range. The slice selection range must be in the same order as in the declaration, i.e.. either ascending (by convention) or descending. If range is not provided, then whole vector is considered.

In the right example, we have two same size vectors input and output. If they are assigned to each other without specifying a range, then, each index element of input will be assigned to same index element of output. We can also choose to select individual elements of vector. Or we can also select vector slice, necessarily slice size need not be the same.

In the left example, we have two vectors declared with same size but different index. When input reg is assigned to output reg, then, they are assigned from msb to lsb respectively, as shown. In the last line, it is wrong to slice ascending vector with descending bounds.

Assigning Between Different Widths

Vector widths do not need to match in an assignment!

More on signed/unsigned vectors later

- If the source is wider than the target, the value truncated is from the left-most bit.
- If the unsigned source is shorter than the target, the value zero-extended is from the left-most bit.
- If the signed source is shorter than the target, the value is sign-extended.
 - Selections and concatenations are not considered signed.

```
logic [3:0] zbus;      // 4 bits
logic [5:0] widebus; // 6 bits

always_comb
  zbus = widebus;    // same as
  //           <- widebus[5]
  //           <- widebus[4]
  // zbus[3] <- widebus[3]
  // zbus[2] <- widebus[2]
  // zbus[1] <- widebus[1]
  // zbus[0] <- widebus[0]
```

```
logic [3:0] zbus;      // 4 bits
logic [5:0] widebus; // 6 bits

always_comb
  widebus = zbus; // same as
  // widebus[5] <- 0
  // widebus[4] <- 0
  // widebus[3] <- zbus[3]
  // widebus[2] <- zbus[2]
  // widebus[1] <- zbus[1]
  // widebus[0] <- zbus[0]
```

308 © Cadence Design Systems, Inc. All rights reserved.



You can assign between vectors of different widths:

- SystemVerilog truncates from leftmost bits when assigning a wider vector to a narrower vector. If you want to select some range other than the rightmost bits, then you must specify that range.
- SystemVerilog pads the leftmost bits with 0 when assigning a narrower vector to a wider vector. If you want to assign something other than 0 then you need to use the concatenation operator to construct a wider expression
- If you have defined explicitly signed vectors (see later), then padding is done with the sign bit of the shorter vector to maintain sign information.

Here, in the left example, when the wide bus is assigned to the z bus, 3 down to 0 bits of the wide bus will be assigned to 3 down to 0 bits of the z bus.

Here, in right example, since a narrower vector is assigned to the wider vector, the remaining bits of wider vector will be assigned with 0.

Array Types and Dimensions

2

1

Indexing order

SystemVerilog allows arrays:

- Of any type.
- With any number of dimensions.

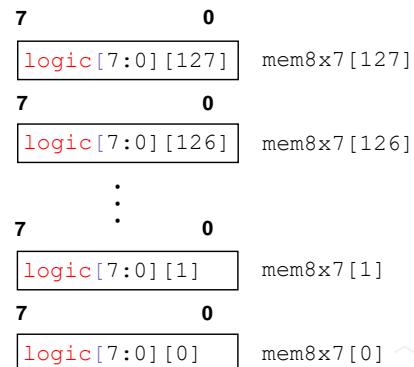
```
logic[7:0] mem8x7 [127:0];  
  
// mem8          = array of bytes  
// mem8x7[0]      = 0th byte  
// mem8x7[0][0]   = 0th bit of 0th byte
```

Multi-dimensional arrays in RTL are used for:

- Modeling memories.
- Declaring register arrays.

Therefore, the typical RTL arrays are 2D.

- Address index declared after name.
- Each array element is stored as a separate variable.
- Indexing priority is address first, then data.



309 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog supports arrays of any type and with any number of dimensions.

Typically, in RTL code, multi-dimensional arrays are restricted to 2 dimensions to model memories or register arrays.

With simple 2-d arrays, the "data" dimension is declared to the left of the array variable name, and the "address" dimension is declared after the name to the right. Indexing priority is "address" first, then "data". So, a single index on `mem8x7` will apply to the `[127:0]` range to access one byte, and a second index will apply to the `[7:0]` range to access a single logic element.

Each element of this form of an array is stored as a separate variable in the simulator and, by default, will synthesize to separate connections, e.g., 128 separate bytes. Your synthesis tool may need further guidance to map the array to an embedded memory.

SystemVerilog also has the option of packed arrays which can synthesize down to a single variable, but packed arrays are beyond the scope of this course.

Defining Literal Values

You can specify a literal value as:

```
<size>'<base><value>
```

- size is an optional positive decimal number of bits.
 - If omitted, is at least 32 bits.
- base is a character to indicate binary, octal, decimal, or hexadecimal radix.
 - B/b, O/o, D/d, H/h.
 - If omitted, defaults to decimal.
- value is legal digits for base.
 - Can include underscores “_” if not the first character.
 - Non-consecutive.
 - Can include Z/z and X/x digits if the base is binary, octal, or hexadecimal.

```
...
logic [3:0] abus;
...
abus = 4'b1001; // 1001
abus = 4'd14;   // 1110
abus = 4'h2f;   // 1111
...
```

More Examples

8'b1100_0001	8-bit binary
10'd1000	10-bit decimal
16'hff01	16-bit hexadecimal
12	32-bit decimal
'h83a	32-bit hexadecimal



A fully-defined literal has three parts – a size, base, and value, with no whitespace between the parts.

size specifies the width of the literal in bits. If omitted, it defaults to at least 32 bits.

base is the radix of the literal represented as a single character. Binary for b, Octal for o, decimal for d, and hexadecimal for h. The character can be upper or lowercase. If omitted, base defaults to decimal.

value is the literal value in legal digits according to the base. You can embed underscores within the value, as long as they are non-consecutive. For any base other than decimal, unknown z or x bits are allowed in the value.

The top right example declares an array of width 4 bits. You can assign the value in various formats as shown above.

Literal Assignment Rules (Unsigned)

Size and value of a literal need not match the target.

Value is extended or truncated to literal size.

- Extended with 0 if leftmost bit is 0 or 1.
- Extended with leftmost bit if Z or X.

Value then further extended or truncated to the target size.

- According to normal rules.

Mismatches can lead to unexpected values.

Avoid zero-extension using unsized literals.

- If size omitted, literal is sized to target.
- If base also omitted, value fills entire vector.

```
logic [5:0] databus;

// zero value extension
databus = 6'b0; // 000000
databus = 6'b1; // 000001

// leftmost bit value extension
databus = 6'bz; // zzzzzz
databus = 6'bx; // xxxxxx

// value truncation by literal size
databus = 6'hff; // 111111

// under-sized value zero-extended
databus = 4'bx; // 00xxxx

// unsized value extended to target
databus = 'bx; // xxxxxx

// unsized, unbased value
databus = '1; // 111111
```

311 © Cadence Design Systems, Inc. All rights reserved.



You need to be careful with sizing literals. The size and value need not match the target variable or net.

First, the value is extended or truncated to match the literal size. If the value is greater than the size, then, value is truncated to the size number of bits from the left-hand side. If the value is smaller than the size, then value is extended according to the following rules.

- If the leftmost bit of the value is 1, then the value is padded with 0 up to the size.
- If the leftmost bit is not 1, then the value is padded with the leftmost bit up to the size. For example, for the literal 6'bz. The size is 6, the value is a single z bit, so value is extended to 6'bzzzzz. This makes it easy to fill an entire vector with 0, x or z.

Once the literal has been sized, then it can be used in an expression or assignment. If the expression result or literal does not match the target size in the assignment, then the normal truncation and padding rules apply as described previously. You should match your literal size to the target size to avoid issues.

One option to avoid size mismatches between literal and target in an assignment is to use unsized literals. By omitting the size, the literal value is automatically sized to the target, which can avoid zero-extension.

You could also omit the base. An unsized, un-based literal value fills the entire vector. In an exception to the normal rules, an unsized, un-based value of 1 will be extended to fill the entire target vector with 1s.

integer and int

Predefined vector variables:

- `integer` – 4-state signed 32-bit (`logic`).
- `int` – 2-state signed 32-bit (`bit`).

Used for certain constructs in RTL.

- E.g., loop indices.

Avoid using as variables in RTL code

- `integer` and `int` are signed.
 - 2s complement.
- `logic` and `bit` vectors are unsigned.
- Assignment between `logic` vectors and `integer` can lose sign information.
- Normal truncation and extension rules apply.

```
integer i32;
logic [3:0] vec4;

initial begin
    i32 = 17;      // 00...10001
    vec4 = i32;    // 0001
    i32 = vec4;   // 00...00001
    i32 = -3;     // 11...11101
    vec4 = i32;    // 1101
    i32 = vec4;   // 00...01101
end
```

312 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog declares predefined 32-bit arrays of 4-state `logic` type, called `integer`, and 2-state `bit` type called `int`.

These can be useful in certain RTL constructs, such as loop indices, but should be generally avoided, especially for variables, in RTL code.

Firstly, because they are 32-bit, and will synthesize, at least initially, to 32-bit buses.

Secondly, because `int` is 2-state, and will initialize to 0. We should never use 2-state types for RTL variables.

Thirdly, because both are signed (2s complement), whereas user-defined arrays of `bit` and `logic` are unsigned (by default). Assignments between `integer` and user-defined arrays will obey the normal padding and truncation rules in SystemVerilog, but signed values in `integer`, add an extra layer of complexity.

This example assigns the value 17 to the `integer` `i32`, and assigns `i32` to a 4-bit unsigned vector `vec 4`. Truncation means `vec4` is assigned 1. On assigning `vec4` back to the `i32`, zero-extension assigns `i32` to 1.

The example then assigns the value minus 3 to `i32` and assigns `i32` to `vec4`. Truncation assigns 13 to `vec4`. The sign of the negative value is lost. On assigning `vec4` back to `i32`, zero-extension assigns 13 to `i32`.

Signed Vectors

logic vectors are unsigned.

- Binary.

`signed` defines vector types as signed quantities

- 2s complement.
- Easiest solution for signed arithmetic.

Assignments between `signed` and `integer` types maintain sign information.

Vector types can also be defined as `unsigned`.

- Default behavior.

```
logic      [3:0] usvec; // 4-bit vector 0 to 15
logic signed [3:0] svec; // 4-bit vector -8 to 7
```

```
logic signed [7:0] svec8;
logic [7:0] usvec8;
integer i32;

initial begin
    svec8 = 8'b11001101;      // -51
    usvec8 = svec8;           // 205
    i32 = -42;
    svec8 = i32;              // -42
    usvec8 = i32;              // 214
    ...

```

313 © Cadence Design Systems, Inc. All rights reserved.



By default, user-defined arrays of logic are unsigned, i.e., binary. To handle signed arithmetic, you can define any user-defined array as 2s complement using the qualifier `signed`.

Signed vectors still follow normal truncation rules, which means a negative value could be turned into a positive simply by truncation.

However, for extension, signed vectors should be sign-extended. i.e., the most significant bit, holding the sign information, is used to extend a shorter signed vector to a larger signed vector target.

For clarification, if you are mixing signed and unsigned vectors, binary vectors can be explicitly defined as `unsigned`, although this is the default.

In the example here, we have signed `vec8` and unsigned `vec8`. Initially, we are assigning minus 51 to signed `vec8`. and, when signed `vec8` value is assigned to unsigned `vec8`, it loses its sign, and now, the value in unsigned `vec8` becomes 205. Then, `i32` is stored with a value of minus 42. Since `i32` is declared as an integer, remaining bits of msb are sign extended. So, when that value is assigned to signed `vec8`, even though its truncated, the value is protected. Again, when it is assigned back to unsigned `vec8`, it loses its sign information.

Declaring Module Parameters

A module parameter is an instance-specific constant.

- Parameterizes the module definition.
 - Width, depth, etc.
- Has a default value in declaration.
- Type is derived from value.
- Can be overridden for each individual instance.
 - Using a named parameter override.
 - Could also use positional override.

```
module mux
#(parameter WIDTH = 2)
(input logic [WIDTH-1:0] a, b,
 input logic sel,
 output logic [WIDTH-1:0] op);

always_comb
  if (sel)
    op = a;
  else
    op = b;
endmodule
```

Generic MUX
default width 2

Allows generic modules.

- Scaled on instantiation.

Tip: use uppercase identifier for parameter.

- Avoids clash with local identifiers.
- Helps readability.

```
logic [1:0] a2, b2, op2;
logic [3:0] a4, b4, op4;
logic sel;

mux mux2 (.a(a2), .b(b2), .sel, .op(op2));

mux #( .WIDTH(4) ) mux4 (.a(a4), .b(b4), .sel, .op(op4));
```

Default width 2

Named override and sets width to 4

314 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Module parameters that you declare with the `parameter` keyword are constants that you can change for each instance of the module; thus, you can use them to “parameterize” the instance, for example, to establish different widths and depths for each instance of a memory module. Parameters are not variables, so you cannot change them during the simulation runtime.

Parameters can be used to size module ports but must be declared before they are used in the port list. SystemVerilog has an optional parameter declaration scope before the module port list, identified by the character `#`. This is a list of parameter declarations, in parentheses, separated by commas.

Each parameter declaration must define a default value for the parameter, which will be used if the value is not changed in an instantiation. The parameter declaration does not need a type – the type is derived from the default value.

The parameter value can be overridden for each instance of the module in a similar way to port mapping. Like port mapping, you can use named association to override selected or all parameters for the module. Ordered association is also possible but less readable for modules with multiple parameters.

It is common to use uppercase to declare parameter names. Remember SystemVerilog is case-sensitive. Uppercase parameter names avoid clashes with local declarations using the same name and are more readable.

Here, in the above code snippet, we have declared a parameter with default value of 2. That parameter value is used to declare the module ports. In the above code, the MUX module is instantiated as `mux2` with the default parameter value. In the next line it is instantiated as `mux4` and overridden parameter value with 4.

Local Parameters

A localparam is a true constant.

- Unlike parameter, localparam cannot be *directly* overridden hierarchically.
 - Although, it can be derived from parameter values.

```
localparam DWIDTH = 16;
localparam AWIDTH = 6;
localparam MSB = 7;
localparam MEMSIZE = DWIDTH * (1 << AWIDTH);
```

Use for:

- Constants that should not change.
- Naming literals or expressions.
- Deriving values from parameters.
 - Where these values should not be overridden.

Can also be declared in parameter list of module.

- For use in port list.

```
module us_mult
#(parameter WIDTH_A = 4, WIDTH_B = 4,
  localparam WIDTH_OP = WIDTH_A + WIDTH_B)
(input logic [WIDTH_A-1:0] a,
 input logic [WIDTH_B-1:0] b,
 output logic [WIDTH_OP-1:0] op);

assign op = a * b;

endmodule
```

315 © Cadence Design Systems, Inc. All rights reserved.



A localparam construct is exactly like a parameter, except that you cannot directly override it. For a module parameter that the user should not change, you should use localparam instead of parameter.

Local parameters can be used as true constants, for values that will never change, or to name a literal or constant expression.

Local parameters can also be used to size port lists or arrays by deriving a value from parameters. In the example, the user can parameterize the width of the two inputs, but the output width is the addition of the two input widths. We don't want to declare the output width as a parameter, because that would allow the user to override the value, potential with the wrong value, and hence break the design. We use a localparam to derive the output width from the input parameters and protect the value from user modification.

As we need to use the output width in the module port list, we can declare local parameters in the parameter scope, identified with the # character.

Submodule Summary

In RTL, most signals are declared as 4-state logic variables.

- Exception: multiply-driven signals must be declared as nets (wire).

Declarations must be explicit.

- Ports that omit logic or identifiers without a declaration default to wire.

Vectors declared as arrays with bounds.

- By convention, descending to 0.

Automatic truncation/extension used for array assignments of different sizes.

Literals declared with `<size>'<base><value>`.

- Always size to target or use unsized literals.

Parameters are instance-specific constants.

Local parameters are true constants.

You can declare multi-dimensional arrays of any type.



In summary, we understood 2-state and 4-state logic variables. We also learned how to declare and use literals, arrays, parameters and local parameters.

Test Your Understanding

1. What are the four values in the `logic` value set?
2. Why do we use `logic` types rather than `bit` in RTL code?
3. Rewrite the binary value `8'b11010011` in hexadecimal.
4. What is the primary difference between a `wire` net and a `logic` variable for RTL code?



Please pause here for a moment to consider these questions. Refer to the module contents as needed. When you are ready, compare your answers to those on the next slide

Solutions: Test Your Understanding

1. What are the four values in the `logic` value set?
 - 0,1,X,Z
2. Why do we use `logic` types rather than `bit` in RTL code?
 - `bit` types initialize to 0, which is a known value, whereas `logic` types initialize to X, which is unknown. An unknown initial type value can help detect initialization problems in RTL code.
3. Rewrite the binary value 8'b11010011 in hexadecimal.
 - 8'hd3
4. What is the primary difference between a `wire` net and a `logic` variable for RTL code?
 - `wire` nets are used for multiple-driven signals as in bidirectional and tri-states, whereas `logic` variables can only have a single driver. Also, net types cannot be driven from procedural blocks.

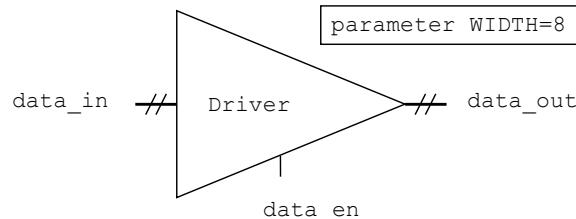


This page does not contain notes.

Lab

Lab 1 Modeling a Data Driver

- To describe and instantiate a parameterized-width bus driver.

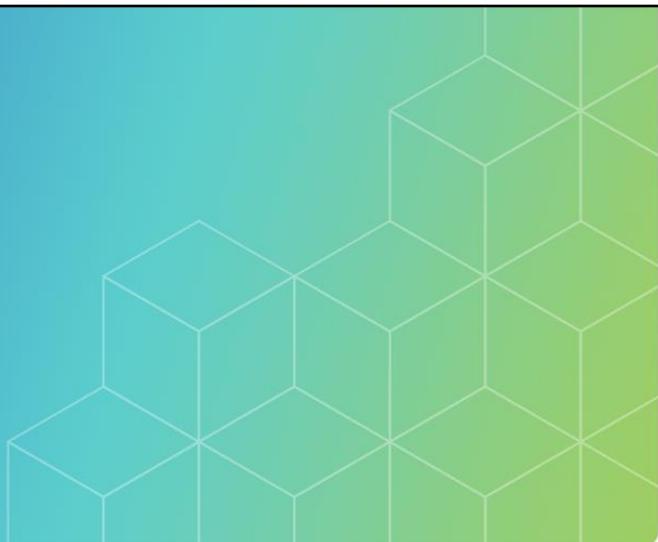


319 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to use basic SystemVerilog constructs to describe a simple design.

For this lab, you use basic SystemVerilog constructs to describe a parameterized-width two-to-one multiplexor. The lab instructions explain how to declare module parameters and vector ranges.



Submodule 4-3

Making Procedural Statements

cadence®

This submodule describes procedural programming statements.

Submodule Objective

In this submodule, you will:

- Use procedural blocks to describe design behavior.

Topics include:

- Procedural blocks review
- Making procedural assignments
- Making conditional statements
- Making case statements
- Making loop statements

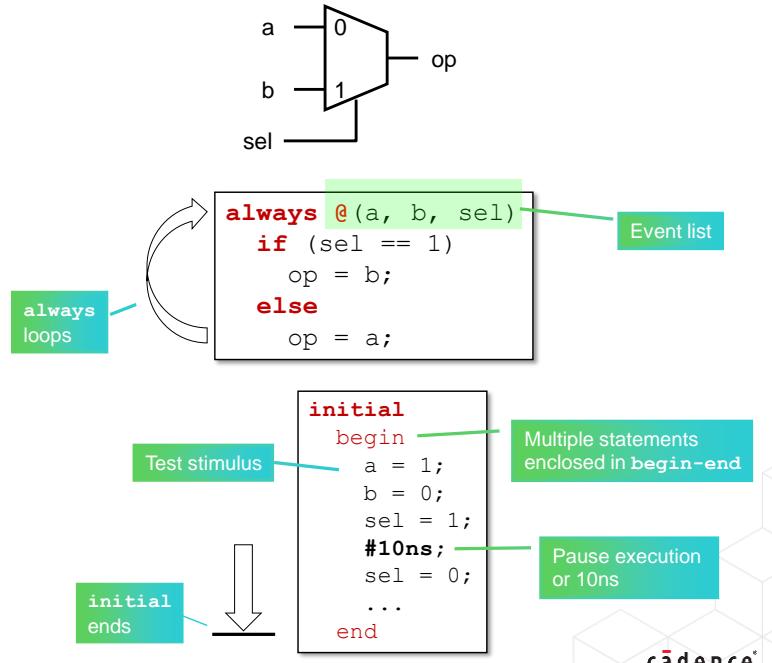


Your objective is to know generally about procedural blocks and procedural statements, and more specifically about the branching procedural statements.

Overview: Procedural Blocks

There are two general procedural blocks used in testbenches.

- always
 - Executes at start of simulation.
 - When at end, loops back to the beginning.
 - Further execution controlled by the event list.
 - Change in value of any variable in list triggers block.
- initial
 - Executes at start of simulation.
 - Embedded timing pauses execution.
 - When at end, terminates.
- Multiple procedural statements need begin...end.
- Procedural blocks cannot be assigned to nets (wire).



322 © Cadence Design Systems, Inc. All rights reserved.

Previous examples described the half adder behavior by making continuous assignment statements.

To describe complex behavior, SystemVerilog provides procedural blocks which contain procedural statements. Procedural statements can define much more complex behavior such as conditional or looped functionality. You must also use procedural blocks to infer registered logic.

Multiple procedural statements within a block must be grouped within begin and end keywords. A single statement within a block does not require begin and end.

SystemVerilog provides two general procedural blocks:

- always starts executing at the start of simulation. Upon executing the last statement, execution loops back to the beginning of the construct. always typically has an event list that controls further execution. Any change in value for a variable in the event list retriggers the always execution. always may also (but rarely) contain embedded timing to suspend and resume execution.
- initial starts executing at the start of simulation. Embedded timing within the block suspends and resumes execution. Upon executing the last statement, the initial terminates. initial is a testbench construct for applying a set number of simulation data.

Within a procedural block, the statements execute sequentially in their order of appearance.

A module can contain any number of procedural blocks and no execution order is implied between them. Multiple procedural blocks triggered by a change in a variable value can execute in any order.

Specialist RTL Procedural Blocks

RTL code only uses special-always blocks.

`always_comb`

- Combinational logic
- Implicit, complete event list
- Uses `=` assignment

`always_ff`

- Registered logic
- Requires an edge-triggered event list
 - Clock and reset signals only
- Uses `<=` assignment

Variables driven by these blocks cannot be driven by anything else.

```
always_comb
  if (sel == 1)
    op = b;
  else
    op = a;
```

All variables read in block automatically added to event list

```
always_ff @ (posedge clock)
  d <= q;
```

Positive-edge triggered

Different assignment (see later)

323 © Cadence Design Systems, Inc. All rights reserved.



SystemVerilog has implementation-specific procedural blocks (`always_comb`, `always_latch`, `always_ff`) for RTL design code.

The always combinational (`always_comb`) procedural block models combinational logic.

- It infers a sensitivity list that includes every variable read by the procedural block. So, it is automatically complete.
- It cannot contain any embedded timing or event control.
- It uses blocking assignment (`=`). More on this later.

The always flip-flop (`always_ff`) procedural block model registered (sequential) logic

- It cannot contain any additional embedded timing or event control.
- It uses nonblocking assignment (`<=`) to infer registered logic. More on this later.

Both blocks prohibits their assigned variables from being driven anywhere else in the design. With these blocks, it is illegal for a variable to be written to by more than one driver, even if these drivers are in different modules or interfaces. This fixes a common issue in the SystemVerilog design.

Making Conditional Statements

```
if (expression) statement
[ { else if (expression) statement } ]
[ else statement ]
```

Condition is boolean expression.

Conditions can overlap.

It checks conditions in order of declaration.

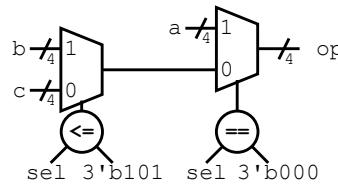
- Executes statement associated with first known true condition.

If there is nothing to do in a branch, a statement can be null.

- Semicolon by itself ;

```
logic [3:0] a, b, c;
logic [2:0] sel;
logic [3:0] op;

always_comb
  if (sel == 3'b000)
    op = a;
  else if (sel <= 3'b101)
    op = b;
  else
    op = c;
```



324 © Cadence Design Systems, Inc. All rights reserved.



The conditional statement is a two-way branch. If the conditional expression evaluates to 1 then the first branch executes and otherwise if the expression is 0 or unknown, the second branch executes (if it exists). The conditional expression can be multiple bits.

In this example, the procedural block is triggered when a transition occurs on at least one of the variables read within the block (a, b, c, or sel). The if statement evaluates the `sel == 3'b000` expression. If the expression is 1, then the first branch executes to assign `op` to `a`. If the expression is 0 or unknown, we evaluate the second if expression `sel <= 3'b101`. If the expression is 1, we execute the branch `op = b`. Otherwise we move to unconditional `else` to assign `op` to `c`.

Note the conditional statements imply priority. If `sel` is `3'b000`, then both the first and second condition expressions are 1, but because we check `sel == 3'b000` first, it is only if this expression is not 1 do we evaluate `sel <= 3'b101`.

Making Case Statements: case

```
case (expression)
  item { , item } : statement(s)
  {item { , item } : statement(s) }
  [ default [:] statement(s) ]
endcase
```

Checks an expression against a series of matches.

- Executes statement(s) associated with first match.
- Multiple statements in a branch enclosed in begin...end.

Case item match expressions can overlap.

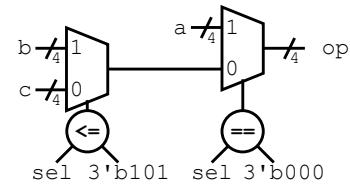
- Compares matches to case expression in the order they appear.
 - Using case equality comparison (==).

Optional `default` item executed if no other matches.

- In this example, inputs containing Z or X.

```
logic [3:0] a, b, c;
logic [2:0] sel;
logic [3:0] op;

always_comb
  case (sel)
    0          : op = a;
    1,2,3,4,5 : op = b;
    6,7        : op = c;
    default    : op = 'b0;
  endcase
```



325 © Cadence Design Systems, Inc. All rights reserved.

cadence®

`case` is a multi-way prioritized branching statement. `case` evaluates the expression in brackets, and then tries to match this value against a series of item expressions in order of their declaration. The first match found executes the branch statements associated with the item match.. High-impedance (z) and unknown (x) values in the case expression will only match x or z values in the item expressions. You can comma-separate multiple item expressions for each branch. If you have multiple statements in a branch, they must be enclosed in begin end statements.

You can optionally provide a single `default` match item to be matched when no other item expressions matches.

In this example, the `case` statement evaluates `sel`. If the value is 0 then it executes the first branch. If the value is between 1 and 5 then it executes the second branch. If the value is between 6 and 7 then it executes the third branch. If the value is anything else, it executes the `default` branch, which sets the output to zero.

Making Case Statements: casez

Case statement with do-not-cares.

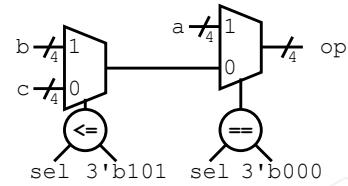
- Treats Z and ? characters as do-not-care bit positions in:
 - Case expression (sel)
 - Case item expression 3'b0??
- Do-not-care bits are not considered in matching
 - E.g., 1'b? will match 1'b0, 1'b1, 1'bZ or 1'bX

```
logic [3:0] a, b, c;
logic [2:0] sel;
logic [3:0] op;

always_comb
  casez (sel)
    3'b000      : op = a;
    3'b0??, 3'b?0? : op = b;
    3'b11Z      : op = c;
    default      : op = 'bx;
  endcase
```

Also casex

- Treats Z, X or ? as do-not-care.
 - Dangerous as treats uninitialized X value in case expression as do-not-care.
 - Can hide initialization issues.
 - Not recommended to use.



326 © Cadence Design Systems, Inc. All rights reserved.



casez is a variation of case that treats z, and ? characters as “don’t-care” bit values in both the case expression and case item selectors.

Don't-care bits are ignored for matching, and so effectively will match any value, including z or x.

There is also a casex variant, but since this treats the uninitialized value x as a don't-care in the case expression, it can hide design initialization problems in RTL code and therefore should be avoided.

Alternate Case Statement Form

Case statement can be inverted.

- Case expression can be a literal.
- Case item can be an expressions.

Example checks enables in order:

- `en_a` has highest priority.
- `en_c` has lowest priority.

```
logic [3:0] a, b, c, op;
logic en_a, en_b, en_c;

always_comb
  case (1'b1)
    en_a    : op = a;
    en_b    : op = b;
    en_c    : op = c;
    default: $display("no enables active");
  endcase
```



We can invert the case statement by declaring the case expression as a literal, and case items as expressions, as in the example here. So, this compares each enable to 1, and executes the branch of the first enable which is 1. Priority is in the order of checking, so `en_a` has the highest priority. If no enables are 1, we execute the `default` branch.

Case Issues

Non-overlapping case items are more efficient.

- Implemented in parallel rather than priority hardware.
- Called *parallel case*.

A fully decoded case expression has a case item branch for every value.

- Called *full case*.
- Avoids several common synthesis issues:
 - Accidental latch inference.
 - Mismatches between RTL and gate level behavior.
- Use `default` to make case full.

Case can optionally be declared as `unique`.

- Runtime warning if the case is not parallel and full.
 - For example, if none or more than one case items are executed.

```
logic [1:0] fullpara;
case (fullpara)
  0:          op = a;
  1,2 :      op = b;
  3:          op = c;
  default: $display("unknown..");
endcase
```

One branch only for
every `fullpara` value

```
logic [1:0] fullpara;
unique case (fullpara)
  0:          op = a;
  1,2 :      op = b;
  3:          op = c;
  default: $display("unknown..");
endcase
```

`unique case`
for runtime checks

More on case synthesis later...

Case is normally implemented in priority logic, as the case item declaration order implies a priority between the branches. If there is no overlap between the case items, that is, no more than 1 case branch is ever true, then the case can be implemented in more efficient parallel logic. A case without branch overlapping is called a parallel case.

A case that has a branch for every possible value in the case expression is called a full case. Full cases avoid several common synthesis issues such as accidental latch inference and mismatches between RTL and gate-level behavior. The simplest way to make a case full is to include a default branch.

In synthesis, a case that is both parallel and full is implemented in the most efficient hardware and avoids common synthesis issues. You should always try to make your case statements parallel and full by design.

SystemVerilog introduces a modifier for case statements with the `unique` keyword. Unique cases will give you a runtime warning if the case execution is not parallel and full, that is, if there is zero or more than 1 case branch to be executed. Unique cases are not infallible. If there is a case expression value that will match more than 1 or zero branches, and you do not apply that value to the case in simulation, then you will not see a runtime error and not detect the issue.

For Loop

```
for ( initialization;
      termination_condition;
      step_expression)
statement(s)
```

For loop has 3 clauses, and is executed as follows:

- Declare and initialize loop variable.
 - Good use for 32-bit 2-state type `int`.
 - Loop variable only visible inside the loop.
- While termination condition true.
 - Executes loop statements.
 - Then executes step expression to modify loop variable.

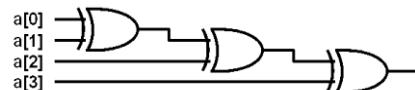
```
logic [3:0] a;
logic parity;

always_com begin
    parity = 1'b0;

    for (int i = 0; i <= 3; i++)
        parity = parity ^ a[i];

end
```

i++ shortcut
for i=i+1



329 © Cadence Design Systems, Inc. All rights reserved.



A `for` loop has three parts and is executed as follows.

The first part declares and initializes the `for` loop variable. As the variable is only used for counting the loop iterations and is not directly synthesized, this is a good use for the 2-state type `int` (32-element bit vector with values 0 and 1 only).

The second clause is the termination condition. The condition is executed, and if it is true, the `for` loop statements are executed. Multiple statements must be enclosed in `begin end`.

The third clause is the step expression; after every loop execution, the step expression is executed (usually to increment or decrement the loop variable), and the terminating condition is checked again.

The loop executes until the termination condition is no longer 1.

This example calculates the parity of vector `a`. The `for` loop initializes the loop variable `i` to 0, and while `i` is less than or equal to 3, it exclusive-ORs the partial result with each successively indexed bit of the input, each time incrementing the index by 1.

Foreach Loop

```
foreach ( array [loop_variable] )
```

foreach iterates over all elements of an array.

- Bounds and direction extracted from declaration.
- More convenient than `for` loop.

foreach loop variable:

- Does not have to be declared.
- Only visible inside loop.

Use multiple loop variables for multi-dimensional arrays.

- Equivalent to nested loops.

```
logic [3:0] a;
logic parity;

always_com begin
    parity = 1'b0;

    foreach (a [i])
        parity = parity ^ a[i];
end
```

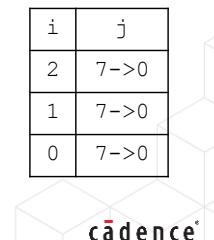
```
for (int i = 3; i >= 0; i--)
    parity = parity ^ a[i];
```

Equivalent for

```
logic [7:0] vecarr [2:0];

always_comb
    foreach (vecarr [i, j])
        vecarr[i][j] = i + j;
```

i	j
2	7->0
1	7->0
0	7->0



The `foreach` loop iterates over all the elements of an array. It is equivalent to a `for` loop, which iterates over the full width of the array from left bound to the right. This construct is useful for simple loop-based array initialization or processing.

The `foreach` loop variable is automatically declared (and typed) as part of the `foreach` statement. As a local variable, it is visible only within the `foreach` statement and any nested blocks.

If the array is multidimensional, you can declare a loop variable for each dimension. This creates nested loops. Loop variables are mapped to dimensions in the order of indexing priority. In the example above, `i` iterates from 2 to 0, and `j` from 7 to 0.

Repeat Loop

`repeat (expression) statement`

Executes for a fixed number of iterations.

- Number can be an expression.

Used where an index is not required.

- For example, not indexing an array.

Example is a shift-multiplier.

```

logic [3:0] a, b;
logic [7:0] result;

logic [7:0] temp_a;
logic [3:0] temp_b;

always_comb begin
    temp_a = a;
    temp_b = b;
    result = 0;

repeat ( 4 ) begin
    if ( temp_b[0] )
        result = result + temp_a;
    temp_a = temp_a << 1;      // left
    temp_b = temp_b >> 1;      // right
end

end

```

331 © Cadence Design Systems, Inc. All rights reserved.



A repeat loop executes its following statements the number of times specified by its parenthesized expression.

This example multiplies two 4-bit inputs to produce an 8-bit output. As it uses a shift multiplier implementation, it does not need a loop index, so uses a repeat loop instead of a for loop.

break and continue

`break` and `continue` are allowed in loops.

- `break`
 - Jumps to the end of the loop.
 - Usually under conditional control.
 - Example left rotates `data` until msb = 1.
- `continue`
 - Jumps to the next iteration of a loop.
 - Usually under conditional control.
 - Example counts number of zeros in `data`.

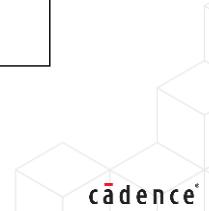
```
repeat (8) begin
    data = {data[6:0], data[7]};
    if (data[7])
        break;
    end
    ...

```



```
foreach (data [i]) begin
    if (data[i])
        continue;
    count = count + 1;
end

```

You can use `break` and `continue` statements only within RTL loop statements (`for`, `foreach`, `repeat`) and verification loop statements (`do`, `while`, `forever`).

`break` terminates the current loop, i.e., it forces a jump to the end of the loop. The `break` is usually conditional. This `break` example rotates `data` to the left until the rightmost bit is 1.

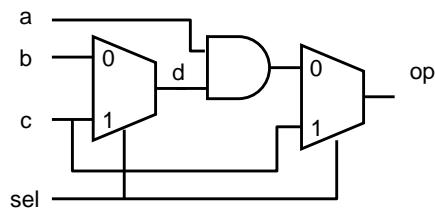
`continue` starts the next iteration of the loop, i.e., it forces a jump to the beginning of the loop. Again, `continue` is usually conditional. This `continued` example counts the number of zeros in the `data` vector. The `foreach` loop iterates through the `data` bits and jumps over the increment statement if the current `data` bit is 1.

Local Declarations in Procedures

Procedures can have local variables.

- Declared inside the procedure.
 - After beginning and before any executable lines.
- Only visible in the procedure where declared.
- Used as temporary variables.
 - Helps simplify expressions.
 - Helps with readability.
 - Access to intermediate values for debugging.

Must be declared before the first executable line.



```
always_comb begin
  logic d;
  d = sel ? c : b;
  op = sel ? c : (a & d);
end
```



Procedural blocks can contain local variable declarations. These must be declared after the procedure begins and before any executable lines. Variables declared in a procedure are only visible inside the procedure where they are declared. They are mainly used for temporary variables inside the procedure, for example, to store common sub-terms or intermediate values in expressions. As such, they can simplify your expressions, making your code easier to read. They can also be added to a waveform window in interactive debug to give better visibility of the procedure execution in debug.

Local declarations must be declared before the first executable line of the procedure. Effectively the compiler has a mode switch. When the compiler enters the procedure, the switch is set to executable mode, and the compiler expects executable statements. If the first statement after the beginning is a declaration, then the compiler switches to declarative mode. When the compiler finds the first executable statement, it switches back to executable mode, and no more local declarations are allowed in the current scope.

In SystemVerilog, any begin...end block (i.e., any local scope) can have local declarations, although, other than procedures, this is very rare.

Test Your Understanding

1. If more than one case item expression matches the case expression, which associated statement(s) execute?
2. What is a full case and what is a parallel case?



Please pause here for a moment to consider these questions. Refer to the module contents as needed. When you are ready, compare your answers to the next slide.

Solutions: Test Your Understanding

1. If more than one case item expression matches the case expression, which associated statement(s) execute?
 - The case expression is compared to the case item expressions in the order they appear, and the statement(s) associated with only the first match is executed.
2. What is a full case and what is a parallel case?
 - A full case is a case statement where there is a case item branch for every value in the case expression
 - A parallel case is a case statement that has no overlapping conditions in the case item expressions. i.e., no more than one case item is true for any value in the case expression.

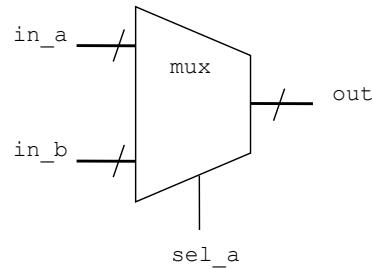


This page does not contain notes.

Lab

Lab 2 Modeling a Simple Multiplexor

- To use SystemVerilog procedural constructs to model a simple multiplexor.

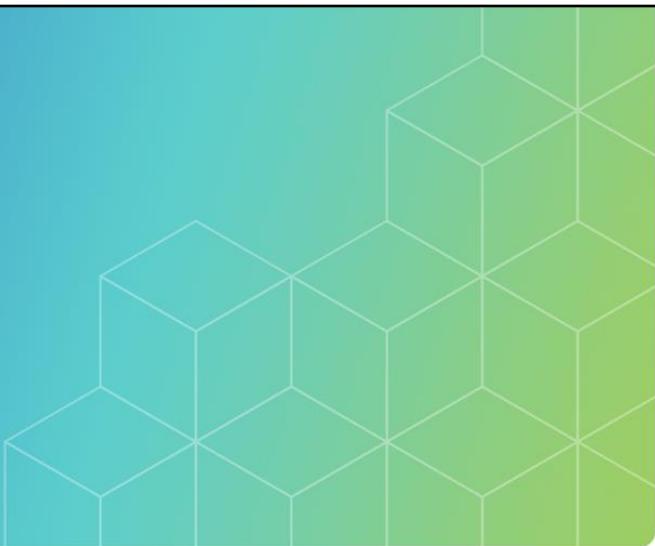


336 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to appropriately and correctly procedurally describe design behavior.

In this lab, you use procedural statements to model a simple multiplexor.



Submodule 4-4

Using Operators

cadence®

This submodule provides an explanation and example for each operator.

Submodule Objective

In this submodule, you will:

- Use SystemVerilog operators.

Topics include:

Category	Symbol(s)
bit-wise	<code>~ & ^ ~^</code>
reduction	<code>& ~& ~ ^ ~^ ~~</code>
arithmetic	<code>** * / % + -</code>
shift	<code><< >> <<< >>></code>
relational	<code>< > <= >=</code>
equality	<code>== != === !==</code>
logical	<code>! && </code>
conditional	<code>? :</code>
concatenation	<code>{ }</code>
replication	<code>{ { } }</code>

338 © Cadence Design Systems, Inc. All rights reserved.



Your objective is to know what operators are available and what they do.

Bit-Wise Operators

not	\sim
and	$\&$
or	$ $
xor	$^$
xnor	$\sim^ \quad ^\sim$

- Bit-wise operators operate on vectors.
- Operations are performed bit-by-bit on individual bits.
- The result is $1'b0$, $1'b1$ or $1'bx$.
- Unknown bits in an operand do not necessarily lead to unknown bits in the result.

```

logic [3:0] veca, vecb, vecc;
logic [3:0] num;

initial begin
    veca = 4'b1001;
    vecb = 4'b1010;
    vecc = 4'b11x0;

    num = ~veca;           // num = 0110
    num = veca & 4'b0111;  // num = 0001
    num = veca & vecb;   // num = 1000
    num = veca | vecb;   // num = 1011
    num = vecb & vecc;   // num = 10x0
    num = vecb | vecc;   // num = 1110
end

```



The bitwise operators perform logical operations in a bitwise manner.

The bitwise unary negation operator inverts the logical sense of each bit of its operand, i.e., each 0 becomes 1, and each 1 becomes 0, and each high-impedance bit becomes unknown.

The bitwise binary operators first zero-extend a smaller operand to match the width of a larger operand and then perform logical operations on individual bit positions. Depending upon the operation, bits in one operand that are 0 or 1 can mask bits in the same position of the other operand that is high-impedance or unknown, so unknown bits in an operand do not necessarily produce unknown bits in the result.

In the example shown here, last but one assignment to number shows bitwise, and operation between vector b and vector C. Since 0th and 2nd bits of vector b are zero, irrespective of other counterpart values, those bits of the number variable are always zero.

Unary Reduction Operators

and	&
or	
xor	^
nand	~&
nor	~
xnor	~^ ~~

- Reduction operators perform a bit-wise operation on all the bits of a single operand.
- The result is always 1'b0, 1'b1 or 1'bx.

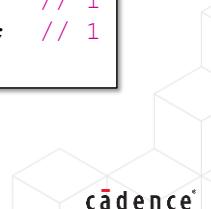
```

localparam CONST_A = 4'b0100,
          CONST_B = 4'b1111;

logic val;

initial begin
    val = &CONST_A; // 0
    val = |CONST_A; // 1
    val = &CONST_B; // 1
    val = |CONST_B; // 1
    val = ^CONST_A; // 1
    val = ^CONST_B; // 0
    val = ~|CONST_A; // 0
    val = ~&CONST_A; // 1
    val = ^CONST_A && &CONST_B; // 1
end

```



Unary reduction operators operate on all bits of a single operand to produce a single-bit result. The effect is as if they first applied the logical operation to the first two bits of the operand, then iteratively applied the logical operation to the current partial result and the next bit. The result of the operation is always a single bit that is 0, 1, or unknown (x).

You will see these same operators also used as bitwise binary operators. When used with a single operand, they are reduction operators.

Arithmetic Operators

add	+
subtract	-
multiply	*
divide	/
modulus	%

```

localparam CONST_INT = -3,
        CONST_5    = 5;

logic [3:0] veca, vecb, vecc, num;

integer val;

initial begin
    veca = 3;
    vecb = 4'b1010;
    vecc = 14;
    val = CONST_5 * CONST_INT; // -15
    val = (CONST_INT + 5)/2; // 1
    val = CONST_5/CONST_INT; // -1
    num = veca + vecb; // 1101
    num = veca + 1; // 0100
    num = CONST_INT; // 1101
    num = vecc % veca; // 0010
end

```

In the examples, note that division of the integer division discards the fractional part. See where assigning 3 to an unsigned 4-bit vector logic keeps the same rightmost four bits but now interprets the value as +13.

Some additional “tidbits” about arithmetic operators:

- Any z or x bit in either operand produces an unknown (x) result.
- Integer division discards any remaining fractional part.
- Division or modulo by 0 produces an unknown (x) result.
- Raising 0 to a negative power produces an unspecified result.
- Raising a negative value to a real power produces an unspecified result.

Shift Operators

logical shift << >>

- Ignores operand signs.
- Fills extra bits with 0.
- Implements division or multiplication by powers of two.

arithmetic shift <<< >>>

- Ignores the right operand sign.
- Left shift operates like logical left shift.
- Right shift preserves the left operand sign if the result is a signed expression.

```
logic      [7:0] veca, vecb;
logic signed [7:0] vecsign;

initial begin
    veca     = 8'b10011001;
    vecsign = 8'b10011001;

    vecb = veca << 1;      // 00110010
    vecb = veca >> 1;      // 01001100
    vecb = vecsign <<< 1;   // 00110010
    vecb = vecsign >>> 1;  // 11001100
    vecb = veca << -1;     // 00000000
end
```

Negative integer implemented in 2s complement
but treated as binary for number of shifts: -1 is $2^{32}-1$

342 © Cadence Design Systems, Inc. All rights reserved.



The logical shift operators shift the left operand by the number of bit positions given by the right operand interpreted as a positive number, filling vacated bit positions with 0. You can use the logical shift operators to implement integer division or multiplication by powers of 2.

The arithmetic left shift operator operates exactly as does the logical left shift operator. The arithmetic right shift operator preserves the sign bit if the resulting expression is signed.

The example initializes `veca` to 153 (8'h99). Left-shifting this value once is equivalent to doubling its value, and placing the result back into an 8-bit vector truncates the value back down to 50 (8'h32). Right-shifting this value once is equivalent to halving its value and losing the fractional part, so it produces 76 (8'h4C). Left-shifting by -1 is equivalent to left-shifting a very large number of times, so it produces 0.

Relational Operators

less than	<
greater than	>
less than or equal to	<=
greater than or equal to	>=

The result is:

- 1'b0 if the relation is false.
- 1'b1 if the relation is true.
- 1'bX if either operand contains any z or x bits.

```
logic [3:0] veca, vecb, vecc;
logic val;

initial begin
    veca = 4'b0011;
    vecb = 4'b1010;
    vecc = 4'b0x10;

    val = vecc > veca ; // val = X
    val = vecb < veca ; // val = 0
    val = vecb >= veca; // val = 1
    val = vecb > vecc ; // val = X
end
```

343 © Cadence Design Systems, Inc. All rights reserved.



If at least one operand is unsigned, the comparison treats both operands as unsigned and zero-extends a smaller operand to match the width of a wider operand.

If both operands are signed integer quantities, the operation sign-extends a smaller operand to match the width of a wider operand and treats the comparison as a signed integer comparison.

If at least one operand is real, the operation, if necessary, converts the other operand to real and treats the comparison as a real comparison.

The result is 0 if the relation is false and 1 if the relation is true. Any high-impedance (z) or unknown (x) bit in either operand produces an unknown result regardless of the truth of the relation.

In code snippet, any operation involving vector C results into unknown, since vector C has unknown bits.

Logical Equality and Case Equality Operators

logical equality ==

- Unknown bits give an unknown result.

	0	1	z	x
0	1	0	x	x
1	0	1	x	x
z	x	x	x	x
x	x	x	x	x

- Also logical inequality !=

```
...
a = 2'b1x;
b = 2'b1x;

if (a == b)
    // values match & do not contain Z or X
else
    // values do not match or contain Z or X
```

else branch executed

case equality ===

- Result is always known.

	0	1	z	x
0	1	0	0	0
1	0	1	0	0
z	0	0	1	0
x	0	0	0	1

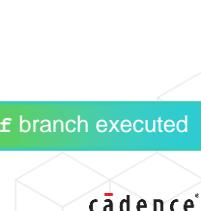
- Also case inequality !==

```
...
a = 2'b1x;
b = 2'b1x;

if (a === b)
    // values match exactly
else
    // values do not match
```

if branch executed

344 © Cadence Design Systems, Inc. All rights reserved.



The difference between logical equality and case equality is the handling of high-impedance and unknown values. The logical equality treats high-impedance and unknown bits as truly unknown bit values, while the case equality treats high-impedance and unknown bits as values to be matched. A case equality operation thus always produces a 0 or 1 result and never an unknown result.

The case equality operator gets its name from the fact that the `case` statement matches items in the same manner. Some people informally call it the “identity” operator because it checks that the bits are identical instead of that their values are the same.

In the given code snippet, the difference is clear in the result when logical equality and case equality operators are used on the same operands.

Wildcard Equivalency Operator

Checking selected *bits* from a large vector may require complex if expressions.

Wildcard equality allows bits to be defined as do-not-care.

- Like `casez`

An X, Z or ? in the **right-hand** operand matches any value in the left.

- Asymmetric – only right side can have wildcard bits.

Also wildcard inequality (`!=?`).

```
logic [7:0] status = 8'b11001001;
if ((&status[7:6]) & status[3] & status[0])
    $display("pass");

if (status ==? 8'b11??1??1)
    $display("pass");

if (status ==? 8'b110???0?)
    $display("pass");
```



The wildcard equality operator (`==?`) and the wildcard inequality operator (`!=?`) treat unknown (X) and high-impedance (Z) values in the given bit position of the **right operand** as “don’t-care” bits to ignore when making the comparison. When the right operand is a literal, you can also use ? as a wildcard. Using a ? is more readable than X or Z. These operators return a 1-bit result that can be 0, 1, or unknown.

These operators are useful to check selected bits or bit ranges from a large vector. Normally this may require complex if conditional expressions. With a wildcard equality operator, the check is much simpler and more readable.

In the example, the status variable is compared with selected bits of an expression by using the wildcard equivalency operator.

Set Membership Operator

Checking a vector against a range of *values* may require complex if expressions.

`inside` does a comparison:

- True if expression value is contained within a value list.
- List values can be variables (including arrays), literals, or ranges.
- List values can overlap.

Uses the wildcard equality operator (`==?`) for checking values in the list.

- Result is `1'b0`, `1'b1`, `1'bX`.
- Allows list values to have wildcard bits.

```
if ((a==2'b01) | (a==2'b10))
...
if ( a inside {2'b01, 2'b10} )
...
Instead of this...
... write this
```

```
logic [7:0] lookup [1:0];
...
if ( a inside {lookup, [0:2]} )
...
// equivalent to:
if ( a inside {lookup[1], lookup[0], 0, 1, 2})
```

```
if ( a inside {2'b0?} )
...
// equivalent to
if ( a inside {2'b00, 2'b01, 2'b0X, 2'b0Z})
```



The set membership (`inside`) operator performs a comparison between a left-side expression and a right-side list of values. The right-side list can utilize wildcards, value ranges, and the values can overlap.

Range values must be expressed in the form `[lower_bound : higher_bound]`. Either bound can be `$`, indicating the minimum or maximum value for the expression.

For integral expressions, SystemVerilog performs a wild equality match operation, which is an asymmetric match. Therefore, the result can be `0`, `1`, or unknown.

In the example, instead of checking, if, '`a`' equals 1 or 2, we can just use '`inside`' operator and put those values on the right-side for a match. Second example uses as array to perform the check and last example uses a wild card and makes the check easier with '`inside`' operator.

Logical Operators

Reduce each operand to a single bit...

- Using OR reduction

...then perform a single bit operation.

not	!
and	&&
or	

Operands are reduced to either true (`1'b0`) or false (`1'b1`) or unknown (`1'bX`).

0 – if all bits 0

1 – if any bit 1

X – if any bit is Z or X and no bit is 1

```
localparam FIVE = 5;
localparam CONST_A = 4'b0011,
CONST_B = 4'b10xz,
CONST_C = 4'b0z0x;

logic ans;

initial begin
ans = !CONST_A; // 0
ans = CONST_A && 0; // 0
ans = CONST_A || 0; // 1
ans = CONST_A && FIVE; // 1
ans = CONST_A && CONST_B; // 1
ans = CONST_C || 0; // X
end
```

347 © Cadence Design Systems, Inc. All rights reserved.



Logical operators reduce each operand to a single bit and then perform a single-bit operation.

The rules for reduction of an operand are:

- If the operand contains all zeroes, it reduces to logic 0;
- If the operand contains any ones, it reduces to logic 1; and
- If the operand contains no ones but does contain one or more high-impedance or unknown values, it reduces to an unknown, as its logical value is unknown.

The unary logical negation operator then inverts the logical sense of its operand, i.e., 0 becomes 1 and 1 becomes 0.

The binary conjunction and disjunction operators produce the logical conjunction and disjunction, respectively, of their operands.

Conditional Operator

Short form of the `if` statement for simple conditions.

Syntax

```
result = <condition> ? <true value> : <false value>;
```

```
logic [3:0] a, b, op;
logic sel;

assign op2 = sel ? a : b;
```

Operator use in `assign`

```
logic [3:0] a, b, op;
logic sel;

always_comb
  op3 = sel ? a : b;
```

Operator use in procedural block

```
logic [3:0] a, b, op;
logic sel;

always_comb
  if (sel == 1)
    op = a;
  else
    op = b;
```

Equivalent `if` statement

348 © Cadence Design Systems, Inc. All rights reserved.



The conditional operator is a *ternary* operator, i.e., it has three operands.

- If the first operand, i.e., the condition expression, is 1 then the operation result is the value of the second operand;
- If the first operand is 0 then the operation result is the value of the third operand; and
- If the first operand is unknown (x), then the operation result is the value of the second operand merged with the value of the third operand such that if any bit position has the same value in both operands, then that bit position of the result also has that value.

So, the conditional operator selects between two operands like a 2-to-1 multiplexor.

The operands can be any expression. It is very common to nest conditional operations.

First example shows usage of conditional operator in an assign statement. If select is true, 'a' is continuously assigned to output else 'b' is assigned.

Second example shows usage of conditional operator in always combinational block. If there is any change in select, or a, or b, it triggers the block and makes the assignment to output. Third example is equivalent of second example using (if) condition.

Concatenation Operator

concatenation { }

- Can select and join bits from different vectors to form a new vector.
 - Forms unsigned expression.
- Can reorganize vector bits to form a new vector.
 - Endian swaps / reverse / rotate
- Can use on either side of an assignment!

Literals in a concatenation must be explicitly sized.

- To calculate bit positions of result.

```

logic [7:0] veca, vecb, vecc, vecd, new;
logic [3:0] nib1, nib2;

initial begin
  veca = 8'b00000011;
  vecb = 8'b00000100;
  vecc = 8'b00011000;
  vecd = 8'b11100000;

  new = {vecd[6:5], vecc[4:3], vecb[3:0]};
  // new = 8'b11_11_0100

  new = {2'b11, vecb[7:5], veca[4:3], 1'b1};
  // new = 8'b11_000_00_1

  new = {vecd[4:0], vecd[7:5]};
  // rotate vecd right 3 places
  // new = 8'b00000_111

  {nib1, nib2} = veca;
  // nib1 = 4'0000, nib2 = 4'0011
end

```

349 © Cadence Design Systems, Inc. All rights reserved.



A concatenation operation joins together the bits of one or more comma-separated operands. You must size literal constant operands so that the operation can determine exactly where to place each bit.

In the code example, the new variable is declared as an 8-bit operand. Then, while updating its value, in the first concatenation operation, 3 down to 0 bits of 'new' variable are replaced with 3 down to 0 bits of vector b, and 5 down to 4 bits of 'new' variable are replaced with 4 down to 3 bits of vector c, and 7 down to 6 bits of 'new' variable are replaced with 6 down to 5 bits of vector d.

Here are some examples that fail to size their operands:

```

a[7:0] = {5'b01010, 2}; //decimal value 2 unsized
c[3:0] = {3'b011, 'b0}; //binary value 'b0 unsized

```

Replication Operator

replication { { } }

- Reproduces a concatenation a set number of times.
- Syntax
`{replicate{sized_expr}}`
- replicate must be:
 - A constant number or expression.
 - Without any Z or X values.

```

logic      veca = 1'b1;
logic [1:0] vecb = 2'b11;
logic [3:0] vecc = 4'b1001;
logic [7:0] bus;

initial begin
    // single bit veca replicated 8 times
    bus = {8{veca}}; // bus = 11111111

    // 4x veca concatenated with 2x vecc[1:0]
    bus = { {4{veca}}, {2{vecc[1:0]}} }; // bus = 1111_01_01

    // vecc concatenated with 2x vecb
    bus = { vecc, {2{vecb}} };           // bus = 1001_11_11

    // vecc concatenated with 2x 1'b1
    // and replicated 2 times
    bus = { 2{vecc[2:1]}, {2{1'b1}} }; // bus = 00_1_1_00_1_1
end

```

350 © Cadence Design Systems, Inc. All rights reserved.



A replication operation replicates a concatenation a non-negative constant number of times. A replication operation applies only to a concatenation – you will not see it in any other context. The replication count cannot have any high-impedance (z) or unknown (x) values.

This example:

- 1st – replicates the value of the single-bit veca variable value eight times to form an 8-bit value it assigns to bus;
- 2nd – concatenates four replications of the value of the single-bit veca variable with two replications of the lowest two bits of the vecc variable to form an 8-bit value it assigns to bus;
- 3rd – concatenates the value of the four-bit vecc variable with two replications of the value of the two-bit vecb variable to form an 8-bit value it assigns to bus; and
- 4th – concatenates the value of the middle two bits of vecc with two replications of the sized literal constant 1'b1, and replicates that 4-bit concatenation twice to form an 8-bit value it assigns to bus.

Declaring Multi-Dimensional Arrays

2

1

Indexing order

SystemVerilog allows arrays of any type:

- Including events, structures, and enumerates.

Allows arrays with any number of dimensions

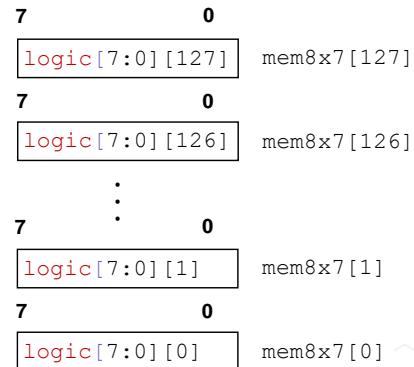
```
logic [7:0] mem8x7 [127:0];  
  
// mem8x7          = array of bytes  
// mem8x7[0]        = 0th byte  
// mem8x7[0][0]     = 0th bit of 0th byte
```

Use in RTL:

- Modeling memories.
- Declaring register arrays.

Therefore, typical RTL arrays are 2D.

- Address index declared after name.
- Data elements stored separately.
- Indexing priority is address first, then data.



351 © Cadence Design Systems, Inc. All rights reserved.

As with programming languages, SystemVerilog supports arrays of any type and any number of dimensions.

In RTL, we typically use arrays to model memories or arrays of registers or to define images for image processing applications. Therefore, most RTL arrays are 2-dimensional. They have a data width range which is defined before the array name, and an address width range which is defined after the array name. Technically this is called an unpacked array in SystemVerilog. The individual data elements are stored in the simulator as separate signals. You can index the 2D array to extract individual data elements or data bits. The "address" ranges to the right of the array name have the highest priority, followed by the "data" ranges to the left. If you have multiple ranges in either position, indexing priority is always left to right.

In the example here: a memory is declared having 128 memory locations and size of each location is 8 bits. When just mem8x7 is mentioned, entire array is selected. If we use memory of 0, it selects 0th byte in the memory. If we say memory of 0 of 0, it selects 0th bit of 0th byte of the memory.

Assignment Patterns

Define a list of values for an assignment to:

- Array elements.
- Structure fields (see later).

It is equivalent to individual assignments:

- Un-sized literals are allowed.

Pattern “keys” can be used:

- Array element index.
- default keyword.

There must be a pattern value for every array element:

- Either explicitly or using default.

```
logic[7:0] regarr [3:0];
logic[7:0] mem8x7 [127:0];

regarr = '{0,1,2,3};
/* equivalent to
regarr[3] = 0;
regarr[2] = 1;
regarr[1] = 2;
regarr[0] = 3; */

regarr = '{3:1, default:0};
/* equivalent to
regarr[3] = 1;
regarr[2] = 0;
regarr[1] = 0;
regarr[0] = 0; */

// initialize mem to 0
mem8x7 = '{default:0};
```

352 © Cadence Design Systems, Inc. All rights reserved.



You can make pattern assignments to unpacked arrays and also unpacked structures. (we will see more about structures in the User-Defined Data Types module).

Assignment patterns are aggregate assignments equivalent to an individual assignment to each array element. It is a compilation error if the number of values in the pattern is more than or fewer than the number of elements in the target array.

The assignment pattern starts with an acute accent ' , as opposed to a compilation directive, which begins with a grave accent ` .

The assignment pattern syntax is very similar to a concatenation, but unlike concatenation, individual values do not need to be sized as the construct expands to individual assignments, SystemVerilog pads or truncates each assignment as needed.

You can make pattern assignments by position or by key, but not by both in the same assignment. For arrays, the key can be an element index or the default keyword, which assigns the default value to all elements not otherwise specified in the pattern.

Use of the default key is the only situation in which it is allowed to have fewer values in the pattern than elements in the array.

In the example here, in the first assignment, the register array is assigned with values 0, 1, 2, and 3 with an assignment pattern. The next assignment shows the usage of 'default'. Here, except 3rd element, the remaining elements are assigned with the value of 0.

Test Your Understanding

1. Explain the difference between `avec & bvec` and `avec && bvec`.
2. Rewrite the following if statement using:
 - a. The inside operator.
 - b. The wildcard equivalency operator.

```
logic [5:0] avec;  
...  
if (avec[5] | avec[3] | avec[1] | avec[0])  
...
```



Please pause here for a moment to consider these questions. Refer to the module contents as needed. When you are ready, compare your answers to those on the next slide.

Solutions: Test Your Understanding

1. Explain the difference between `avec & bvec` and `avec && bvec`.
 - `&` is a bit-wise AND of `avec` and `bvec`, producing a result which is the same size as the two operands.
 - `&&` is an OR reduction of both `avec` and `bvec`, followed by a single AND of the reduced operands.
2. Rewrite the following if statement using:
 - a. The `inside` operator.
 - b. The wildcard equivalency operator.

```
logic [5:0] avec;
...
if (avec[5] | avec[3] | avec[1] | avec[0])
...
//           avec[5]  avec[3]  avec[1:0]
if (avec inside {[63:32], [15:8], [3:1]} )
...
if (avec ==? (6'b1?????) | (6'b??1???) | (6'b????1?) | (6'b?????1) )
...
```

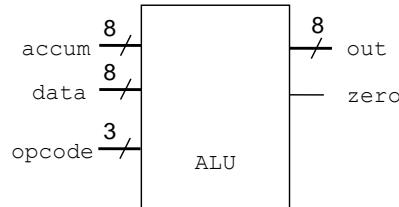


This page does not contain notes.

Lab

Lab 3 Modeling an Arithmetic Logic Unit (ALU)

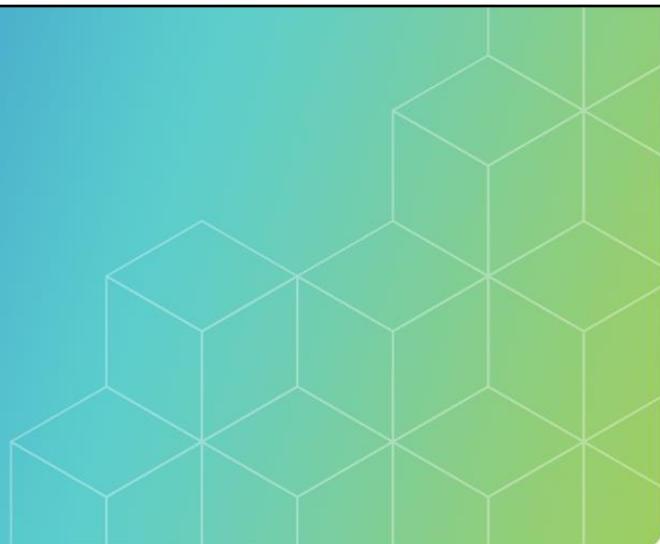
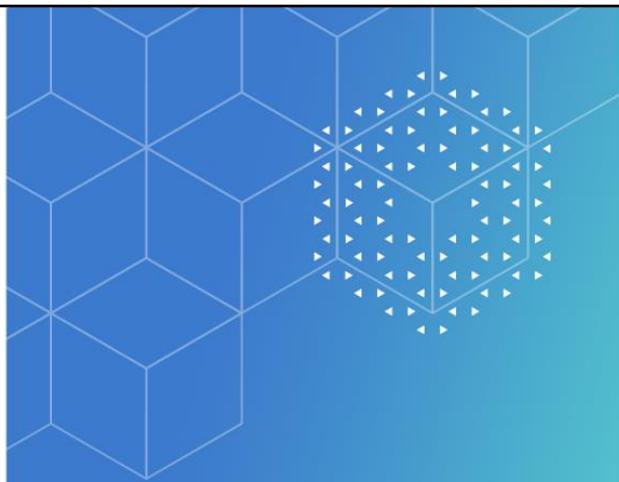
- To use procedural constructs and operators to model an ALU.



355 © Cadence Design Systems, Inc. All rights reserved.



For this lab, you use operators while describing a parameterized-width arithmetic/logic unit (ALU).



Submodule 4-5

Using Blocking and Nonblocking Assignments

cadence®

This submodule reviews blocking and nonblocking assignments, and then examines how to use them in procedural blocks meant to represent combinational logic or sequential logic, and lastly explores issues with statement order and mixing blocking and nonblocking assignments in the same procedural block.

Submodule Objective

In this submodule, you will:

- Use nonblocking assignments to model sequential design behavior.

Topics include:

- Blocking assignment introduction
- Blocking assignments in sequential procedures
- Nonblocking assignment introduction
- Nonblocking assignments in sequential procedures
- Assignments in combinational procedures
- Mixing blocking and nonblocking assignment

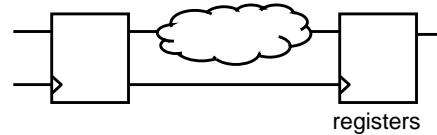


Your objective is to correctly choose between blocking and nonblocking assignments. To do that, you need to know more details about using blocking and nonblocking assignments in combinational and sequential procedures.

Reminder: RTL Partition into Combinational and Sequential Logic

Remember:

- Multiple procedural blocks execute concurrently.
- Execution order is indeterminate.
- = assignment is immediate.
 - Called blocking assignment.
- May execute the block several times before the circuit reaches the steady-state at a given point in time.



We know that multiple procedure blocks execute concurrently, and there is no determined order. we have seen the usage blocking assignment while modeling combinational logic. Blocking assignment assigns the value immediately. RTL blocks generally have both combinational and sequential blocks. We shall see more about these in upcoming slides.

Combinational Logic and = (Blocking Assignment)

Combinational logic uses =

Procedure execution order is not important.

- Same result if P1 or P2 executed first...
- ...although extra execution of P2 if it is first.
- Procedures may execute several times to reach a steady-state at a given time.

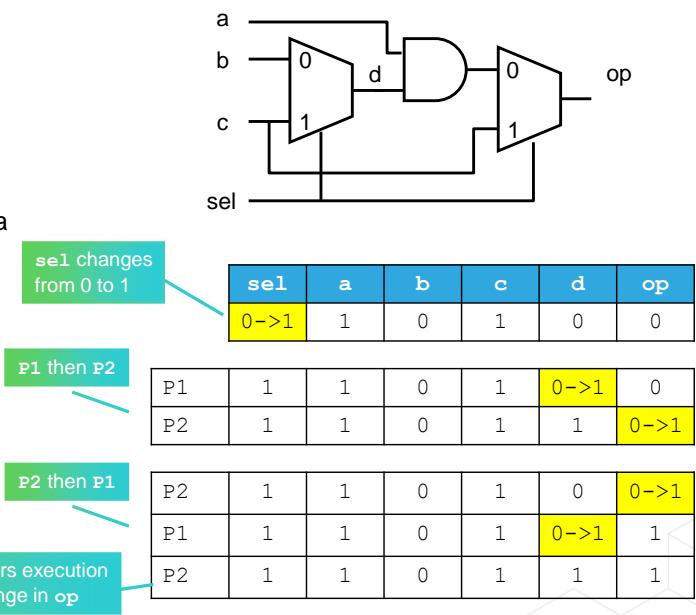
Both P1 and P2 triggered

```

P1 {
    always_comb
    d = sel ? c : b;
}

P2 {
    always_comb
    op = sel ? c : (a & d);
}

```



359 © Cadence Design Systems, Inc. All rights reserved.



The assignment we have used so far uses the symbol =. This is called blocking assignment. Blocking assignment is instantaneous. When the assignment is executed, the variable is updated. Blocking assignment is used in combinational logic.

The example uses two combinational procedures, P1 and P2, to define the required logic.

If the input `sel` changes from 0 to 1, then this triggers both procedures, as they both read `sel` on the right-hand side of an assignment. Both procedures are triggered at the same time, but we cannot determine the execution order of the procedures. The language definition explicitly states the execution order is unknown. However, with blocking assignment in combinational logic, the execution order is not important and gives the same result regardless of whether P1 or P2 is executed first.

- Order P1 then P2: P1 updates `d` to 1. P2 updates `op` to 1. Results are as expected.
- Order P2 then P1: P2 updates `op` to 1. P1 updates `d` to 1. Then as `d` has changed value and is read by P2, this re-triggers P2, although there is no change in output. Results are as expected.

The extra execution of P2 in the second case may look inefficient, but this is very common. It may take several executions of a procedure for the design to reach a steady-state for a change in a given input.

Example: Sequential Logic

On every rising edge `clock`:

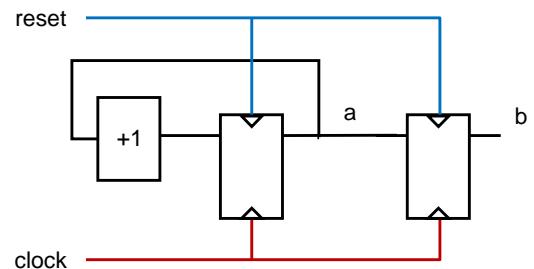
- `a` incremented.
- `b` assigned from `a`.

Synchronous rising edge `reset`.

What happens if we use blocking assignment?

```
P1
  always_ff @(posedge clock)
    if (reset)
      a = 1'b0;
    else
      a = a + 1;

P2
  always_ff @(posedge clock)
    if (reset)
      b = 1'b0;
    else
      b = a;
```



Registered Logic

clock	a	b
(reset)	0	0
1	1	0
2	2	1

Expected Results



360 © Cadence Design Systems, Inc. All rights reserved.

If we look at sequential logic triggered by an edge of a clock signal, the design has two registers. On a rising edge of `clock`, `a` is incremented, and the current (un-incremented) value of `a` is clocked out at `b`. So, for the first `clock` after a reset, we expect `a=1` and `b=0`, followed on the next clock edge by `a=2` and `b=1`, etc.

What happens if we use blocking assignment to model this?

Sequential Logic Cannot Use Blocking Assignment (=)

A rising edge clock triggers both P1 and P2.

Blocking assignments to a and b are immediate.

- But b depends on which procedure executes first.

Blocking assignments can lead to race conditions.

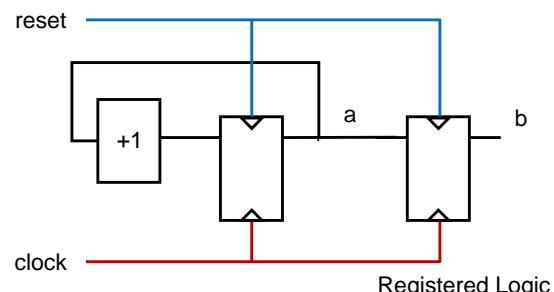
- Specifically, when the clock edge triggers multiple procedures.

```

P1 {
    always_ff @ (posedge clock)
        if (reset)
            a = 1'b0;
        else
            a = a + 1;

    always_ff @ (posedge clock)
        if (reset)
            b = 1'b0;
        else
            b = a;
}

```



P1->P2

P2->P1

clock	a	b
(reset)	0	0
1	1	1
2	2	2

Wrong Results

clock	a	b
(reset)	0	0
1	1	0
2	2	1

Matches Expected Results

361 © Cadence Design Systems, Inc. All rights reserved.



Blocking assignments can lead to race conditions, specifically when the same event triggers multiple procedures that then execute in a nondeterministic order, such that a procedure can read a variable that another procedure may or may not have already written.

In this example, both procedures are triggered on the positive clock edge. The simulator can execute P1 and P2 in any order. The assignment statements use the blocking operator, so a race exists between the assignments to a and to b.

- Order P1 then P2: P1 updates a to get the incremented value, and then P2 updates b to get the new a value. Results are **not** as expected.
- Order P2 then P1: P2 updates b to get the current a value and then P1 updates a to get the incremented value. Results are as expected.

The final b value depends upon the procedure execution order.

Sequential Logic Must Use Nonblocking Assignment ($<=$)

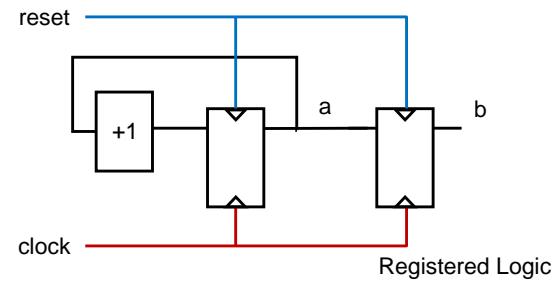
Nonblocking assignment ($<=$) is *scheduled*.

- Value/expression sampled.
- Assignment updated after all procedures are executed.

How does this work?

- Rising edge of clock triggers P1 and P2.
- Executed in any order.
- P1 samples a, increments, and schedules assignment to a.
- P2 samples a and schedules assignment to b.
- After both P1 and P2 finish, scheduled assignments are made.

How is this implemented?



```

P1   always_ff @ (posedge clock)
      if (reset)
          a <= 1'b0;
      else
          a <= a + 1;

P2   always_ff @ (posedge clock)
      if (reset)
          b <= 1'b0;
      else
          b <= a;
  
```



362 © Cadence Design Systems, Inc. All rights reserved.

We use a different form of assignment for sequential logic, using the symbol $<=$ which is called nonblocking assignment. Nonblocking assignment is scheduled. When the assignment is executed, the value or expression on the right-hand side is evaluated, but the assignment does not immediately take place. The assignment will only be made after all the procedures triggered at the current point in simulation time have completed execution.

In this example, both procedures are triggered on the positive clock edge. The simulator can execute P1 and P2 in any order.

When P1 is executed, it increments the current value of a and schedules the assignment to a by placing it on an update queue.

When P2 is executed, it samples the current value of a and schedules the assignment to b by placing it on an update queue.

After both P1 and P2 have completed execution, the scheduled assignments on the update queue are executed.

Scheduled Assignment Implementation

SystemVerilog defines how simulation works.

- Using a stratified event queue.

This is a heavily simplified illustration.

- There are many more regions.

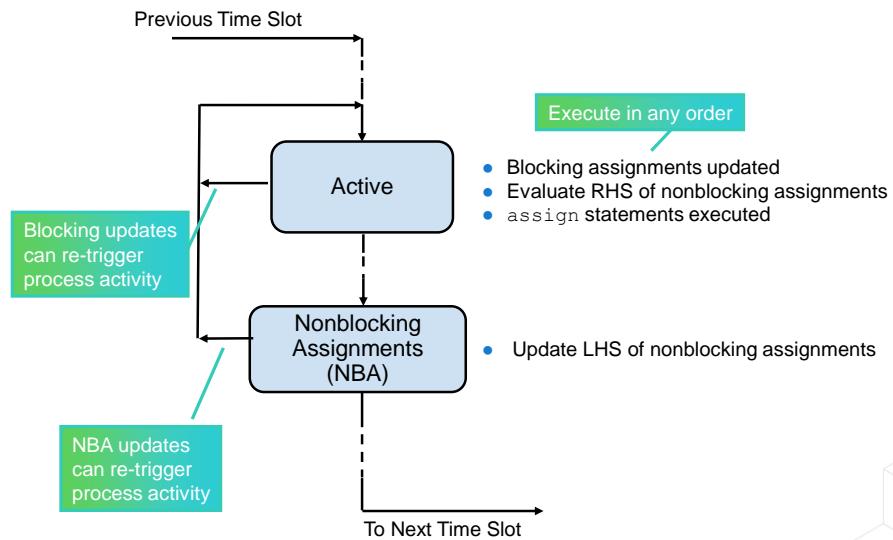
We will just consider 2 regions:

- Active

- Procedures executed.
- Blocking assignments complete.
- assign statements complete.
- Nonblocking assignments evaluated.

- NBA

- Nonblocking assignments complete.



363 © Cadence Design Systems, Inc. All rights reserved.



A simulation timeslot is divided into ordered regions to provide predictable simulation results.

The SystemVerilog event schedule has many regions for each simulation time, but we only need to consider two:

- The Active region is for executing procedures, completing blocking assignments, executing and completing assign statements and evaluating nonblocking assignments.
- The NBA region is our update queue for storing, and after the Active region is complete, executing nonblocking assignments.

These regions are iterative. The NBA can schedule events that require a return to the Active region, which in turn can make further nonblocking assignments for execution in the NBA region. When no more events exist for the current simulation time, the simulator advances simulation time to the next time slot for which events are scheduled. The simulation terminates when no such future events exist.

Nonblocking Assignment Implementation

Scheduled assignment gives us expected results.

Block execution order is no longer important.

- Same result if P1 or P2 is executed first.

```

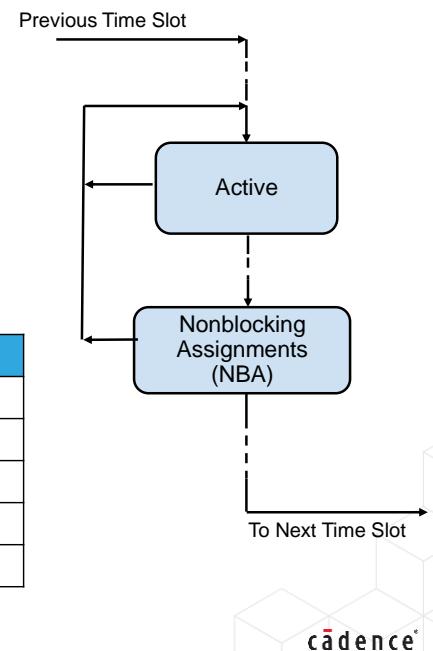
P1 {
    always_ff @ (posedge clock)
        if (reset)
            a <= 1'b0;
        else
            a <= a + 1;

    always_ff @ (posedge clock)
        if (reset)
            b <= 1'b0;
        else
            b <= a;
}

```

		a	b
clock			
(reset)	Active	0	0
	NBA	1	0
	Active	a+1	a
	NBA	2	1

Matches Expected Results



364 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Nonblocking assignment using the event scheduler gives us correct behavior for sequential logic, regardless of the execution order of procedures.

For example, when both P1 and P2 are triggered by a rising edge of the clock, it does not matter in which order they are executed.

a is sampled in the active region, but the assignment is scheduled for the NBA region. b is also sampled in the active region, using the current value of a, and the assignment scheduled for the NBA region. After both procedures have been executed, the simulator moves to the NBA regions and makes assignments to both a and b simultaneously using the sampled values. Now, regardless of the execution order of P1 and P2, the results are as expected.

After we have executed the NBA region, we can move back to the Active region to execute any procedures triggered by changes in value for a and b.

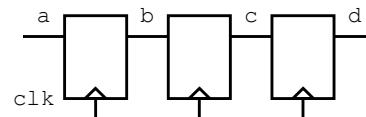
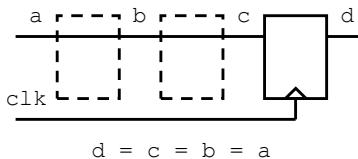
Multiple Blocking Assignments in Sequential Procedures

```
always_ff @(posedge clk)
begin
    b = a;
    c = b;
    d = c;
end
```

b and c are temporary variables

```
always_ff @(posedge clk)
begin
    d = c;
    c = b;
    b = a;
end
```

c from last block execution,
i.e., last clock cycle



Another issue with blocking assignment in sequential logic:

- Implementation is dependent on assignment order.

365 © Cadence Design Systems, Inc. All rights reserved.



There are other issues in using blocking assignment for sequential logic.

The left example uses blocking assignments:

- It assigns a to b and b is immediately updated.
- Then it assigns b to c and c is immediately updated
- Finally, it assigns c to d and d is immediately updated

As each variable is written before it is read, all the variables have the value of a. The b and c variables will be optimized away by synthesis, as the d variable provides the same value.

The right illustration also uses blocking assignments, but it reorders the assignments:

- It assigns c to d and b is immediately updated.
- Then it assigns b to c and b is immediately updated.
- Finally, it assigns a to b and b is immediately updated.

As the variables are read before they are written, all can have different values. All the variables will exist in a hardware implementation.

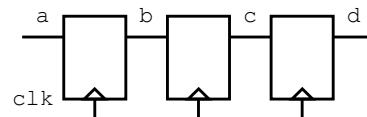
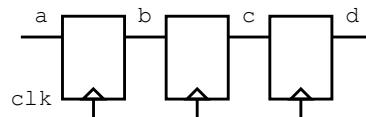
This is an example of position-dependent code. The statement order affects the functionality.

Multiple Nonblocking Assignments in Sequential Procedures

```
always_ff @(posedge clk)
begin
    b <= a;
    c <= b;
    d <= c;
end
```

All nonblocking assignments
infer registers

```
always_ff @(posedge clk)
begin
    d <= c;
    c <= b;
    b <= a;
end
```



Every nonblocking assignment in a clocked procedure infers a register.

- Regardless of order.

Here is the same illustration as previously shown but utilizing nonblocking assignments. By only using nonblocking assignments in sequential logic, the order of the assignment declaration cannot affect functionality. Every nonblocking assignment in a sequential procedure will infer a register for synthesis.

Temporary Variables in Sequential Procedures

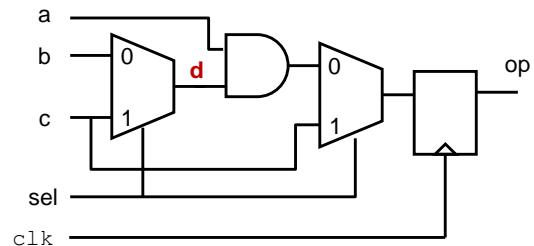
A sequential procedure may contain temporary variables...

- ...and so contain both blocking and nonblocking assignment.

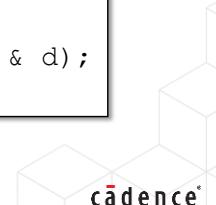
How to use:

- Declare temporary variables locally.
 - So not visible outside the block.
- Assign inputs to temporary variables.
 - Using blocking assignment =
- Evaluate algorithm using temporary variables.
 - Using blocking assignment =
- Assign temporary variables to outputs.
 - Using nonblocking assignment <=
 - These will infer registers.

Do not use both = and <= on the same variable.



```
always_ff @ (posedge clk)
begin
  logic d;
  d = sel ? c : b;
  op <= sel ? c : (a & d);
end
```



A sequential block may contain a large amount of combinational logic to calculate the next value of a register. The complexity of this logic may make it difficult to describe in one operation. We could break the combinational logic down into a series of separate, intermediate steps for easier definition and better readability. We could declare multiple local variables to store the output of the intermediate steps. The temporary variables must be assigned using blocking statements so that the result is immediately available for the next step, and so that registers are not inferred in synthesis. Then the output can be assigned from results of intermediate steps using nonblocking assignment to infer registers.

Note that this is an advanced usage model, and some companies may prohibit this coding style in their coding guidelines. In which case, you must partition the complex combinational logic into a separate combinational block.

You should not use both blocking and nonblocking assignment on the same variable. A simulation tool will typically accept this (although it may give warnings) but the synthesis tool will reject such code with error messages.

For the temporary variables to be truly temporary, they cannot be used anywhere other than that one block, so they must be declared locally.

In this example, we have declared a local variable d, to assist in computing the output value.

Submodule Summary

In this submodule, you learned how to use nonblocking and blocking assignments correctly.

This submodule described:

- Blocking assignment in combinational procedures.
- Nonblocking assignment in sequential procedures.
- Mixing blocking and nonblocking assignment in sequential procedures.
 - In a sequential procedure, making blocking assignments only to *temporary* variables.



You can now correctly choose between blocking and nonblocking assignments. This module reviewed blocking and nonblocking assignments, and then examined how to use them in procedural blocks meant to represent combinational logic or sequential logic, and lastly explored issues with statement order and mixing blocking and nonblocking assignments in the same procedural block.

Test Your Understanding

1. What is the primary difference between blocking and nonblocking assignments?
2. Where should you use blocking assignments?
3. Where should you use nonblocking assignments?
4. Where can you legitimately mix blocking and nonblocking assignments?



Please pause here for a moment to consider these questions. Refer to the module contents as needed. When you are ready, compare your answers to those on the next slide.

Solutions: Test Your Understanding

1. What is the primary difference between blocking and nonblocking assignments?
 - Blocking assignments execute immediately.
 - Nonblocking assignments calculate the RHS expression and schedules an update to the LHS assignment target for execution in the NBA event region.
2. Where should you use blocking assignments?
 - Use blocking assignments for combinational logic.
3. Where should you use nonblocking assignments?
 - Use nonblocking assignments for inferring registers in sequential logic.
4. Where can you legitimately mix blocking and nonblocking assignments?
 - You can make blocking assignments to intermediate (temporary) variables in a sequential procedure.

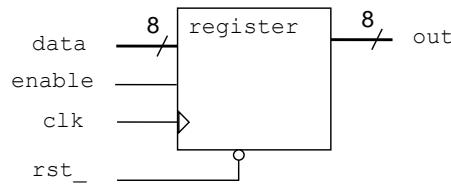


This page does not contain notes.

Lab

Lab 4 Modeling a Simple Register

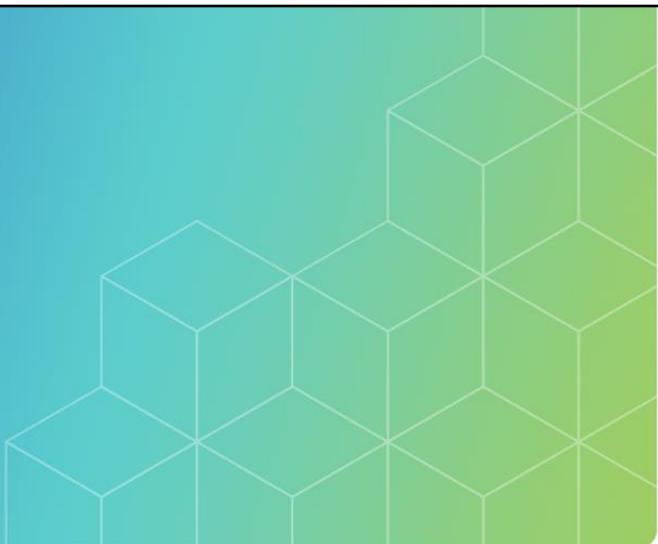
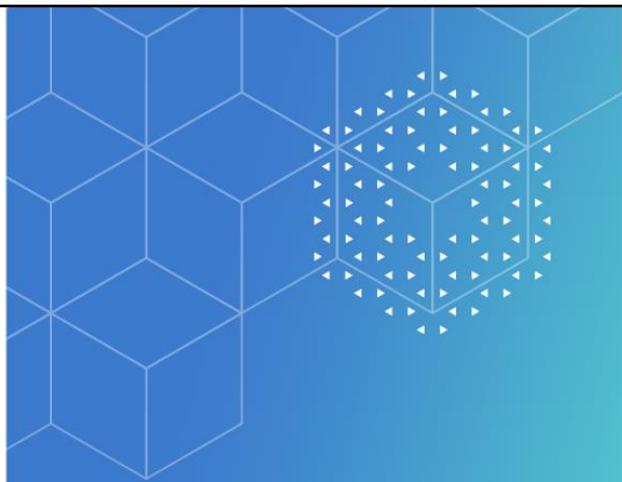
- Use procedural constructs to model a simple register.



371 © Cadence Design Systems, Inc. All rights reserved.



In this lab, model a simple register using the procedural constructs which we have discussed.



Submodule 4-6

User-Defined Data Types and Structures

cadence®

This page does not contain notes.

Submodule Objectives

In this submodule, you will:

- Name type declarations in SystemVerilog with `typedef`.
- Create types with user-defined, enumerated values.
 - Typically used for FSM state values and processor opcodes.
- Use structures to create arrays of different data types.



This page does not contain notes.

Defining Types

You can declare and name any type using `typedef`:

The type name can be used anywhere the type declaration is legal:

- For example: variable or port declaration.

Rules:

- You must declare `typedef` before the name is used.
- The declaration must be visible in the scope where the name is used.

An un-named type is known as an anonymous type.

```
typedef logic [7:0] vec8_t;
vec8_t busa, busb;
```

```
// typed (named) type
typedef logic [15:0] vec16_t;
vec16_t wordvar;
```

```
// anonymous (un-named) type
logic [15:0] wordvar;
```



A `typedef` declares and names a user-defined type, allowing the name to be used in the declaration of variables, ports and other types. You must declare a `typedef` before you use the name, either:

- Inside any declarative scope (for example, a module), which makes it visible only within that scope, and any nested declarative scopes.
- In the compilation unit. (See later for more details.)
- Inside a package, which makes it visible in all design elements that import the package. (See later for more details.)

Enumerated Types

A type with user-defined values:

- Value names must be unique in the current scope.

Declared as follows:

- Name enumerate with `typedef`.
- Keyword `enum`.
- Define base type.
 - Constrain size to number of values.
- Declare value name list in brackets `{ }`.
 - Typical to use uppercase for names.
 - Avoid identifier clashes with local variables.

```
typedef enum logic[1:0]
  {IDLE, START, PAUSE, DONE} state_t;

state_t state, next_state;
...
state = IDLE;
state = next_state;
```

Values mapped to the base type in order of the declaration.

00	01	10	11
IDLE	START	PAUSE	DONE



375 © Cadence Design Systems, Inc. All rights reserved.

An enumerate type is a user-defined type where the user defines the value set of the type. A SystemVerilog enumerate type is declared with the keyword `enum` followed by a declaration of the base type of the enumerate, followed by a list of values for the type. The base type should be a vector sufficient to store the number of values in the enumerate in (by default) a binary encoding. For example, 2-element vector for 4 enumerate values. The value names must be unique in whichever scope the enumerate type is used. A common policy is to use uppercase names for values to ensure uniqueness.

Enumerate type declarations should be typed (with `typedef`). There are significant benefits to naming the enumerate type as we shall see.

By default, the values of the enumerate are mapped to the base type in the order of declaration using binary encoding. This can be modified as we shall see.

For RTL code, your base type should always be a 4-state `logic` type to avoid initialization issues – see later.

Effectively an enumerate type is the base type with specific names (mnemonics) for selected values.

Enumerated Type Use

Enumerated types are strongly typed.

Should only be *directly* assigned:

- A named value of the type.
- A variable of the same type.

Direct literal assignment gives runtime warnings.

- Should use the type name to "cast" the literal.
 - `type_name'(value)`

An enumerate in an expression is replaced by its base type value.

```
typedef enum logic[1:0]
  {IDLE, START, PAUSE, DONE} state_t;

state_t state, next_state;
integer aint;
...
state = IDLE;
state = next_state;           Not legal

state = 2'b00;    // run time warning
state = 2;        // run time warning

state = state_t'(2); // PAUSE Legal

aint = state * 2; // 4

enum in expression
```



SystemVerilog enumerate types are strongly typed. An enumerate variable should be assigned only from a variable of the same type or from an enumeration value of its type.

To assign any other value requires a static cast to the enumerated type. A cast can only be made using the name of the enumerate type. Casting the value to the enumerated type does not check the validity of the value (see following slides).

Traditionally, SystemVerilog is not a strongly typed language. Therefore, most compilers allow the direct assignment of an integral value to an enumerated type variable but generate a runtime warning.

Tip: Always use a `typedef` to declare your enumerate types.

In this example, we have two enum variables `state`, and `next_state`, which are of same type. It is ok to assign the name directly as shown. It is also fine to assign one enum variable to other as they are of same type. Then, it is wrong to assign a value directly without casting and it results in run time warnings. After that, a value is assigned with casting, which is ok. Then, we can use the return value of the enum variable, which will be a number to use in other operations.

Explicit Enumerate Value Encoding

You can define explicit encoding for values.

- Allows one-hot coding, etc.

Value encodings must be unique.

- It is a compilation error if encodings overlap.

Explicit values must be sized to the base type length.

- To avoid truncation/padding.

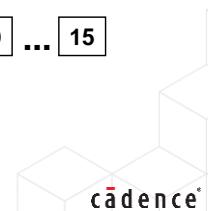
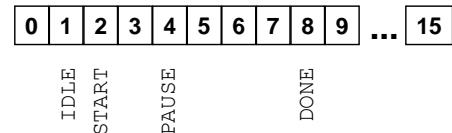
An enumerate variable can hold *any* value from the base type.

- Even if there is no matching enumerate value name.
- Casting does not check value.

An enumerated type is the base type with mnemonics for specific values.

```
typedef enum logic[3:0]
{
    IDLE   = 4'b0001,
    START  = 4'b0010,
    PAUSE  = 4'b0100,
    DONE   = 4'b1000} onehot_t;
onehot_t state;
...
state = onehot_t'(4'b0010); // START
state = onehot_t'(4'b0011); // !
```

No matching value name,
but `state` still assigned 4'd3



377 © Cadence Design Systems, Inc. All rights reserved.

Explicit encoding can be included in an enumerated type declaration. This allows one-hot, Gray or other encodings to be defined for enumerated values.

It is a compilation error if two or more values have the same encoding. Also, the encoding must be explicitly sized to match the size of the base type of the enumerate. Unsized or incorrectly sized values generate compilation errors. This is to avoid truncation or padding of the encoded value leading to duplication of encoding.

In RTL code, you can use only the binary values in explicit encoding (no Z or X).

Again, an enumerate type is the base type with specific names (mnemonics) for selected values. Therefore, `state` could hold **any** 4-element logic value (including x and z bits). `state` just has names (or aliases) for four of its possible values.

Note: For these encodings to be reflected in hardware, the synthesis tool must support explicit encoding. If, for example, you want to optimize the encoding of a state variable, it may be better to allow the synthesis tool to choose its own encoding based on your synthesis goals.

In this example, instead of default binary encoding, states have been explicitly encoded with one hot. Encodings have been explicitly sized to 4 bits to match the size of the base type.

Enumerated Type Access Methods

Methods are automatically defined for all enumerate variables.

Method	Description
first()	Returns first value
last()	Returns last value
next(N)	Returns next Nth value from current
prev(N)	Returns previous Nth value from current
num()	Returns number of values
name()	Returns string equivalent of value

```
typedef enum logic[3:0] {IDLE = 4'b0001,
                           START = 4'b0010,
                           PAUSE = 4'b0100,
                           DONE = 4'b1000} onehot_t;

onehot_t state;

initial begin
    state = state.first();
    repeat(5) begin
        $display ("%s = %0d", state.name(), state);
        if (state == state.last())
            $display ("-----");
        state = state.next();
    end
end
```

Iterates over value names only
i.e., next **START** is **PAUSE** not 4'd3

IDLE = 1
START = 2
PAUSE = 4
DONE = 8

IDLE = 1

378 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Access functions (called methods) are automatically defined for every enumerate variable. They give access to the values of the enumerate type and are useful for:

- Iterating over the valid values of an enumeration type.
- Extracting the current value as a string for printing.

As functions, the methods return a single value, which must be used in an expression.

The `next()` and `prev()` methods iterate through the defined values of the enumerate type in the order of declaration, regardless of the encoding or base type of the enumerate. `N` has a default value of 1 and is `unsigned`.

In the example, when `state = DONE`, then `state.next()` returns `IDLE`.

Therefore, these methods are a safe way to access and iterate over the enumerate values.

The `name()` enumeration method returns a string representation of the value which can be used with the “percent s” (%s) string format specifier to display the value of an enumerate variable.

If an enumerate variable contains an integral value that does not match the encoding for a named value from the declaration, then the `name()` method returns an empty string.

This example explicitly defines one hot encodings. Here initially, `state` is assigned with first value using `first` method, which is `IDLE`. Then, it displays the value and name, and it is assigned with next value using `next` method. It repeats this until it reaches last element. It identifies the last value using `last` method.

Enumerate Type Initialization

`enum` has a default base type.

- Default is `int` (32-bit 2-state).

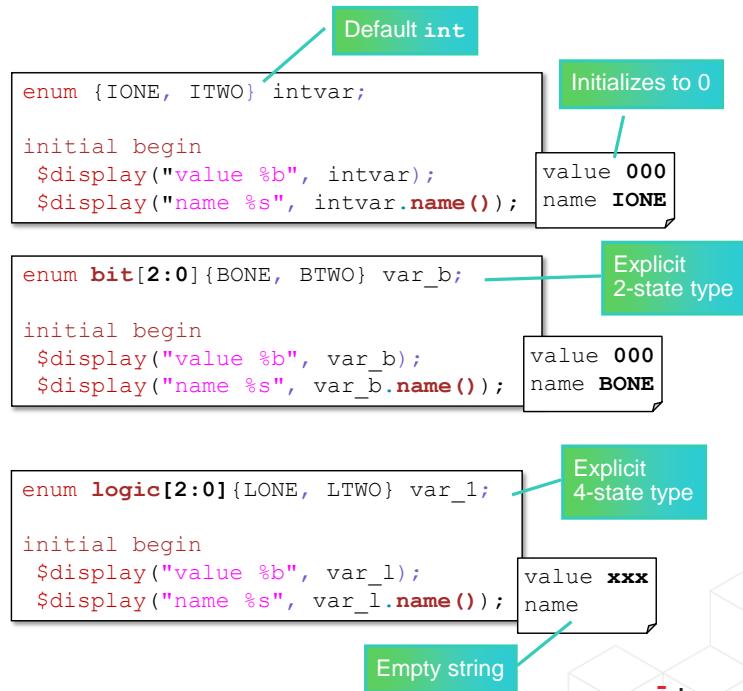
The initial value of an `enum` variable depends on its base type:

- 0 for default `int` or any explicit 2-state type.
 - May correspond to a value name.
- `x` for an explicit 4-state type.

Need a 4-state type for RTL.

- To detect initialization problems.

`name` for unknown value returns an empty string.



379 © Cadence Design Systems, Inc. All rights reserved.



The initial value of an enumerate variable depends on the base type of the enumerated declaration.

- For a default `int` base type, the initial value is 0, which would match the default encoding of the first enumeration value. Therefore, by default, the variable initializes to the first enumerate value. This can hide initialization issues and so you should never use the default `int` base type in RTL code.
- For any explicit 2-state type (e.g., bit array), the initial value is 0, which would match the default encoding of the first enumeration value. Again, do not use 2-state types for RTL variables.
- For an explicit 4-state base type (e.g., logic array), the initial value is all bits unknown ('x'), which would not match only an explicitly encoded value.

In the third example, the initial value of `enumerate var_1` is 3'bxxx. As this value does not match the encoding for any of the declared values, the `name()` method returns an empty string.

Therefore, for RTL code, you need an explicit 4-state `logic` base type for all enumerate declarations.

This will expose any initialization issues in the code.

Structures

It is a collection of data items of potentially different types.

Declare using the `struct` keyword.

Use the “dot” notation to access individual fields.

You can make pattern assignments to structures:

- Ordered or keyed:
 - Not both in the same pattern.
- Keys can be by name, type, default or a mix of these.

You can declare arrays of structures.

Structures can be nested.

```
typedef struct {
    logic      id, par;
    logic[3:0] addr;
    logic[7:0] data;
} frame_t;

frame_t f1, two_frame[1:0];
logic[7:0] data_in;
...
// individual field access
f1.id = 1'b1;
data_in = f1.data;

// ordered assignment pattern
f1 = '{1'b1, 1'b1, 4'h0, 8'hff};

// named assignment pattern
f1 = '{id:0, par:1, addr:0, data:0};

// nested ordered assignment pattern
two_frame = '{'{0,0,0,255}, '{1,1,1,0}};
```

380 © Cadence Design Systems, Inc. All rights reserved.



A structure is an array of elements that can be of different data types. It can be used to group complex data information, which would be otherwise spread over several separate variables into one object.

The individual fields of the structure can be accessed from a structure variable using the dot notation.

By default, the fields of a structure are stored and must be accessed separately. Therefore, the easiest way to load a structure is using a pattern assignment. An assignment must be made to every field but can be either ordered or keyed. With structures, the key can include assignment by type as well as assignment by name and default (see next slide).

Structures are just like any other type in SystemVerilog, and therefore you can declare arrays of structures or structures containing fields that are of another structure type (nested structures).

Packed Structures

RTL structures should be packed.

- Using keyword packed.

All fields must be “packable”:

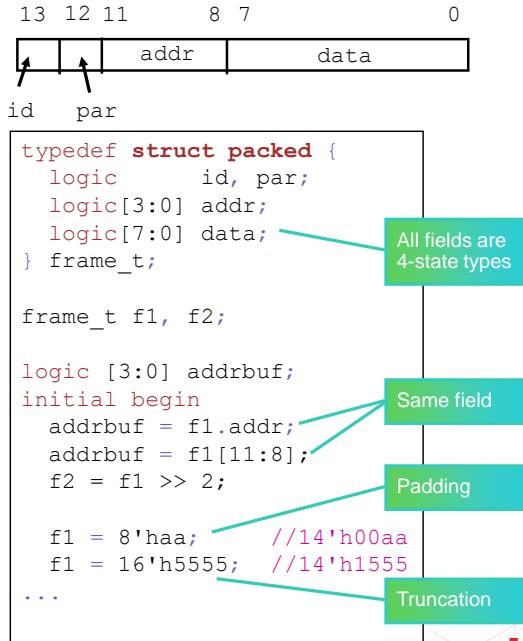
- Integral values only.
 - Any field that is an array or structure must also be packed.

A packed structure can be treated as a one-dimensional vector.

- Can still access individual fields.

Much more convenient, e.g.:

- Load structure through individual fields.
 - Transmit one bit at a time using a shift.



By default, a structure's fields are stored and assigned individually. However, a structure can be stored and accessed as a single one-dimensional array by using the keyword `packed`.

There are some limitations on the field types for a packed structure. Only integral fields can be packed. Multidimensional arrays are allowed if they are also packed (see module on static arrays), and any nested structure fields must also be packed.

A packed structure can be assigned, read, sliced, shifted, and operated upon as if it were a single one-dimensional array. There are two ways to access the fields of a packed structure. You can still use the dot notation to access a field, but you can also slice the structure.

For example, `f1.addr` and `f1[11:8]` both access the same information.

Submodule Summary

This submodule covered user-defined data types.

User-defined type declarations (`typedef`):

- Lets you define a type that is used throughout the design.

Enumerations (`enum`):

- Names states in a state machine or opcodes in an instruction set.

Structures (`struct`):

- A vector of different data types, referenced by a single name.
- Makes for more readable code, with less typing.
- Simplifies the passing of data across module/task/function boundaries.
- Packed structures allow bit-vector operations on the whole structure and define an implementation for synthesis.



We have learnt how to use user defined types using `typedef`. We also seen how to utilize enumerations. Finally, we have seen about structures and packed structures.

Test Your Understanding

```
typedef enum logic[1:0] {UP=2'b00, STAY=2'b01, DOWN=2'b11} cstate_t;  
cstate_t cstate;  
  
typedef struct {logic[3:0] tag;  
                cstate_t mode;  
                logic[7:0] data;} tagdata_t;  
tagdata_t tdata;  
  
typedef enum {IDLE, GO, STAY, RESET} mechstate_t;  
  
logic [7:0] slice;  
  
initial begin  
    tdata.mode = RESET;  
    tdata = '{tag:4'h0, data:8'hff};  
    slice = tdata[7:0];  
    cstate = cstate + 1;  
    cstate = cstate_t'(2'b10);  
    slice = cstate + 1;  
end
```



Question

Which of these statements are incorrect?
How could you correct them?

Please pause here for a moment to consider these questions. Refer to the module contents as needed.
When you are ready, compare your answers to those on the next slide.

Solution: Test Your Understanding

```

typedef enum logic[1:0] {UP=2'b00, STAY=2'b01, DOWN=2'b10} cstate_t;
cstate_t cstate;

typedef struct {logic[3:0] tag;
               cstate_t mode;
               logic[7:0] data;} tagdata_t;
tagdata_t tdata;

typedef enum {IDLE, GO, STAY, RESET} mechstate_t;

logic [7:0] slice;

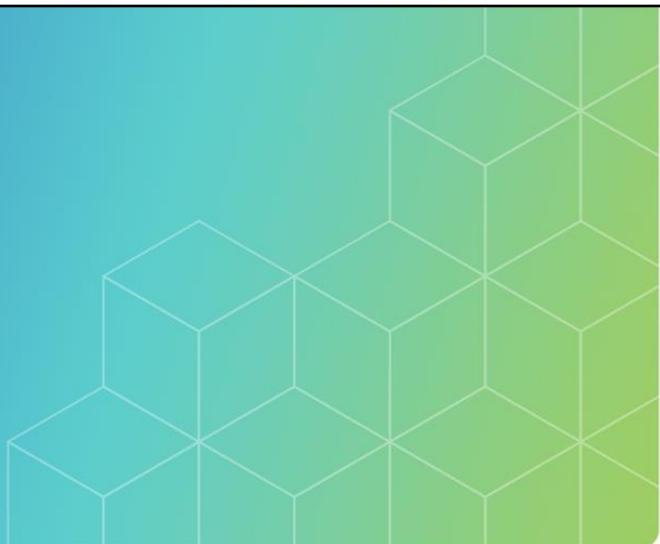
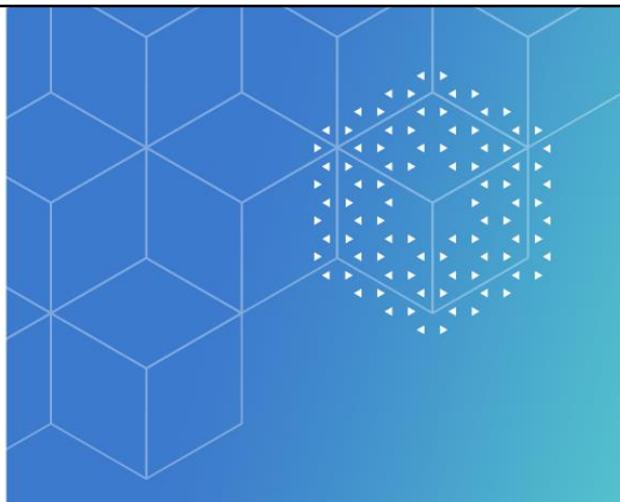
initial begin
    tdata.mode = RESET; Duplicate declaration for stay
    tdata = '{tag:4'h0, data:8'hff}; Unknown value RESET in cstate_t
    slice = tdata[7:0]; Missing assignment for mode field
    cstate = cstate + 1; Cannot slice unpacked structure
    cstate = cstate_t'(2'b11); Should replace this with
    slice = cstate + 1; cstate = cstate.next();
end

```

384 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Submodule 4-7

Packages

cadence®

This page does not contain notes.

Submodule Objective

In this submodule, you will:

- Define and reference declarations in packages.



This page does not contain notes.

Shared Declarations

Declarations may need to be shared.

- E.g., between several modules.
 - Maybe in a port connection.

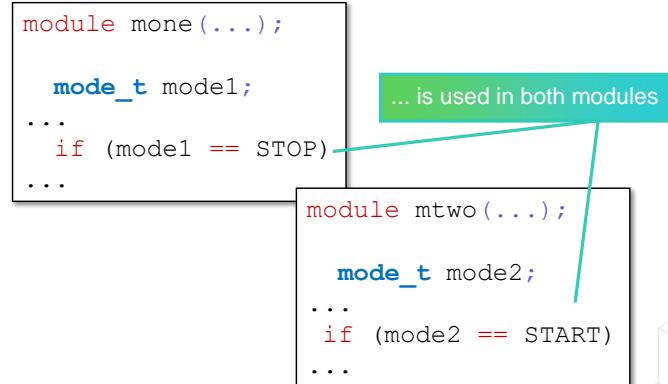
```
mode_t declaration...
typedef enum logic {START,STOP} mode_t;
```

Duplicate declarations are a bad design practice.

- Difficult to read and maintain.
- Can cause compilation issues.

Declarations should be made once and made visible to multiple modules.

- Packages allow this.



387 © Cadence Design Systems, Inc. All rights reserved.

cadence®

In real world designs, we have to share the declarations between several modules. Declaring them in all the places is a bad design practice. It causes compilation issues and is difficult to maintain. If a shared declaration needs some modification, it has to be modified in all the places it is used. Instead, if we had a provision, where we can declare it once and make that declaration visible to any module where that declaration is used, it would be perfect. SystemVerilog provides packages for this purpose.

To resolve these issues, SystemVerilog declares packages which can be used for sharing declarations.

Packages

New design element similar to a module:

- Must be compiled separately.
- Must be compiled before elements that reference the package.

They contain shared declarations:

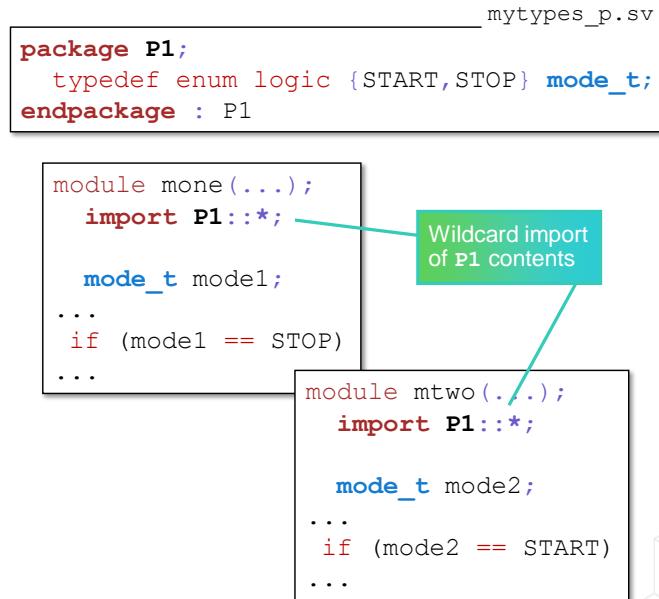
- Types, variables, subroutines...

`import` the package to access declarations.

- `::` is a scope resolution.
- `*` is an implicit (wildcard) import.

Declarations must be imported before use.

- Place imports at the beginning of the module.
 - Fewer visibility issues.
 - Better readability.



388 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The package is a design element. A design element is a top-level simulation building block that is typically declared in a separate file and must be separately compiled.

A package allows you to share declarations between multiple design elements. You can place almost any declaration – types, variables, tasks and functions – within a package.

You can reference packages from within other design elements – modules, interfaces, and other packages. There are three ways to reference the declarations from a package:

- An implicit wildcard import allows you to reference all the package declarations.
- You can also reference a declaration using the package name and the scope resolution operator.
- An explicit import allows you to reference selected package declarations.

These example declares a package p1. modules m-one and m-two implicitly import the entire contents of the package with wildcard import.

Wildcard Import

Everything from the package is imported...

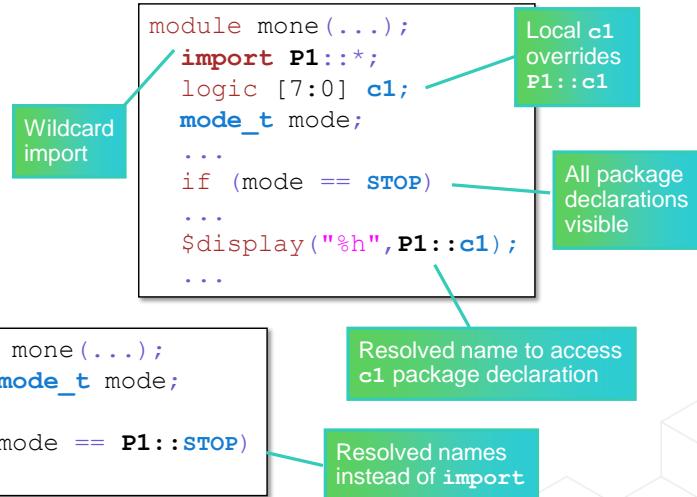
... but declarations are only candidates for import:

- Local declarations take precedence over wildcard imported declarations....
- ...although package declarations can still be accessed using a resolved name.
- <package name>::<declaration>

You could use resolved names instead of import...

- ... but lose readability.
 - Harder to see package dependency.

```
package P1;
localparam c1 = 10;
typedef enum logic {START,STOP} mode_t;
endpackage : P1
```



389 © Cadence Design Systems, Inc. All rights reserved.

cadence®

An implicit wildcard import makes the package types candidates for import. It imports each type only when the type is used. If the design element can have visibility of another declaration using the same identifier, either through a local declaration or an explicitly imported package declaration, and the other declaration is visible before the identifier is used, then the wildcard imported declaration is overridden.

However, declarations from the package are still visible even if overridden by local or explicitly imported declarations by using a resolved name.

A wildcard import makes all the declarations in the imported package visible, including enumerate type values.

This example contains a constant and an enumerate declaration in package p1. module m-one implicitly imports all the contents of p1. It also declares a variable with same name as constant in the package. Since we have implicitly imported package contents, local declarations overrides the package declaration. If you want to access package constant, you can use resolved name using scope resolution operator as shown.

Explicit Import

Alternative to wildcard import.

Directly loads declaration into the module.

Declaration must be unique in the current scope:

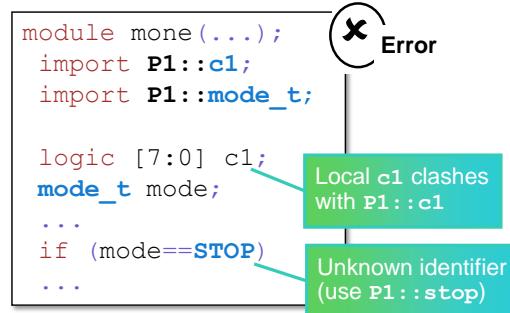
- Compilation error if local or other explicitly imported declarations have same name.

Only the symbols specifically referenced are imported.

- Need to explicitly import every value of an enum.
 - Or use a resolved name.

Not widely used.

```
package P1;
  localparam c1 = 10;
  typedef enum logic {START,STOP} mode_t;
endpackage : P1
```



390 © Cadence Design Systems, Inc. All rights reserved.



An explicit import directly loads a package declaration into the design element as if it was declared in the design element. As such, if the design element can have visibility of another declaration using the same identifier in the same scope, either through a local declaration or another explicitly imported package declaration, then it is a compilation error.

An explicit import only makes the identifiers in the import statement visible. For example, if you explicitly import an enumerated type, that does not automatically also import the enumeration value names. However, if you explicitly import a structure type, the field names are visible in the design element.

Explicit import is not widely used and should be avoided if possible. Its only benefit is to prevent local declarations from using the same identifier as the explicitly imported declarations. However, you may not have control over the identifiers used in either package or module. Explicit import disadvantages usually outweigh the advantages, and the same functionality can usually be achieved using wildcard import and resolved names.

In this example, same contents of package are imported explicitly. Since, you have a local declaration with same name, due to explicit import with same name, it would result in compilation error. Also, explicit import of mode_t, does not import its value names.

Multiple Packages and Import

Can list multiple packages in one import.

- `import P1::*, P2::*;`

Packages can import other packages.

However, imports **do not** chain:

- `P1` is imported into `P2` to use `c1`.
- Importing `P2` into `mone` does not make `c1` visible in `mone`.
- `P1` must be separately imported into `mone` for access.
- ...or add an export to `P2` to make `P1` visible as part of `P2`
 - `export P1::*`;
 - Rarely used.

```
package P1;
localparam c1 = 5;
...
endpackage : P1
```



```
package P2;
import P1::*;
localparam c2 = c1;
...
endpackage : P2
```

 Error

```
module mone(...);
import P2::*;

initial begin
$display("%0d",c2);

$display("%0d",c1);
...
```

Error –
c1 undefined

391 © Cadence Design Systems, Inc. All rights reserved.



Imports do not chain. In the example, package `P2` contains an `import` for package `P1`, which allows it to access the value of `c1`. However, when we import `P2` into the module `mone`, we only see the declarations from `P2`, not `P1`. Therefore, the reference to `c1` fails. To see declarations from `P1` in module `mone`, package `P1` must be separately imported into the module.

An alternative is to make the declarations imported from `P1` visible as part of the `P2` package by adding an export statement to `P2`, although exports are rarely used in practice.

`export P1::*;`

The export can be wildcard or explicit. A special wildcard export form exports all declarations imported into a package:

`export *::*;`

Shared Declarations in Port Lists

Package declarations may be used in module port lists...

...but package must be imported before use.

- import inside module is insufficient.

The import could be moved before the module header.

- Into the Compilation Unit Scope (COS).

This is not recommended.

- The scope of a COS can be changed with compile options:
 - One file, multiple files or all files.
- Hard to read and maintain.
- There are better options...

```
package P1;
  typedef enum logic {START,STOP} mode_t;
endpackage : P1
```

```
module mone (input mode_t mode,
  ...);
  import P1::*;
  ...
endmodule
```

Error - mode_t undefined

```
import P1::*; //COS
module mone (input mode_t mode,
  ...);
  ...
endmodule
```



If you are using package declarations in an ANSI C module port or parameter list, then using an import statement inside the module is insufficient. The import is compiled after the package declaration use, and so the declarations cause compilation errors.

An obvious option is to move the import statement to before the module header. The import is still in the same file as the module declaration, but outside the module itself. This is called the Compilation Unit Scope (COS). We could put any declaration in the COS, not just package imports. However, using the COS can be dangerous, specifically in that, the visibility of the scope can be modified with compiler options. So, the COS declaration in a specific file could be made visible just to the module in that file, or to modules in multiple files compiled at the same time, or to all files in the simulation. As this is a compile-time option, then the visibility of a COS declaration could be modified from one simulation run to the next. Obviously, having many files in your design dependent on a COS declaration, in just one file, is hard to read, and maintain. Therefore, we should avoid COS declarations if possible.

Importing Packages in the Module Header

Several options for shared declarations in module port lists.

- Use resolved names:
 - Inefficient for multiple declarations.
- Import package as part of module header:
 - Imported before parameter and port lists.
 - Import clause must be terminated with a semicolon.

```
package P1;
  typedef enum logic {START,STOP} mode_t;
endpackage : P1
```

```
module mone (input P1::mode_t mode,
  ...);
  import P1::*;
  ...
endmodule
```

Resolved name

```
module mone import P1::*;
  (input mode_t mode,
  ...);
  ...
endmodule
```

import in
module header

import must
include semicolon

393 © Cadence Design Systems, Inc. All rights reserved.



If you are using package declarations in an ANSI C module port or parameter list, then using an import statement inside the module is insufficient. The import is compiled after the package declaration use and so the declarations cause compilation errors.

There are two solutions to this issue:

1. Use resolved names for all package declarations, although this is inefficient if you are using multiple declarations from the package.
2. Packages can also be imported as part of a module header. This makes all the package items visible throughout the module and these package items can also be used in parameter or port declaration of the module. This option can also be used in other design elements such as interfaces and programs. The module import must come before the ANSI C port or parameter list of the module and must be terminated with a semicolon, as if you were executing an import statement.

Submodule Summary

This submodule covered packages:

- Defining packages for shared declarations.
- Wildcard and explicit package import.
- Multiple packages and package chaining.
- Importing packages into the module header to avoid use of the Compilation Unit Scope.

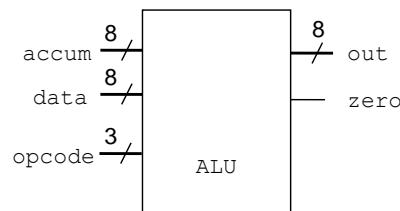


We have learned how to define packages for shared declarations, how to do various types of package imports, and their applications. We also seen how to import the package as part of module declaration.

Lab

Lab 5 Remodeling the ALU

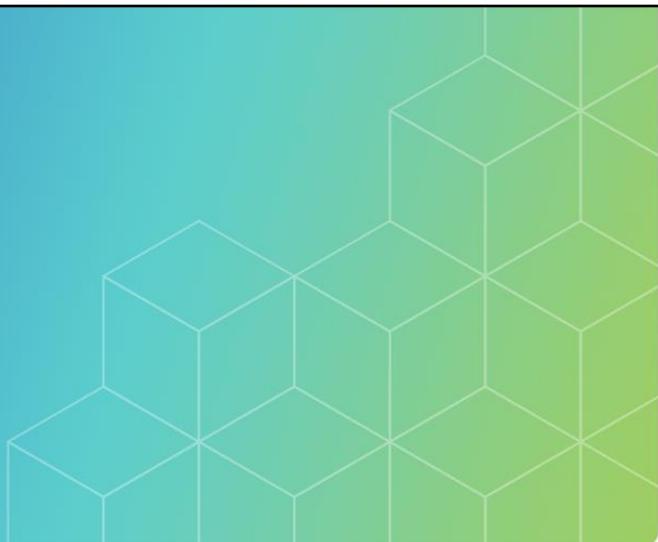
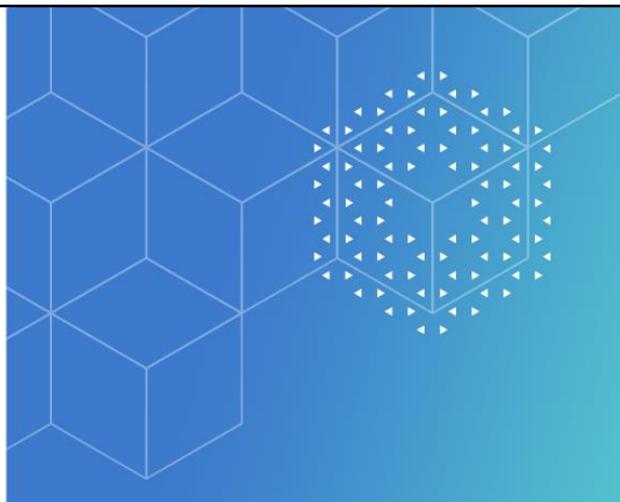
- Using enumerated types and packages.



395 © Cadence Design Systems, Inc. All rights reserved.



In this lab, you will model an ALU using enumerate datatypes, enumerate methods and packages.



Submodule 4-8

Coding RTL for Synthesis

cadence®

This submodule intends to provide you with the information you need to code your RTL design.

Submodule Objective

In this submodule, you will:

- Code design behavior for logic synthesis.

Topics include:

- Modeling combinational logic
- Modeling sequential logic
- Modeling latch logic
- Modeling three-state logic
- Using synthesis attributes



Your objective is to code design behavior for logic synthesis.

To do that, you need to know about:

- Modeling combinational logic
- Modeling sequential logic
- Modeling latch logic
- Modeling three-state logic

Modeling Combinational Logic

Output is at all times a combinational function solely of the inputs.



```
assign sum = a ^ b;
```

Modeled by:

- assign statements.
 - always_comb procedural block.
 - Implicit, complete event list.
 - Uses = blocking assignment.
 - Variables driven always_comb cannot be driven by anything else.
 - Detects common error condition.

```
always_comb
begin
  if (sel == 1
    op = b;
  else
    op = a;
  ...
end
```

All variables read in block automatically added to event list

op cannot be
driven elsewhere

398 © Cadence Design Systems, Inc. All rights reserved.



You can model combinational logic with a continuous assignment or an always comb statement.

An `always_comb` has a complete, implicit event list, in that all variables read in the block are automatically added to the event list, to trigger block execution if any change value.

Assignments to variables should be made with blocking assignments in always comb blocks.

Also, if an `always_comb` block assigns to a variable, then the block owns the driver on that variable, and the variable cannot be driven by anything else, such as another `always_comb`, `assign`, or module output.

The complete event list and driver ownership fix common synthesis issues with combinational logic. This is why we use `always comb` instead of a general `always` block.

Incomplete Assignments

Output is at all times a combinational function solely of the inputs.

Outputs should be updated every time `always_comb` is executed.

- If not, this is an incomplete assignment.

```
always_comb  
if (b)  
y = a;
```

```
always_comb  
case (b)  
1: y = a;  
endcase
```

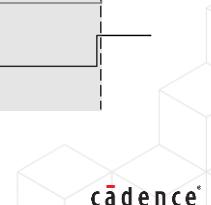
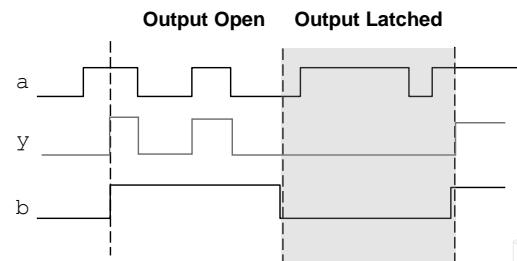
What is the value of `y` when `b` is 0?

- If `y` is not updated, it retains its previous value.

How is this implemented in hardware?

- Using a latch.

How can we fix these issues?



399 © Cadence Design Systems, Inc. All rights reserved.

If an execution path through a combinational procedure exists that does not update the value of some output, then that output variable must retain its previous value. The synthesis tool infers a latch to implement this behavior. Latch inference is almost always not intended, and you can easily avoid it.

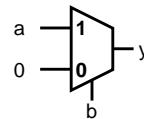
This example fails to update the `y` output variable when the `b` input is not 1.

How would you modify this code to ensure inference of purely combinational logic?

Complete Assignments

Provide outputs with a value for every known combination of inputs.

- Unconditional else for if.
- default for case.
- Default assignments:
 - At start of block.
 - Can be overridden by later assignments.
 - "Last one in wins."
 - Easiest for complex conditional logic.



```
always_comb
if (b)
    y = a;
else
    y = 1'b0;
```

```
always_comb
case (b)
1:           y = a;
default:     y = 1'b1;
endcase
```

Default assignments

```
always_comb
begin
    y = 1'b0;
    if (b)
        y = a;
end
```

```
always_comb
begin
    y = 1'b0;
    case (b)
    1: y = a;
    endcase
end
```

400 © Cadence Design Systems, Inc. All rights reserved.



Here are several methods you can use to prevent latch inference:

- Use an explicit else clauses for if statements
- Use default in a case statement
- Define default values for all outputs at the start of the always_comb.

Which is the best technique in a real design?

If you have a procedure with a complex set of conditional assignments, you can easily miss an assignment for one or more of these branches. Making default assignments at the start of the procedure ensures that all procedure outputs have an assignment.

Complete Assignments Are Binary Only

Unknown values (with x/z) do not need an output for a complete assignment.

- Only binary encoding must be complete.
- Latches not inferred for missing unknown values.

May help in debug to detect unknown values.

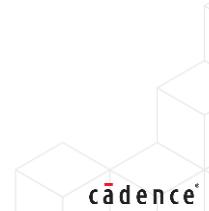
- So, default or unconditional else still useful.

```
logic [1:0] select;

always_comb

    case (select)
        2'b00: y = a;
        2'b01: y = b;
        2'b10: y = c;
        2'b11: y = d;
    default: y = 1'b1;
    endcase
```

Missing this
default will not
infer a latch



To avoid latch inference in case statements, we only need to consider binary values for complete assignment. Latches will be inferred on missing binary values, but not missing unknown values (containing x or z). It might not be necessary to have a default statement every time.

Therefore, in the example, if the `default` statement was missing, we would not infer latches as the full binary encoding of `select` is covered in the case branches.

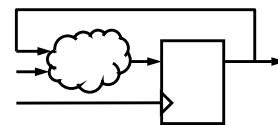
This should not discourage you from using `default` branches in all your case statements.

Modeling Sequential Logic

Outputs are sampled in registers on a clock edge; thus, storage is required.

Modeled with `always_ff` procedural blocks using a specific code template.

- Event list must contain one or more `posedge` or `negedge` events.
 - One event represents the active clock edge.
 - Other events may represent asynchronous set or reset.
- Uses `<=` nonblocking assignment to infer registers.
 - May also use blocking assignment for temporary variables.
- Variables driven `always_ff` cannot be driven by anything else.
 - Detects common error condition.



```
always_ff @ (posedge clock)
  d <= q;
```

402 © Cadence Design Systems, Inc. All rights reserved.



You model sequential logic using an `always_ff` block that has one or more `posedge` or `negedge` events in exactly one event list. Exactly one of those events represents the active clock edge that stores the value. Any additional `posedge` or `negedge` events represent asynchronous set and reset behaviors. The event list must not contain level-sensitive events.

To avoid simulation clock/data races, you should make only nonblocking assignments to variables that represent storage. You make blocking assignments to temporary variables. Temporary variables are those written and then read in the same procedure and nowhere else.

If an `always_ff` block assigns to a variable, then the block owns the driver on that variable, and the variable cannot be driven by anything else, such as another `always_ff`, `assign`, or module output.

The driver ownership fixes common synthesis issues with sequential logic. This is why we use `always_ff` instead of a general `always` block.

Simple (But Wrong) Counter

Specification for 0-9 counter.

- On rising edge of `clk`:
 - If `count` is maximum (9), set `count` to 0.
 - Otherwise increment `count`.

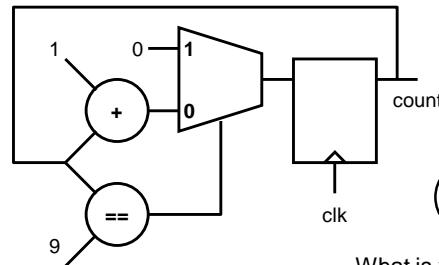
Synthesis infers registers on all nonblocking assignments.

```
logic [3:0] count;

always_ff @(posedge clk)
  if (count == 9)
    count <= 4'd0;
  else
    count <= count + 4'd1;
```

However, this code has an issue.

- What is the problem?
 - Hint: what is the starting value of `count`?



What is wrong with this?



403 © Cadence Design Systems, Inc. All rights reserved.

Synthesis tools recognize sequential procedures by looking for a particular code template – in this case, a procedure with an event list containing only edge-qualified signals. Although some synthesis tools may support other coding styles for sequential procedures, your adherence to this standard produces code that you can port between all synthesis tools compliant with the standard.

This example has only the positive edge of the clock in its event list. All storage inferred for non-temporary variables that this procedure writes will have a rising active clock edge. The `if` statement describes the combinational logic calculating the new `count` value that is stored on the next rising clock edge.

Note that this example lacks a reset to initialize the count value!

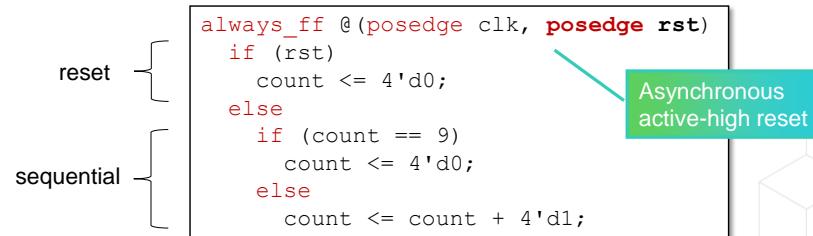
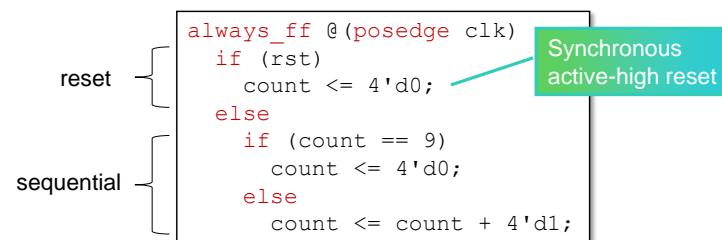
Reset Behavior

Use an if...else statement to define set / reset behavior.

- Set/reset takes priority over clock.
 - Put the set / reset behavior in the first if branch.
- Put the normal sequential behavior in the else branch.

Only difference between synchronous and asynchronous set/reset blocks is event list.

- Asynchronous set/reset are added to list.



404 © Cadence Design Systems, Inc. All rights reserved.



Set and reset behaviors are defined using an `if` statement. This is a requirement for both asynchronous set and reset behaviors and synchronous set and reset behaviors.

In the `if` statement, place the set and reset behaviors in their order of priority in the first conditional branches, and place the normal synchronous behavior in the unconditional last `else` branch. Do not make place statements outside of the `if` statement.

For synchronous set and reset, trigger the procedure on only the clock edge.

For asynchronous set and reset, trigger the procedure also on the active set or reset edge(s).

Code synchronously reset, asynchronously reset and not reset registers in separate procedures.

Sequential Procedure Templates

Code must follow these templates.

Remember to match event edge and if condition for asynchronous set/reset.

- E.g., active-low is negedge and if (!rst).

```
always_ff @(posedge clk)
begin
    // normal
    // sequential
    // behavior
end
```

```
always_ff @(posedge clk)
if (!rst)
begin
    // synchronous
    // active-low reset
    // behavior
end
else
begin
    // normal
    // sequential
    // behavior
end
```

```
always_ff @(posedge clk, negedge rst)
if (!rst)
begin
    // asynchronous
    // active-low reset
    // behavior
end
else
begin
    // normal
    // sequential
    // behavior
end
```

Match edge
and if

405 © Cadence Design Systems, Inc. All rights reserved.



Sequential procedures have the clock edge in the event list, and also the set and reset edge(s) if there is asynchronous set or reset.

Separately code your not reset, synchronously reset and asynchronously reset procedures.

Incomplete Assignments in Sequential Logic

Incomplete sequential assignment infers sequential feedback.

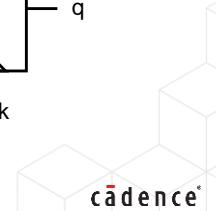
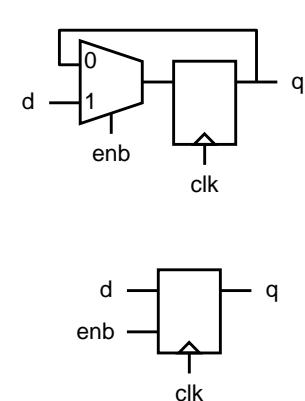
- If the output is not updated, it stays the same.
- Inferred registers store value.
- Implemented as a register feedback path.

```
always_ff @ (posedge clk)
  if (enb)
    q <= d;
```

Inferred sequential feedback is a good practice.

- Simpler implementation:
 - No need for `else`, `default` or `default assignments`.

Potential Implementations



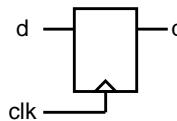
If procedure execution does not update a variable, then the variable value is not changed. In procedures representing combinational logic, this infers a latch to store the previous variable value. For procedures representing synchronous logic, the variable value is stored in the inferred register. For these procedures you do not need default assignment or `else` clauses to prevent latch inference.

Temporary Variables in Sequential Procedures

Temporary Variable

- Written first and then read (in same procedure).
- Variable is an alias for expression, so no register is inferred.

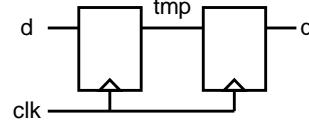
```
// 1 register
always_ff @(posedge clk)
begin
    logic tmp;
    tmp = d;
    q <= tmp;
end
```



Persistent Variable

- Read first and then written (in same procedure).
- Synthesis must infer a register to hold the value to the next read.

```
// 2 registers
always_ff @(posedge clk)
begin
    logic tmp;
    q <= tmp;
    tmp = d;
end
```



407 © Cadence Design Systems, Inc. All rights reserved.



You can use a temporary variable to hold the intermediate result of a calculation before you use the result later in the procedure. You use temporary variables to break up complex expressions and combinational logic into a series of smaller steps – making complex logic easier to describe, understand and maintain.

A temporary variable is one that a procedure writes with a blocking assignment only before the procedure reads it, and no other procedure uses it. You can declare temporary variables locally to ensure that no other procedure uses it.

A typical use model would be to make blocking assignments to temporary variables and then make nonblocking assignments of the temporary variable values to other variables.

If you write the temporary variable first, then read, in the execution of the procedure, then the temporary variable does not store values between block executions and synthesizes to a connection.

If you read the temporary variables first, then write, in the execution of the procedure, then the read is from the previous execution of the procedure, on the last clock edge. Synthesis will infer a register to store this value from the previous clock edge.

You should check the inferred register counts carefully in synthesis reports to ensure they meet your expectations.

Modeling Tri-State Logic

Bi-directional or multiply-driven connections are modelled in tri-state logic.

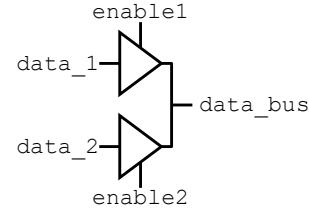
- Connections must be defined as nets (`wire`).
 - Restrictions on multiple drivers for `logic`.

Conditional assignment of a net (`wire`) to high-impedance (`z`) implies tri-state drivers.

- Cannot drive tristate from `always_comb` or `always_ff`.
 - Restrictions on multiple drivers for these blocks.
- Must use `assign`.

```
wire data_bus;
assign data_bus = enable1 ? data_1 : 8'bzz;
assign data_bus = enable2 ? data_2 : 8'bzz;
```

Ensure the enable signals are mutually exclusive!



408 © Cadence Design Systems, Inc. All rights reserved.



You model three-state logic by assigning the high-impedance (`z`) value to a net. Any other assignments to that net must also assign the high-impedance value. Further propagating the high-impedance value by use of net assignments does not also make that downstream logic into three-state logic.

This example codes behavior representing three-state drivers in a form that synthesis can recognize. For this example:

- The `enable1` signal when high drives `data_1` onto the data bus; and
- The `enable` signal when high drives `data_2` onto the data bus.

If multiple enables are simultaneously true, then you have bus contention.

Submodule Summary

You should now be able to properly code hardware for logic synthesis.

This submodule discussed the following:

- Modeling combinational logic
 - In an `always_comb` block with complete sensitivity list, blocking assignments, and complete assignment to avoid latches.
- Modeling sequential logic
 - In an `always_ff` block sensitive to edge-qualified events, nonblocking assignments, and set/reset behavior in early conditional statement branches.
- Modeling three-state logic
 - By conditionally assigning the high-impedance (`Z`) state.



This module explained how to describe hardware for logic synthesis. It addressed combinational, sequential, latch, and three-state logic. It introduced synthesis attributes that you embed in the source code to influence the synthesis process.

Test Your Understanding

1. In what conditions does logic synthesis infer a latch in combinational logic?
2. Explain the conditions under which a synthesis tool may infer a register for a blocking assignment in a sequential procedure.
3. In what construct does synthesis infer a three-state gate?



This page does not contain notes.

Solutions: Test Your Understanding

1. In what conditions does logic synthesis infer a latch in combinational logic?
 - If you fail to specify an output value for at least one combination of input values, then logic synthesis will insert a latch.
2. Explain the conditions under which a synthesis tool may infer a register for a blocking assignment in a sequential procedure.
 - A temporary variable in a sequential procedure is assigned with a blocking assignment. However, if the variable is *read* before it is *written*, in the sequential execution of the block statements, then synthesis infers a register for the variable.
 - If the variable is *written* before it is *read*, then it is only a temporary variable.
3. In what construct does synthesis infer a three-state gate?
 - Synthesis infers a three-state gate if you conditionally assign the high-impedance value to a variable.

411 © Cadence Design Systems, Inc. All rights reserved.

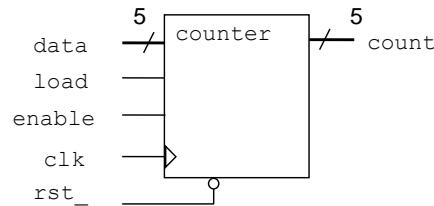


This page does not contain notes.

Lab

Lab 6 Modeling a Simple Counter

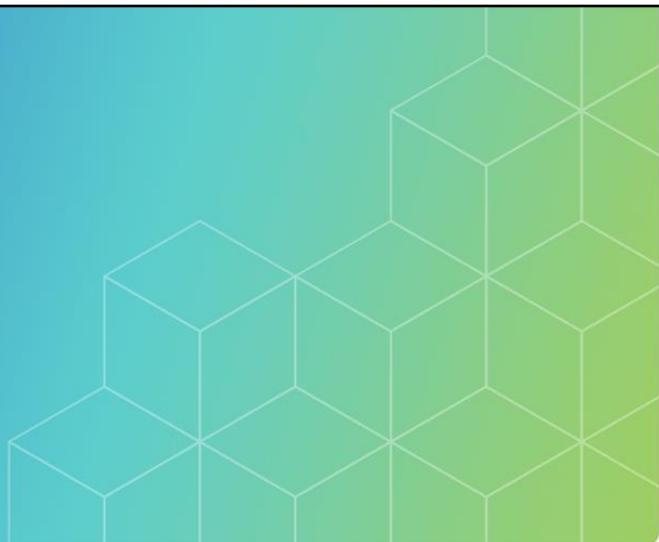
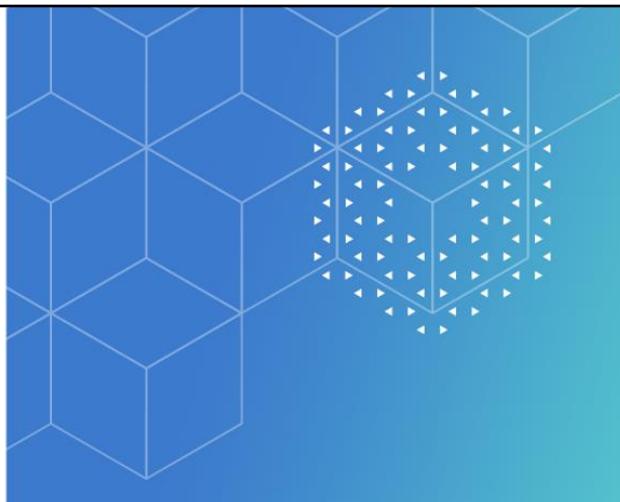
- Using sequential procedural blocks correctly to model a simple counter.



412 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Submodule 4-9

Designing Finite State Machines

cadence®

This submodule more specifically presents the various ways to code a state machine for synthesis.

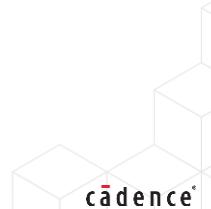
Submodule Objective

In this submodule, you will:

- Code state machines for synthesis.

Topics include:

- FSM introduction
- Defining the FSM states
- Example read-write synchronizer FSM
- Coding FSMs in various styles
- Various ways to optimize FSMs



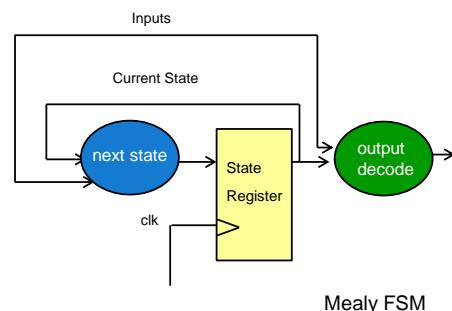
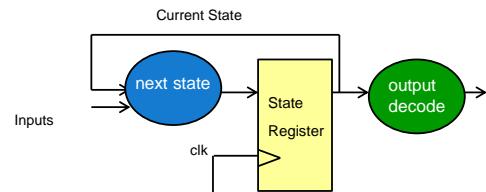
Your objective is to code state machines for synthesis.

To help you do that, this module discusses what an FSM is, how to define FSM states, various ways to code FSMs for synthesis, and various ways to optimize FSMs.

Finite State Machine (FSM) Review

FSM Structure

- State Register
 - Stores current state.
- Next State decode logic
 - Decides next state based on current state and inputs.
- Output Logic
 - Decodes state (or states and inputs) to produce outputs.
- Outputs from the FSM can be a function of:
 - Current state only – Moore.
 - Current state and the current inputs – Mealy.



415 © Cadence Design Systems, Inc. All rights reserved.



An FSM consists of state-encoding combinational logic, state-storing sequential elements, and output-decoding combinational logic.

- A Moore machine has no combinational path from inputs to outputs. The outputs are a function solely of the current state vector.
- A Mealy machine has at least one combinational from an input to an output. The outputs are a function of the current state and at least one current input. Mealy outputs can be available up to one clock earlier path than Moore outputs but may complicate synthesis timing constraint definition.

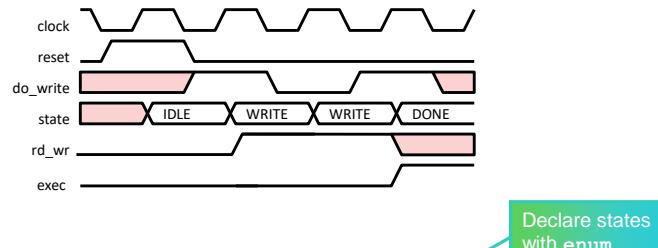
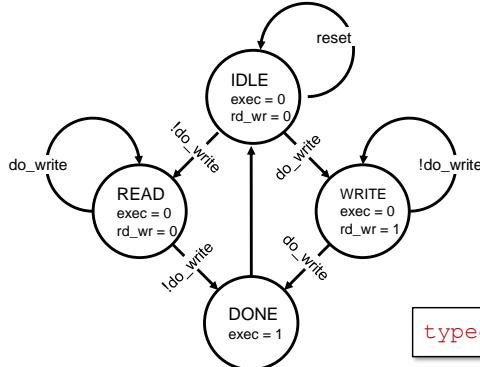
If the current state of your design depends at least partially upon the previous state, then your design is an FSM.

Perhaps one way to remember which is the Mealy machine and which is the Moore machine is the phrase “Mealy is more and Moore is less.” This refers to the outputs, which in the Mealy machine can include unregistered inputs.

Example Read-Write Synchronizer FSM

The following examples refer to this FSM specification:

- If `do_write` is true, transition to `WRITE` and set `exec = 0` and `rd_wr = 1`.
 - When `do_write` is again true, transition to `DONE` and set `exec = 1`.
- If `do_write` is false, transition to `READ` and set `exec = 0` and `rd_wr = 0`.
 - When `do_write` is again false, transition to `DONE` and set `exec = 1`.



```
typedef enum logic [1:0] {IDLE, READ, WRITE, DONE} state_t;
```

416 © Cadence Design Systems, Inc. All rights reserved.

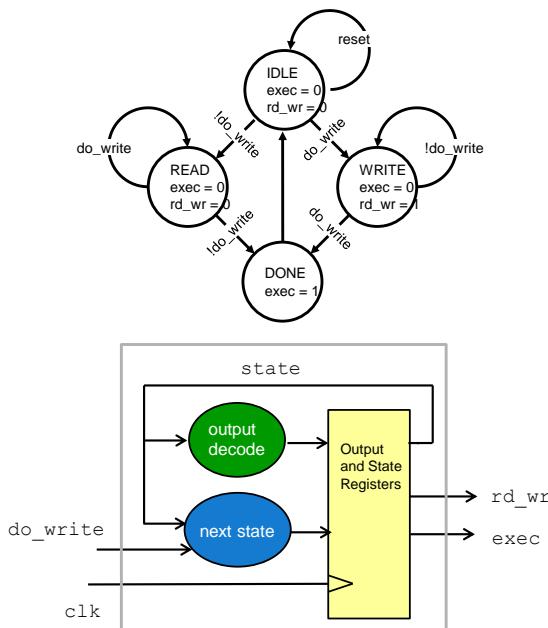


The following examples refer to this state machine specification:

- If `do_write` is true, it transitions to the `WRITE` state and sets `exec` to 0 and `rd_wr` to 1. When `do_write` is again true, it transitions to the `DONE` state and sets `exec` to 1.
- If `do_write` is false, it transitions to the `READ` state and sets `exec` to 0 and `rd_wr` to 0. When `do_write` is again false, it transitions to the `DONE` state and sets `exec` to 1.

The FSM is synchronously reset to the `IDLE` state.

Coding the FSM in One Block: Registered Outputs



```

typedef enum logic [1:0] {IDLE, READ, WRITE, DONE} state_t;
state_t state; State variable

always_ff @ (posedge clock)
  if (reset) begin
    state <= IDLE; Reset state and outputs
    {exec, rd_wr} <= 2'b00;
  end
  else begin
    {exec, rd_wr} <= 2'b00; Default output assignments simplify code
    case ( state )
      IDLE: begin
        state <= do_write ? WRITE : READ;
        rd_wr <= do_write;
      end
      READ: if ( !do_write ) begin
        state <= DONE;
        exec <= 1'b1;
      end
      WRITE: if ( do_write ) begin
        state <= DONE;
        exec <= 1'b1;
      end
      DONE: state <= IDLE;
    endcase
  end

```

Case for next state and output decode

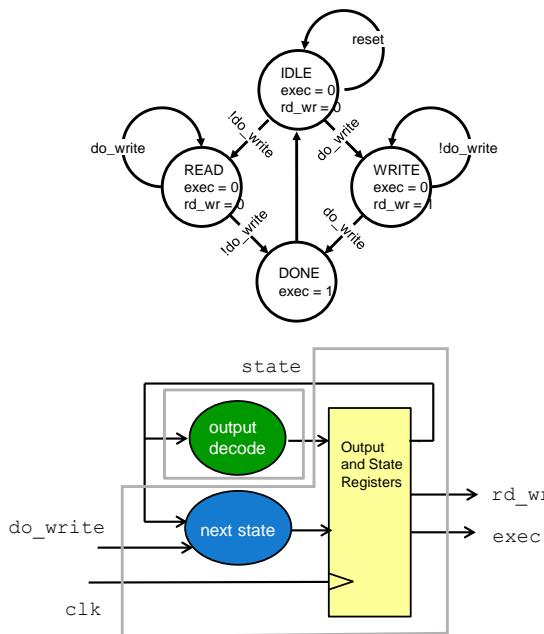
417 © Cadence Design Systems, Inc. All rights reserved.

cadence®

The one-block coding style generates the next-state, state, and outputs all in one sequential block.

The reset is synchronous and resets both the state variable to IDLE and registered outputs to 0.

Coding the FSM in Two Blocks: Registered Outputs



```

State and next state variables
typedef enum logic [1:0] {IDLE, READ, WRITE, DONE} state_t;
state_t state, nstate;

Case for combinational next state
always_comb
case ( state )
    IDLE: nstate = do_write ? WRITE : READ;
    READ: nstate = !do_write ? DONE : READ;
    WRITE: nstate = do_write ? DONE : WRITE;
    DONE: nstate = IDLE;
endcase

Case for registered output decode
always_ff @(posedge clock)
if (reset) begin
    state <= IDLE;
    {rd_wr, exec} <= 2'b00
end
else begin
    state <= nstate;
    case ( nstate )
        IDLE: {rd_wr, exec} <= 2'b00;
        READ: {rd_wr, exec} <= 2'b00;
        WRITE: {rd_wr, exec} <= 2'b10;
        DONE: {rd_wr, exec} <= 2'b01;
    endcase
end

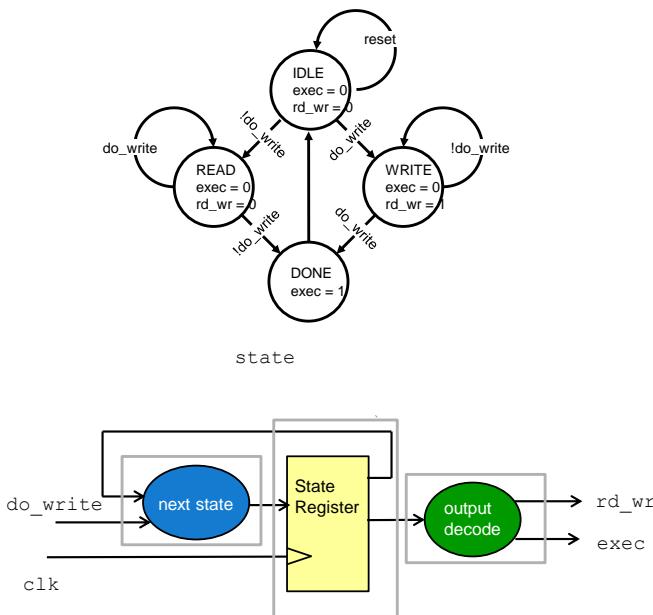
```

418 © Cadence Design Systems, Inc. All rights reserved.



The two-block coding style encodes the next state logic in a separate combinational block. We now need two variables for the state value. `state` holds the current FSM state and `nstate` is used to calculate the next state in the combinational logic block.

Coding the FSM in Three Blocks: Combinational Outputs



```

typedef enum logic [1:0]
    {IDLE, READ, WRITE, DONE} state_t;
state_t state, nstate;

always_comb
    case ( state )
        IDLE: nstate = do_write ? WRITE : READ;
        READ: nstate = !do_write ? DONE : READ;
        WRITE: nstate = do_write ? DONE : WRITE;
        DONE: nstate = IDLE;
    endcase

always_ff @(posedge clock)
    if (reset)
        state <= IDLE;
    else
        state <= nstate;

always_comb
    case ( nstate )
        IDLE: {rd_wr, exec} = 2'b00;
        READ: {rd_wr, exec} = 2'b00;
        WRITE: {rd_wr, exec} = 2'b10;
        DONE: {rd_wr, exec} = 2'b01;
    endcase

```

Combinational next state

Sequential state assignment

Combinational output decode

419 © Cadence Design Systems, Inc. All rights reserved.

cadence®

This is the classic 3-block definition of an FSM.

- A combinational block calculates the next state from the current state and inputs.
- A sequential block assigns next state to current state, and also resets the state variable.
- A combinational block assigns the FSM outputs from the current state.

State Machine Optimization

State encoding can be optimized for:

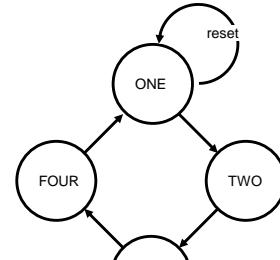
- Reduced next state or output decode logic.
- Reduced register count.
- Power efficiency/noise in state transitions.

Encoding can be defined:

- In the SystemVerilog enumerate declaration.
- With Synthesis tool options.

```
typedef enum logic [3:0] {ONE = 4'b0001,
TWO = 4'b0010,
THREE = 4'b0100,
FOUR = 4'b100} state_t;
```

One-hot encoding



State/ Encoding	ONE	TWO	THREE	FOUR	registers	register count	state/output logic*	Comments
binary	00	01	10	11	2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Default
Gray	00	01	11	10	2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	Fewest bit changes for predictable state transition patterns.
one-hot	0001	0010	0100	1000	4	<input type="checkbox"/>	<input checked="" type="checkbox"/>	Good compromise for unpredictable state patterns. High register count.

420 © Cadence Design Systems, Inc. All rights reserved.



A key optimization technique for synthesizing an FSM is to refine the state encodings. Different state encodings can reduce the size, and hence propagation delay, of the next state or output decode logic. You could reduce the number of registers used for the state variable. Different encodings can also help with power efficiency and noise by reducing the number of bits that change in transitions from state to state. Fewer bits changing is more power efficient, and also better for noise and glitching. For example, a state transition from the encoding $2'b00$ to $2'b11$, as seen by the outputs, may actually be $2'b00 \rightarrow 2'b01 \rightarrow 2'b11$ (or any similar combination) rather than a clean $2'b00 \rightarrow 2'b11$. This may generate glitches on the outputs for pure combinational outputs or a Mealy FSM.

Gray encoding tries to minimize bit changes in state transitions, but is only effective for predictable state transitions, as in the example where one state can only transition to one other state. For a state machine where a state can transition to a number of different states, gray coding is much harder to implement.

One-hot encoding can be a good compromise for a more complex state diagram. Each state is encoded with only 1 bit set, therefore, the transition from any one state to any other state will only involve 2 changed bits. However, the number of registers is much higher (1 register per state) although this can be an advantage in register-rich technologies such as FPGAs.

Submodule Summary

You should now be able to code state machines for synthesis.

This submodule discussed:

- The three parts of an FSM:
 - State register, next state logic, and output decode.
- How to define FSM states:
 - Using enumerates.
- Various ways to code FSMs.
- 1-block, 2-block, 3-block.
- Optimization of FSMs using state vector encoding.



You should now be able to code state machines for synthesis.

This module discussed what an FSM is, how to define FSM states, various ways to code FSMs for synthesis, and various ways to optimize FSMs.

Test Your Understanding

1. What is the difference between a Moore machine and a Mealy machine?
2. How should you define state values for an FSM?
3. What are the advantages and disadvantages of one-hot state vector encoding?



This page does not contain notes.

Solutions: Test Your Understanding

1. What is the difference between a Moore machine and a Mealy machine?
 - The outputs of a Moore machine are a function purely of the FSM state. The outputs of a Mealy machine are a function of both the FSM state and the FSM inputs.
2. How should you define state values for an FSM?
 - Using an enumerated type.
3. What are the advantages and disadvantages of one-hot state vector encoding?
 - One-hot encoding is generally efficient for next-state and output decode logic, but has a higher register count than other encodings. It is a good compromise for low power/noise in unpredictable state patterns.

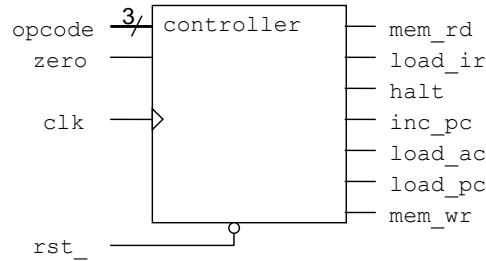


This page does not contain notes.

Lab

Lab 7 Modeling a Sequence Controller

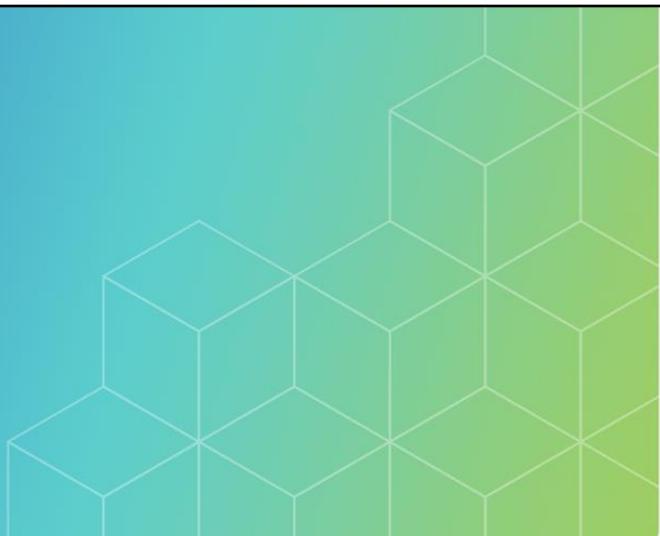
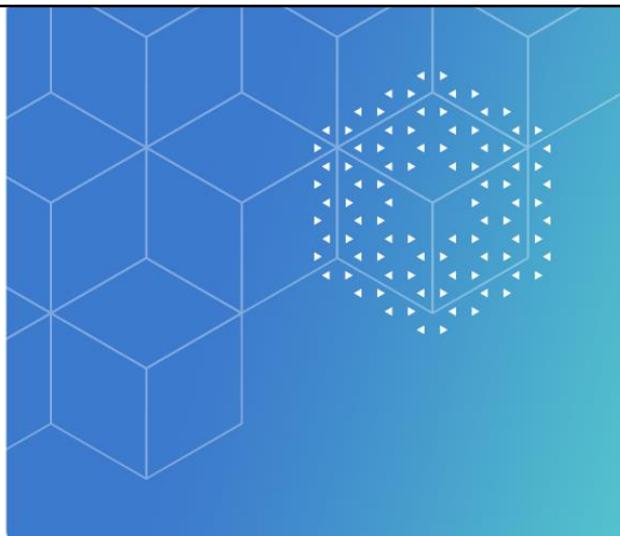
- Using enumerate types, procedural statements and operators to model a state machine.



424 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Module 5

Functional Verification

cadence®

This module spans 4 hours of lecture time.

Module Objectives

In this module, you will:

- Define functional verification.
- Write a simple SV testbench using the FSM example of a drink machine.
- Debug using the waveform viewer, breakpoints and Xcelium simulator tool.
- Recognize the three cornerstones of modern verification flow: randomization, coverage, and assertions and why we need them.
- Define constraint-based randomization.
- Discuss coverage and other metrics (code coverage, functional coverage, SV cover groups).
- Define assertion-based verification (assertions and covers).
- Identify class-based testbenches, UVM and methodology.
- Discuss verification planning, the concept of executable and dynamic verification plan and management, and mapping with metrics and coverage to the vPlan.
- Recognize the MDV Methodology and its different phases.
- Recognize formal verification and its use models
 - Exhaustive proof, Automated Formal Checks, Design Exploration, Equivalence Checking and many others.
- Identify Accelerated Verification (Emulation, Prototyping and multicore simulation).
- Identify the verification completeness problem – How do we know we are done?
- Identify the verification management aspect, that is, “Do we know why it isn’t tested?” (bug rate, coverage, risk management)
- Identify the verification challenges and the process decisions made before tapeout.

426 © Cadence Design Systems, Inc. All rights reserved.



In this module, you will:

- Define functional verification.
- Write a simple SV testbench using the FSM example of a drink machine.
- Debug using the waveform viewer, breakpoints and Xcelium simulator tool.
- Recognize the three corner stones of modern verification flow: randomization, coverage, and assertions and why we need them.
- Define constraint-based randomization.
- Discuss coverage and other metrics (code coverage, functional coverage, SV cover groups).
- Define assertion-based verification (assertions and covers).
- Identify class-based testbenches, UVM and methodology.

You Also:

- Discuss verification planning, the concept of executable and dynamic verification plan and management, and mapping with metrics and coverage to the vPlan.
- Recognize the MDV Methodology and its different phases.
- Recognize formal verification and its use models
 - Exhaustive proof, Automated Formal Checks, Design Exploration, Equivalence Checking and many others.
- Identify Accelerated Verification (Emulation, Prototyping and multicore simulation).
- Identify the verification completeness problem – How do we know we are done?
- Identify the verification management aspect that is, “Do we know why it isn’t tested?” (bug rate, coverage, risk management)
- Identify the verification challenges and the process decisions made before tapeout.

What Is Functional Verification?

- Verification is the process, a continuous one at that, which runs throughout the design flow alongside each major phase.
- It ensures the functionality, timing, and integrity of the changing design throughout the process.
- It ensures a design's implementation meets the design specification requirements.
- Goal of verification:
 - Demonstrate the functional correctness of a design.
 - Attempt to find design errors.
 - Attempt to show that the design implements the specification.
- Which design characteristics are being verified:
 - Functionality
 - Does the design model perform all the operations it is required to?
 - Does the design model perform these operations correctly?
 - Timing
 - Does the design model meet the timing (performance) requirements required by the design specification?



Functional verification is the process that makes sure the coded design is functioning the right way according to the specification it followed for coding.

Verification is the process, a continuous one at that, which runs throughout the design flow alongside each major phase.

It ensures the functionality, timing, and integrity of the changing design throughout the process.

It ensures a design's implementation meets the design specification requirements.

The goal of verification is to:

- Demonstrate the functional correctness of a design.
- Attempt to find design errors.
- Attempt to show that the design implements the specification.

Now, what are the characteristics that are being verified during the verification process:

It verifies the functionality, as in:

- Is the design model performs all the operations it is required to?
- Is the design model performing these operations correctly?

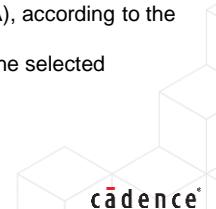
It also checks the timing, that is, to see if the design model meets the timing as well as performance requirements required by the design specification.

Verification Methods

There are three methods to perform verification:

1. Simulation (dynamic)
 - Predominant verification method.
 - Only option for behavioral-level verification.
 - Involves developing testbenches and input vectors.
 - Harder to determine whether or not a verification suite completely checks all corner cases or not.
2. Formal (Static)
 - This derives from `formal` logic
 - Reasoning based on manipulation of formulas, hence static
 - The name `formal` can mean many things, e.g., any of the following:
 - Logic equivalency checking (LEC)
 - Model Checking – Functional formal verification (Formal Analysis)
 - Theorem proving
3. Emulation
 - A design is compiled for different modes, namely In-Circuit Emulation (ICE) and Simulation Acceleration (SA), according to the requirements.
 - After compilation, the compile result is imported into the run-time interface to run and debug the design on the selected emulator system.
 - Execute and debug RTL and software on custom processor hardware 1000's times faster than software simulator.

428 © Cadence Design Systems, Inc. All rights reserved.

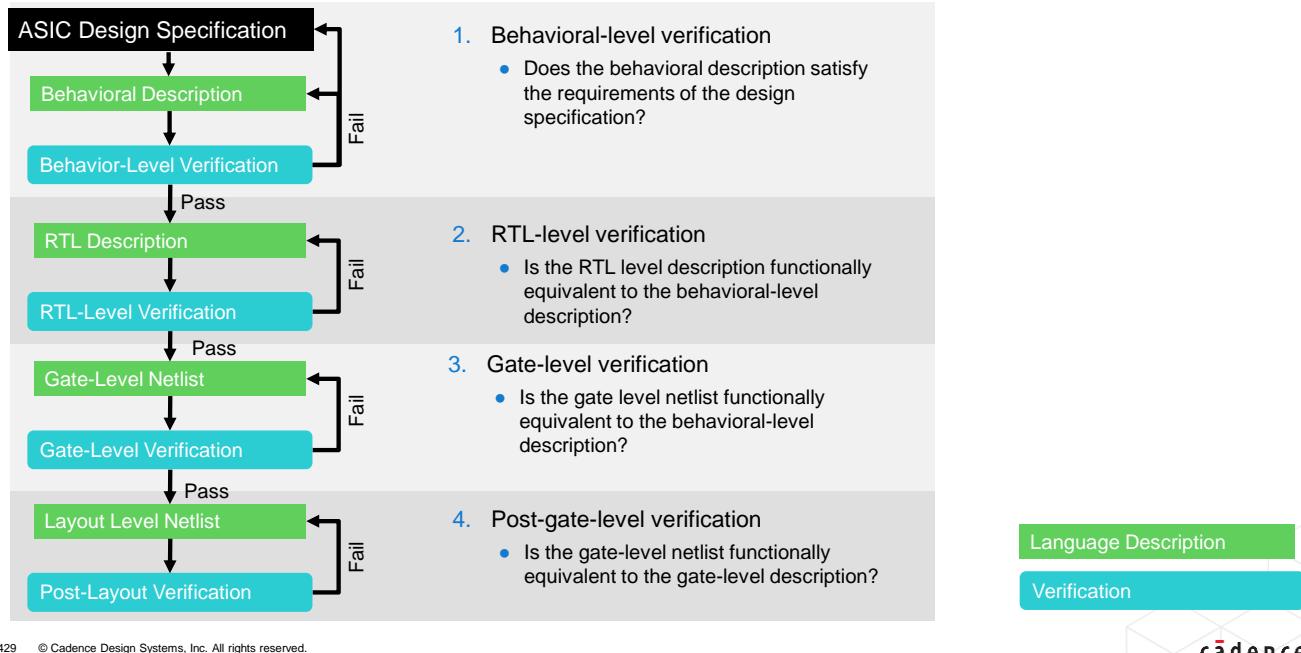


This slide goes over the different methods of verification.

There are three methods followed:

1. The Simulation which is dynamic in nature
 - It is a Predominant verification method.
 - This is the Only option for behavioral-level verification.
 - It Involves developing testbenches and input vectors.
 - It is Harder to determine whether or not a verification suite completely checks all corner cases or not.
2. The Formal which is static in nature
 - This derives from `formal` logic.
 - Its Reasoning is based on manipulation of formulas, hence static.
 - The name `formal` can mean many things, e.g., any of the methods such as:
 - Logic equivalency checking (LEC)
 - Model Checking – Functional formal verification (Formal Analysis)
 - Or Theorem proving.
3. The third method is Emulation
 - In here the design is compiled for different modes, namely In-Circuit Emulation (ICE) and Simulation Acceleration (SA) according to the requirements.
 - After compilation, the compiled result is imported into the run-time interface to run and debug the design on the selected emulator system.
 - Then the RTL is executed and debugged along with software, on custom processor hardware 1000 times faster than software simulator.

Verification of Four Different Abstraction Levels



429 © Cadence Design Systems, Inc. All rights reserved.



We went over the different levels of abstraction on the design coding domain, and now we look at what we verify in each of these levels. This slide shows a flow diagram of an entire ASIC design specification process starting with behavioral phase till post gate level phase. And, in here we discuss what we verify in each phase.

So now, let's go over what we covered as to the coding aspects in each phase and then see what aspects we verify in each phase.

- At The behavioral phase:

- You describe the system using mathematical equations.
- You can omit timing – the system may simulate in zero time, like a software program.

At this phase, for the Behavioral level verification:

- You check if the behavioral description satisfies the requirements of the design specification?

- At The Register Transfer Level (RTL):

- You partition the system into combinational and sequential logic, using constructs and coding styles supported by logic synthesis.
- You define timing in terms of cycles based on one or more defined clocks.

At this phase, for the RTL level Verification:

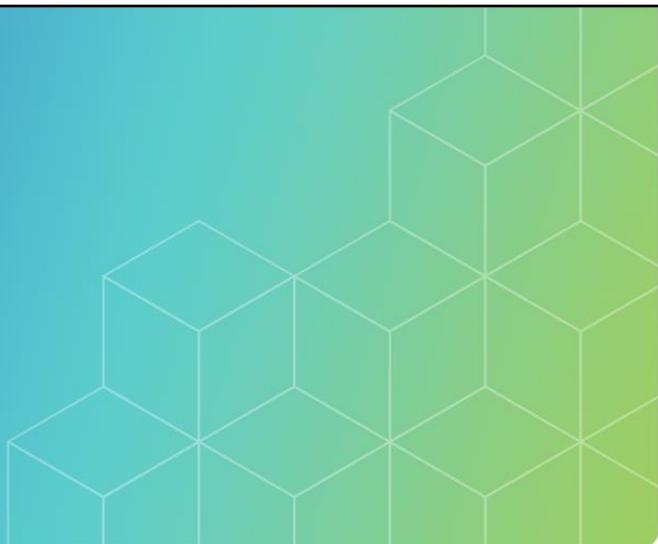
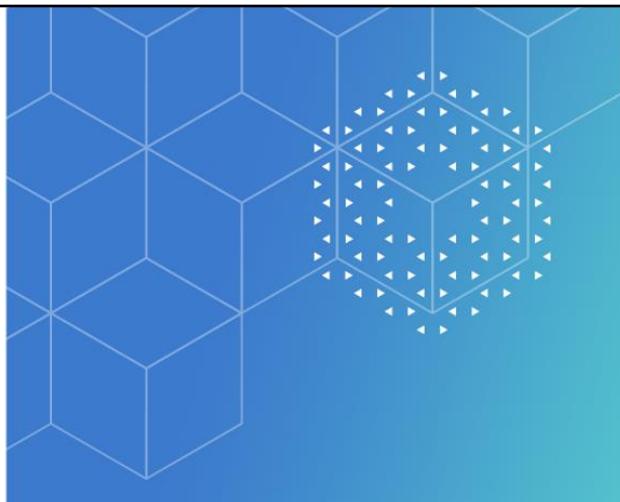
- You check if the gate level netlist functionally is equivalent to the behavioral-level description?
- This is also considered as Gate level Verification

- At The structural level or gate level description:

- You instantiate and interconnect predefined components.
- You Can include vendor-provided macrocells.
- You Can include logic primitives built into the language.

At this phase, for the verification which is called as post gate level verification:

- You check if the gate-level netlist is functionally equivalent to the gate-level description?



Submodule 5-1

Debug Techniques

cadence®

This module talks about the interactive and post processing debug techniques.

Submodule Objectives

In this submodule, you will:

- List the different kinds of debug techniques.
- Debug an error using the SimVision™ GUI with relevant textual commands using the following steps:
 - Explore the drink machine SV design.
 - Execute the single-step *xrun* command to compile, elaborate and simulate the design.
 - Examine the error that displays.
 - View the error in the waveform as well as in the source window.
 - Analyze the method of correcting this error.
 - Fix the source code by opening the editor and re-compiling the code.
 - Re-simulate the design.
 - Examine the various textual commands used in the different stages listed above.



In this submodule, you will:

- List the different kinds of debug techniques.
- Debug an error using the SimVision™ GUI with relevant textual commands using the following steps:
- Explore the drink machine SV design.
- Execute the single-step *xrun* command to compile, elaborate and simulate the design.
- Examine the error that displays.
- View the error in the waveform as well as in the source window.
- Analyze the method of correcting this error.
- Fix the source code by opening the editor and re-compiling the code.
- Re-simulate the design.
- Examine the various textual commands used in the different stages listed above.

What Are Debug Techniques?

Debug is to rectify or resolve an error in the simulation and then accordingly rerun with the fix and complete the simulation process.

We have two kinds of debug techniques for a simulation tool.

- 1. Interactive Debug:** Uses single stepping and break points.
- 2. Post Processing Debug:** Done after you run the simulation.



So lets understand what is debug and the debug technique.

Debug is to rectify or resolve an error in the simulation and then accordingly rerun with the fix and complete the simulation process.

We have two kinds of debug techniques for a simulation tool.

- 1. Interactive Debug** which Uses single stepping and break points.
- 2. And Post Processing Debug** which is Done after you run the simulation.

Interactive Debug Process

This is the debug process performed while you are running your simulation through single stepping and breakpoints.

1. Setting breakpoints.
 - Most common types of breakpoints:
 - Time-based – Stop the simulation on a specific line of code if we have reached a certain time.
 - Value-based – Stop the simulation on a specific line if ever a specific set of signals take on the desired value.
2. Single stepping allows users to run code in a controlled manner:
 - Access to all simulator data is a huge productivity boost for debug.
 - Can explore on the fly – Can examine the context of every line executing.
 - Can examine all variable values at the current simulation time.
 - Can selectively probe items of interest on the fly.
 - Step within the current thread, within all threads, into methods.

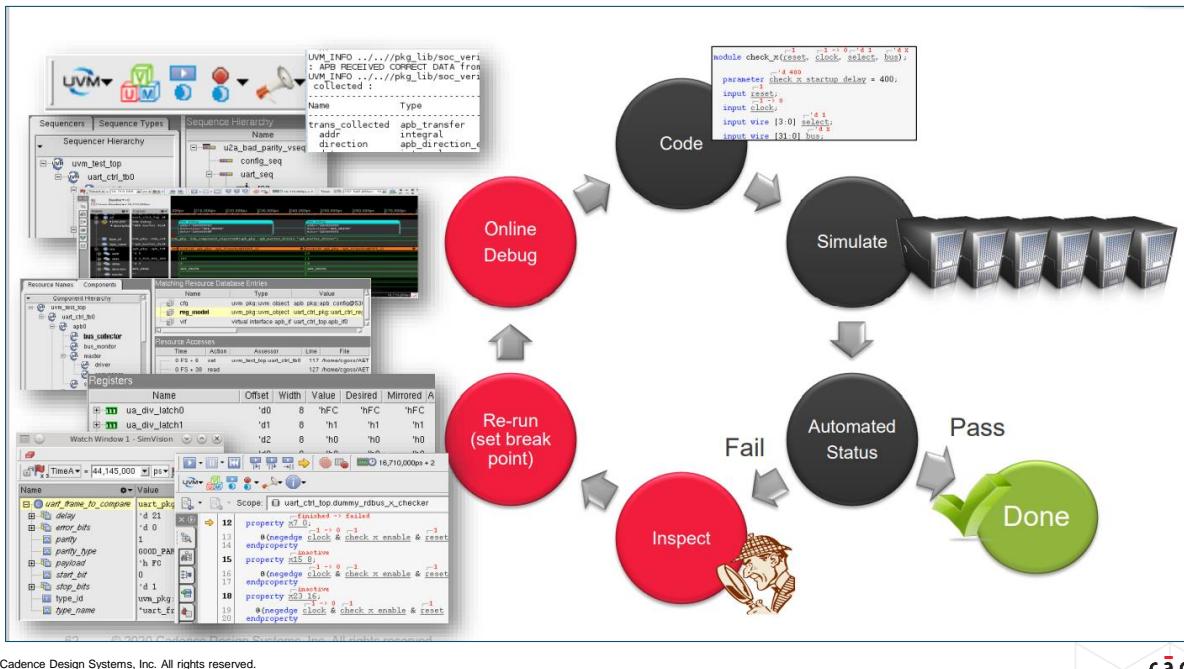


Let's understand the interactive debug process.

This is the debug process performed while you are running your simulation.

- Through single stepping and breakpoints.
- Let's talk about setting breakpoints.
- Most common types of breakpoints are:
 - Time-based – where you stop the simulation on a specific line of code if we have reached a certain time.
 - Value-based – where you stop the simulation on a specific line if ever a specific set of signals take on the desired value.
- Now, about the single-stepping process.
- Single stepping allows users to run code in a controlled manner:
 - Since it can access to all simulator data, it is a huge productivity boost for debugging.
 - Here, you can explore on the fly – you can examine the context of every line executing.
 - You can examine all variable values at the current simulation time.
 - You can selectively probe items of interest on the fly.
 - And step within the current thread, within all threads, into methods.

1. Interactive Debug Process (continued)



434 © Cadence Design Systems, Inc. All rights reserved.

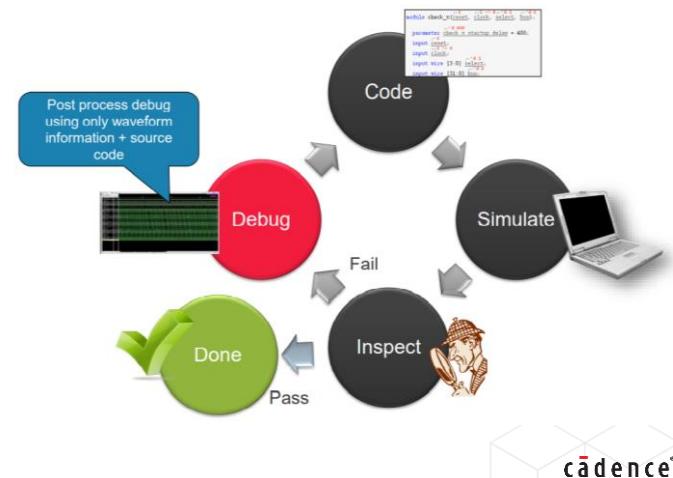
cadence®

This slide shows the snapshots for interactive debug where you apply break points and try to pause simulation and debug and then continue as needed and repeat the iteration if required.

What Is Post Processing Debug?

This is the debug process done after you run the simulation. You open the waveform in your designated debug tool (SimVision) and then pull out all the simulation signals and objects and perform debugging.

- Post Processing Mode (inspection of HDL/HVL Waveforms)
- Waveform signal comparison (using simcompare)
- Driver tracing
- Probing assertions
- Probing HVL objects (transaction recording)
- Probing UVM hierarchy
- Sequence debug



435 © Cadence Design Systems, Inc. All rights reserved.

This slide talks about what is post processing debug.

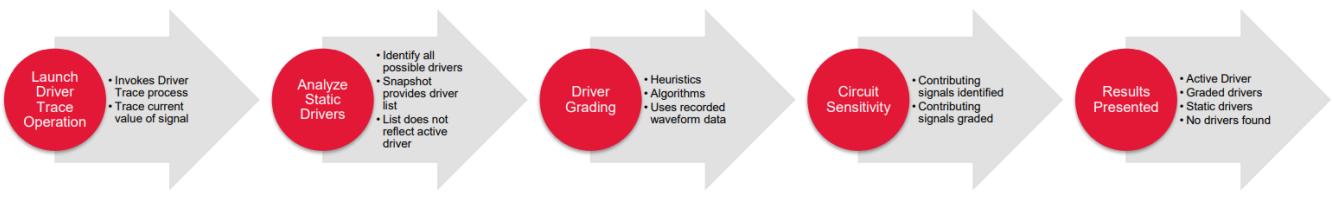
This is the debug process done after you run the simulation. You open the waveform in your designated debug tool like SimVision or Indago™, and then pull out all the simulation signals and objects and perform debug

In here we use:

- Post Processing Mode that is the inspection of HDL/HVL Waveforms)
- Waveform signal comparison (using simcompare)
- Driver tracing
- Probing assertions
- Probing HVL objects (transaction recording)
- Probing UVM hierarchy
- Sequence debug

Post Processing Debug Mode

- Waveform signal comparison using the (Sim)Compare Manager in the (SimVision) debug tool.
- Driver tracing
 - There are several steps that take place under the hood when users perform a driver trace operation within the debug tool:



- Probe assertions in several different ways

- If assertions are present in an already probed module:
 - Assertion state as well as checked_count, finish_count, failure_count and disabled_count counters probed.
 - Contributing signals can be shown (if recorded and if the snapshot is loaded).

436 © Cadence Design Systems, Inc. All rights reserved.



The different sub-processes or the steps that constitute a Post Processing Debug Mode are mentioned here.

Step 1 is a Waveform signal comparison using the (Sim)Compare Manager in the (SimVision) debug tool.

Step 2 is a Driver Tracing which consists of several substeps, which are:

- Invoke the driver.
- Identify all possible drivers. Use driver grading with heuristics and algorithms
- Identify the contributing signals and present the results

Step 3 is probing assertions in several different ways.

If assertions are present in an already probed module, then:

- Assertion state as well as checked-count, finish-count, failure-count, and disabled-count counters are probed.
- And contributing signals can be shown.

Post Processing Debug Mode (continued)

Probing HVL objects allow for debug at several levels of abstraction

- HDL signal level
- Transaction level
- Testbench level

Transaction recording is built in for UVM sequences

- No need to use the Cadence UVM library
- Must be enabled
- Easiest way to enable transaction recording for all objects

Probing UVM hierarchy

- Probing the entire UVM hierarchy
- Probing only a single UVC
- Users must request UVM base class fields via the –uvm switch



Continuing more info on Post Processing Debug Mode:

Probing HVL objects allow for debug at several levels of abstraction like at:

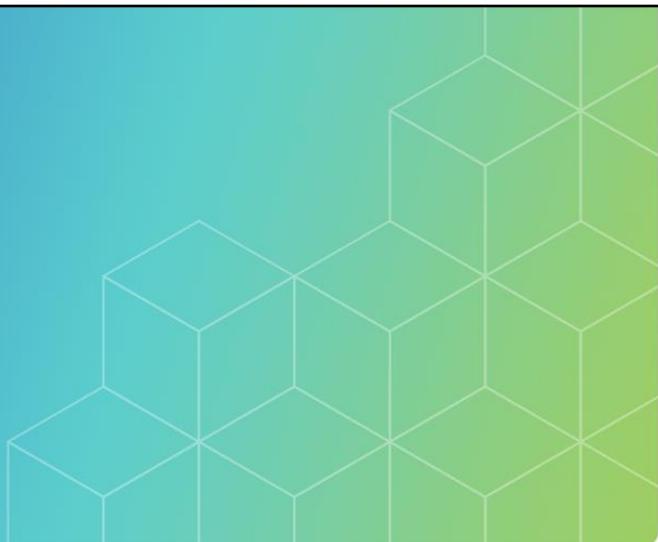
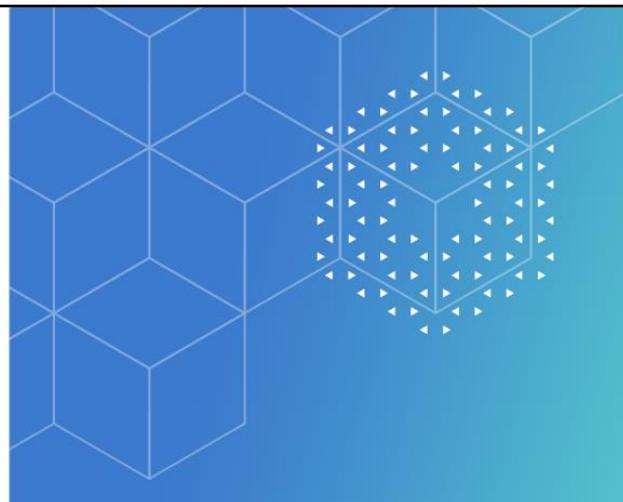
- HDL signal level
- Transaction level
- And at the testbench level

Transaction recording is built in for UVM sequences.

- No need to use the Cadence UVM library
- They must be enabled
- The easiest way to enable transaction recording for all objects

For probing UVM hierarchy, we can have different types like:

- Probing the entire UVM hierarchy
- Probing only a single UVC
- And users must request UVM base class fields via the –uvm switch



Submodule 5-1-1

Debugging Using Interactive Debug with Xcelium™: GUI and Textual/Batch Commands

cadence®

Debugging with Xcelium using both GUI and textual or batch commands.

Submodule Objective

In this submodule, you will:

- Debug using the interactive debug technique.



In this submodule, you will debug using the interactive debug technique.

Scenario: Drink Machine Design

The drink machine example is a simple design and testbench used extensively throughout the Xcelium/SimVision documentation.

There is an error in the drink machine logic. It sometimes does not reset to the *idle* state when a drink is dispensed.

- All the source files for this design are included in the Cadence® installation hierarchy in the following directory:
 - *install_dir/doc/xsctutorial/examples*
- The design itself is made up of the following modules:
 - **Drink_machine**: Counts the amount of change that the user has entered, dispenses a drink, and returns any due change.
 - **Coin_counter**: Loads the machine with coins and determines when the device is out of change.
 - **Can_counter**: Loads the machine with drinks and determines when the machine is empty.
 - **State_machine**: Models the behavior for accepting coins and dispensing drinks:
 - Amount of money that the user has deposited so far defines the current state.
 - The type of coin the user deposits determines the machine's next state.



This page does not contain notes.

Scenario: Drink Machine State Table

A drink costs 50 cents.

The machine accepts:

- Nickels (5 cents)
- Dimes (10 cents)
- Quarters (25 cents)

The FSM must count coins until the cost is met or exceeded

- Then dispense a can...
- ...and return any change

The machine also keeps track of:

- Number of stored coins for change
- Number of cans

Current State	Transition Value	Next State (Change)
idle 4'd0	nickel_in dime_in quarter_in	five ten twenty_five
five 4'd1	nickel_in dime_in quarter_in	ten fifteen thirty
ten 4'd2	nickel_in dime_in quarter_in	fifteen twenty thirty_five
fifteen 4'd3	nickel_in dime_in quarter_in	twenty twenty_five forty
twenty 4'd4	nickel_in dime_in quarter_in	twenty_five thirty forty_five
twenty_five 4'd5	nickel_in dime_in quarter_in	thirty thirty_five idle
thirty 4'd6	nickel_in dime_in quarter_in	thirty_five forty idle (nickel_out)
thirty_five 4'd7	nickel_in dime_in quarter_in	forty forty_five idle (dime_out)
forty 4'd8	nickel_in dime_in quarter_in	forty_five idle idle (nickel_out, dime_out)
forty_five 4'd9	nickel_in dime_in quarter_in	idle idle (nickel_out) idle (two_dime_out)

441 © Cadence Design Systems, Inc. All rights reserved.



The table shows the various states the drink machine can go through.

For example, when no money has been deposited, the machine is in an *idle* state. When the user adds a nickel, the machine transitions to the next state, *five*. When the current state is *five*, and the user adds a quarter, the machine transitions to the next state *thirty*. When the user adds 50 cents, the machine dispenses a drink and transitions back to the *idle* state. When the user adds more than 50 cents, the machine dispenses a drink, returns the correct change, and transitions back to the *idle* state.

Summary of the Problem

- **Problem**

- There is an error in the drink machine logic. It sometimes does not reset to the **IDLE** state when a drink is dispensed.

- **Given**

- Drink machine design and the testbench.
 - SimVision tool and Xcelium simulator.

- **Goal**

- To use your preferred text editor to fix this error.
 - To rerun the simulation to ensure that the behavior is correct.



This page does not contain notes.

Possible Solutions

	Strategy	Pros	Cons
1.	Use Debug GUI	More visual, easy to view the waveform to figure out the error	Takes time to load
2.	Use Debug batch commands	Quicker and efficient time-wise	Visual is lacking

We shall primarily follow the SimVision GUI flow.

At the same time, show the textual batch commands that can be used to achieve the same.

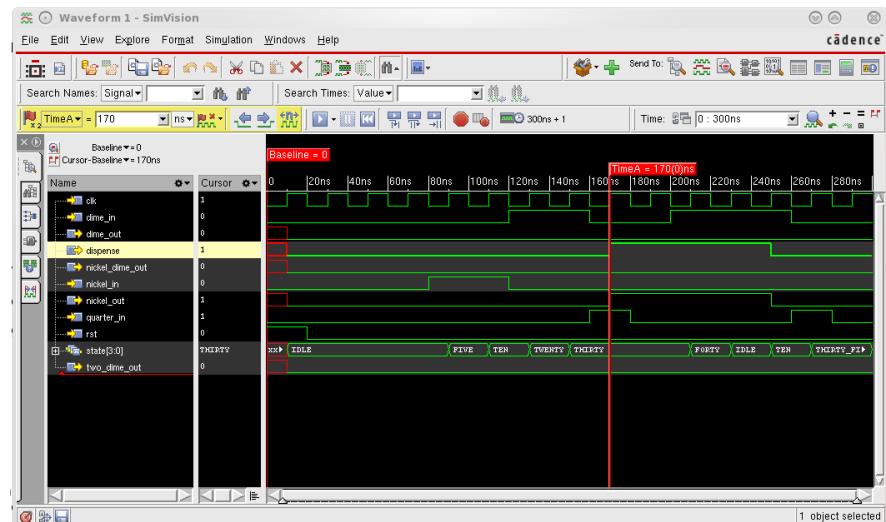
Apart from the SimVision tool, the Cadence Indago™ Debug GUI tool could also be used for this Debugging Task.



This page does not contain notes.

How to Fix the Problem

- Use the waveform window to show how the values of signals change during simulation and locate where the error occurs.
- From the waveform window, go to the cause of the signal transition in the source browser.
- In the source browser, find this logic error.



444 © Cadence Design Systems, Inc. All rights reserved.

cadence®

This page does not contain notes.

Starting the Simulator Run Using the **xrun** Command

```
$ ls  
dkm.sv  dkm_test.sv  run.f  
$ xrun -f run.f
```

```
// run file for FSM  
// files to be compiled  
dkm.sv  
dkm_test.sv  
  
// command line options  
-gui  
-access rwc  
-linedebug  
-timescale 1ns/1ns
```

It is easiest to invoke Xcelium using a run file.

The command-line options are:

- **-access rwc** allows read, write, and connectivity access to the design objects. Required to probe signals and display their waveforms in SimVision.
- **-gui** invokes the SimVision Graphical User Interface.
- **-linedebug** enables support for setting line breakpoints and single-stepping through the code.
- **-timescale 1ns/1ns** sets the default time unit and precision information for the simulator.



This page does not contain notes.

The Terminal Displays the *xcelium* Prompt

```

        module worklib.dkm:sv
            errors: 0, warnings: 0
file: dkm_test.sv
        module worklib.dkm_test:sv
            errors: 0, warnings: 0
                Caching library 'worklib' ..... Done
                Elaborating the design hierarchy:
                Top level design units:
                    dkm:sv
xmLab: *W,DSEMEL: This SystemVerilog design will be simulated as per IEEE 1800-2009 SystemVerilog simulation semantics. Use
tics: Building instance overlay tables: ..... Done
Generating native compiled code:
    worklib.state_machine:sv <0x2df9ba12>
        streams: 55, words: 15263
    worklib.can_counter:sv <0x2fc2d616>
        streams: 5, words: 1704
    worklib.coin_counter:sv <0x6778750d>
        streams: 20, words: 7580
    worklib.dkm:sv <0x6ba8a9a6>
        streams: 12, words: 3904
    worklib.dkm_test:sv <0x48b69bdb>
        streams: 47, words: 24717
Building instance specific data structures.
Loading native compiled code: ..... Done
Design hierarchy summary:
    Instances Unique
    Modules:      5      5
    Registers:   41     41
    Scalar wires: 42     -
    Vectorized wires: 3     -
    Always blocks: 4     4
    Initial blocks: 1     1
    Cont. assignments: 2     2
    Pseudo assignments: 29    29
    Simulation timescale: 1ns
Writing initial simulation snapshot: worklib.dkm_test:sv
xmsim: *W,DSEM2009: This SystemVerilog design is simulated as per IEEE 1800-2009 SystemVerilog simulation semantics. Use -dis
.

-----
Relinquished control to SimVision...
xcelium>
xcelium> source /export/home/bdickins/cadence/install/XCELIUM1903/tools/xcelium/files/xmsimrc
xcelium> 
```

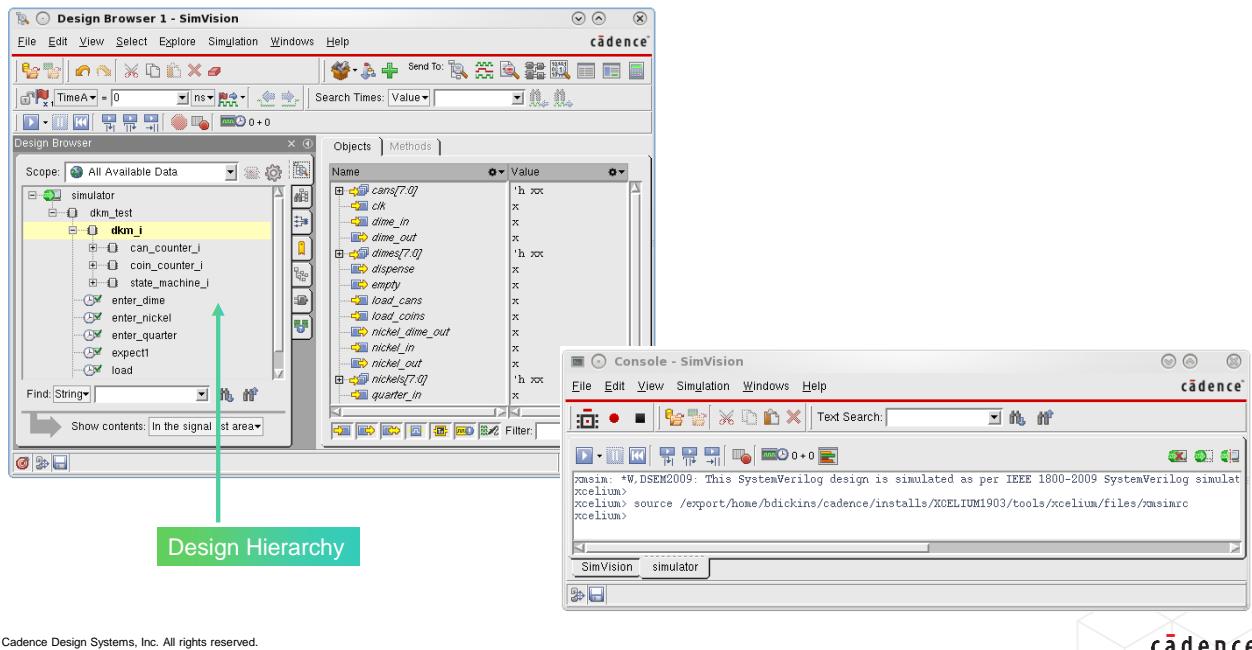
The terminal from which we invoke the *xrun* command also displays the *xcelium* prompt.

446 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Opening the Design Browser and Console Window at Startup



447 © Cadence Design Systems, Inc. All rights reserved.

cadence®

When SimVision starts, the Design Browser and the Console windows appear. You access your design hierarchy in the Design Browser and enter the SimVision and simulator commands in the Console window.

Adding Variables to the Waveform Window

Click down the hierarchy to the state_machine_i instance.

Right-click on state_machine_i.

Select Send to Waveform Window

448 © Cadence Design Systems, Inc. All rights reserved.



Click down the hierarchy to the state-machine-i instance.

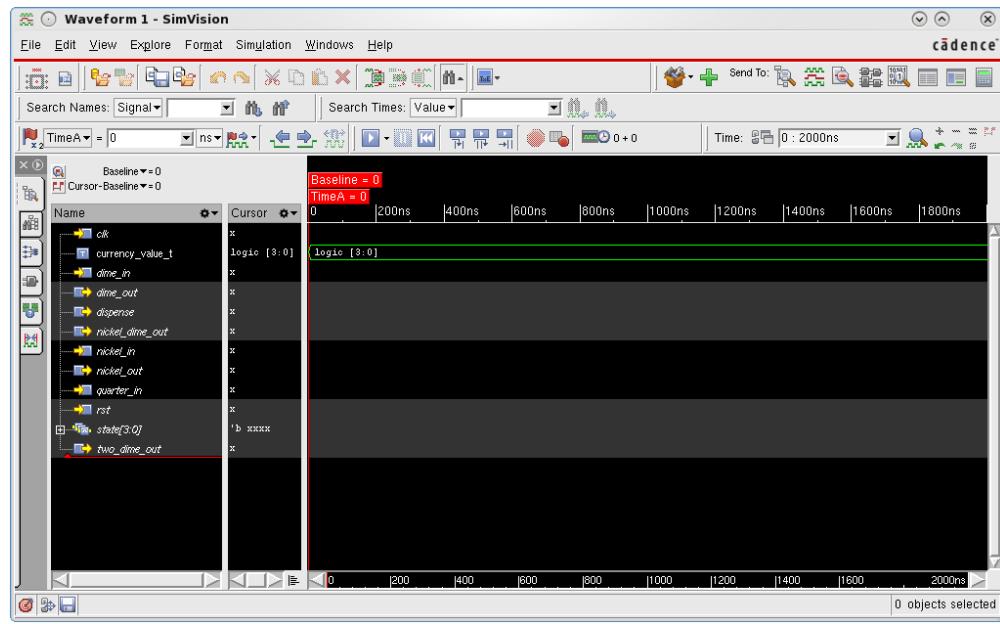
Right-click on the state-machine-I.

Select the **Send to Waveform** option.

Console Window Displaying Textual Commands

SimVision opens a Waveform window and adds all variables under state_machine_i.

Add variables to the waveform **before** running simulation.

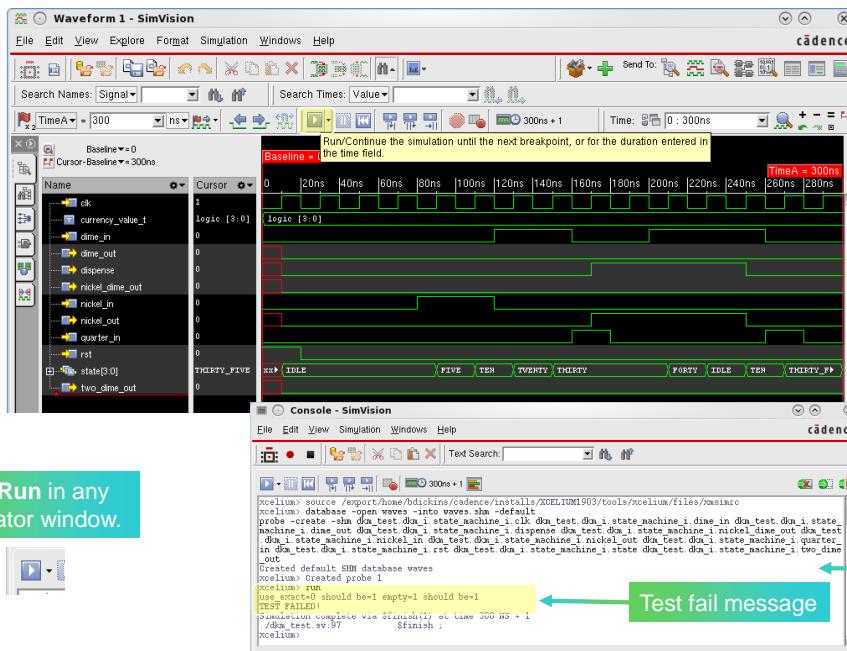


449 © Cadence Design Systems, Inc. All rights reserved.



SimVision displays the simulator messages in the Console window and the first command executed in the console window creates a simulation database and probes the signals that you added to the waveform window.

Running the Simulation



450 © Cadence Design Systems, Inc. All rights reserved.

SimVision runs the test until it reaches a breakpoint or until it reaches the end of the test, and it saves the simulation data to the database.

The Waveform window is also updated during the simulation to display the signal transitions.

Waveform Database Creation

```
$ ls  
dkm.sv dkm_test.sv run.f waves.shm xcelium.d xrun.history xrun.key xrun.log  
$ ls -hl waves.shm/  
total 8.0K  
-rw-r--r-- 1 ... 64K Oct 1 14:31 waves.dsn  
-rw-r--r-- 1 ... 828 Oct 1 14:31 waves.trn
```

1. When the simulation run has been completed, your working directory should contain a new directory named `waves.shm`.
2. The `waves.shm` dir should contain 2 files: `waves.dsn` and `waves.trn`.
3. If these files are significantly smaller than 64k and 828 bytes, respectively, you may not have probed all of the required variables or run the simulation to the end.



This page does not contain notes.

Adding More Relevant Signals to the Waveform Window

The screenshot shows two windows from Cadence Design Vision:

- Design Browser 1 - SimVision**: Shows a tree view of objects under "dkm_test". A context menu is open over the "dime_out" object, with the option "Send selected object(s) to target waveform window" highlighted.
- Waveform 1 - SimVision**: Shows waveforms for various signals over time. The signals include "dime_out", "nickel_in", "nickel_out", "quarter_in", "rst", and "two_dime_out". A cursor is positioned at 300ns.

Annotations provide instructions for adding variables to the waveform:

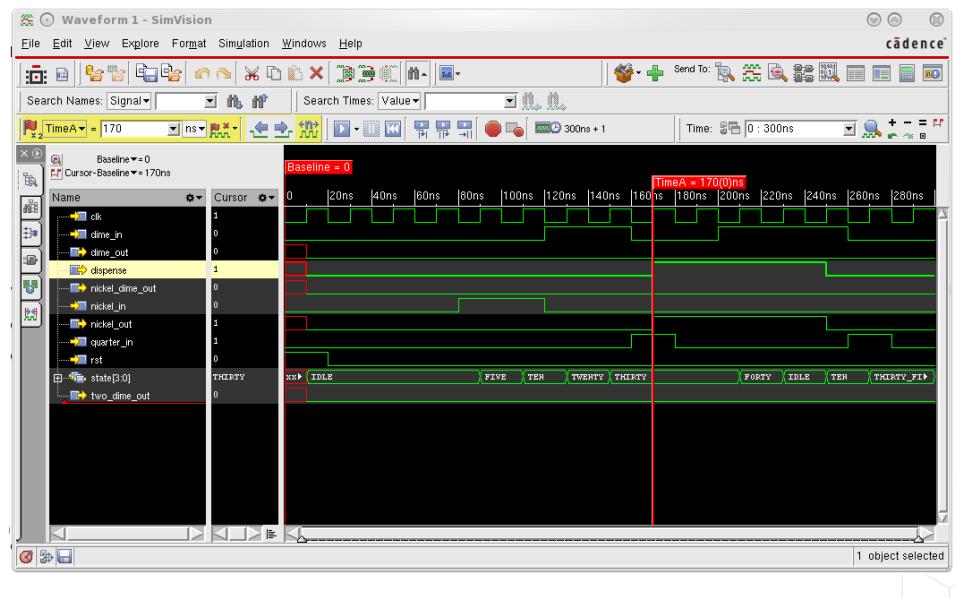
- A green callout points to the context menu in the Design Browser: "You can add more variables to the waveform by selecting and adding from the menu as before."
- A blue callout points to the "Send to Waveform" icon in the toolbar: "Alternatively, select the variable and then select the Send to Waveform icon."
- A green callout points to the "Run" and "Reset" buttons in the toolbar: "Reset and rerun the simulation to see waveforms for the newly added variables."

452 © Cadence Design Systems, Inc. All rights reserved.

This page does not contain notes.

Finding the Error in the Waveform Window

1. Move the Time cursor to the beginning of the simulation time by clicking at time **0** in the waveform.
2. Select **dispense**, then click **Next Edge** until the value of **dispense** is 1 at around 170ns.
3. Immediately prior to that time, **state** has the value **THIRTY**, and a quarter is added to the machine.
4. At the current cursor, **dispense** is 1, and the machine returns a nickel in change, but the state variable is not set to **IDLE**. Its value stays at **THIRTY** until the next coin is added, which is incorrect behavior.

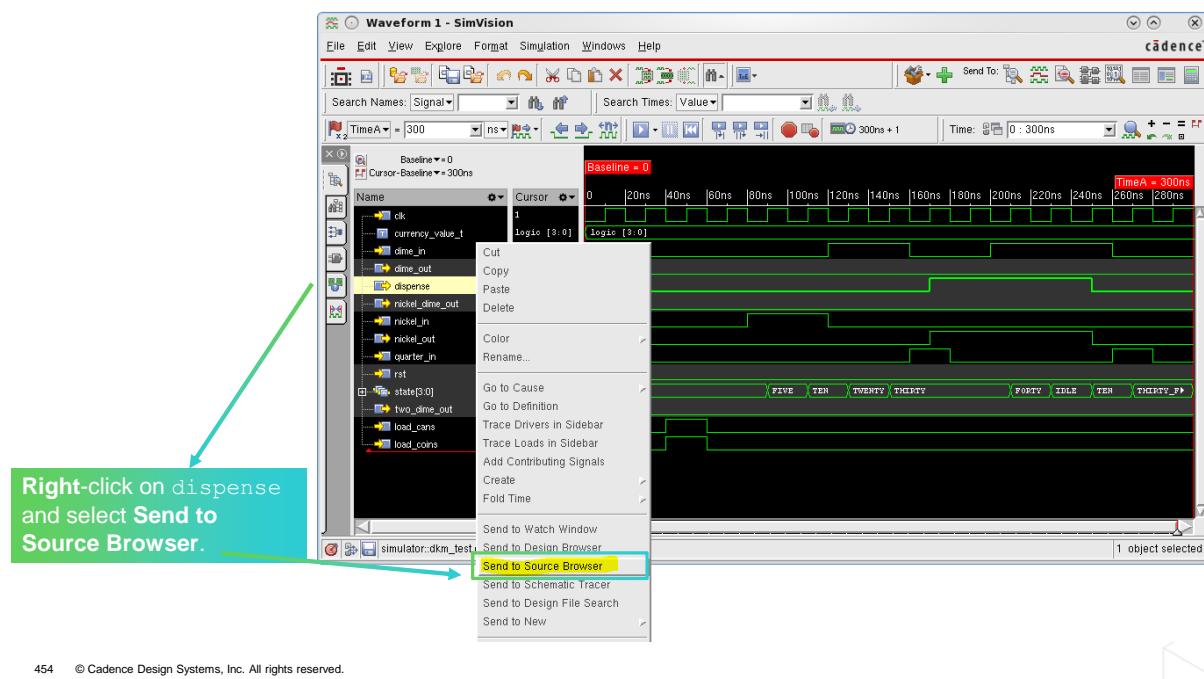


453 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Opening the Source Browser Window



This page does not contain notes.

Find the Cause of the Issue in the Source Browser

- Scroll down to the THIRTY state in the FSM.
 - Once a drink has been dispensed, the FSM state should go to IDLE.
 - So, the FSM can start counting coins for the next delivery.
 - The case statement for the THIRTY state does not set state back to IDLE after it dispenses a drink.

Source Browser 3 - SimVision [...]drinks_fsm/debug_module/dkm.sv

File Edit View Select Format Simulation Windows Help

Send To:

TimeA = 170 ns 300ns+1

Scope: dkm_test.dkm_i.state_machine_i Files: /export/home/.../dkm.sv [133:273]

```
207     if ( nickel_in == 1 )
208         state <= THIRTY ;
209     else if ( dime_in == 1 )
210         state <= THIRTY_FIVE ;
211     else if ( quarter_in == 1 )
212     begin
213         dispense <= 1 ;
214         state <= IDLE ;
215     end
216     THIRTY:
217     if ( nickel_in == 1 )
218         state <= THIRTY_FIVE ;
219     else if ( dime_in == 1 )
220         state <= FORTY ;
221     else if ( quarter_in == 1 )
222     begin
223         dispense <= 1 ;
224         nickel_out <= 1 ;
225     end
226     THIRTY_FIVE:
227     if ( nickel_in == 1 )
228         state <= FORTY ;
229     else if ( dime_in == 1 )
230         state <= FORTY_FIVE ;
231     else if ( quarter_in == 1 )
232     begin
233         dispense <= 1 ;
234         dime_out <= 1 ;
235         state <= IDLE ;
236     end
237     FORTY:
238     if ( nickel_in == 1 )
239         state <= FORTY_FIVE ;
240     else if ( dime_in == 1 )
241     begin
242         dispense <= 1 ;
```

THIRTY state does not set state back to IDLE after it dispenses a drink.

455 © Cadence Design Systems, Inc. All rights reserved.

This page does not contain notes.

Fixing the Error

1. Open dkm.sv in your favorite editor.
2. Navigate to the THIRTY case branch in the state_machine module.
3. Set state to IDLE when dispense is set to 1.
4. Save the file!

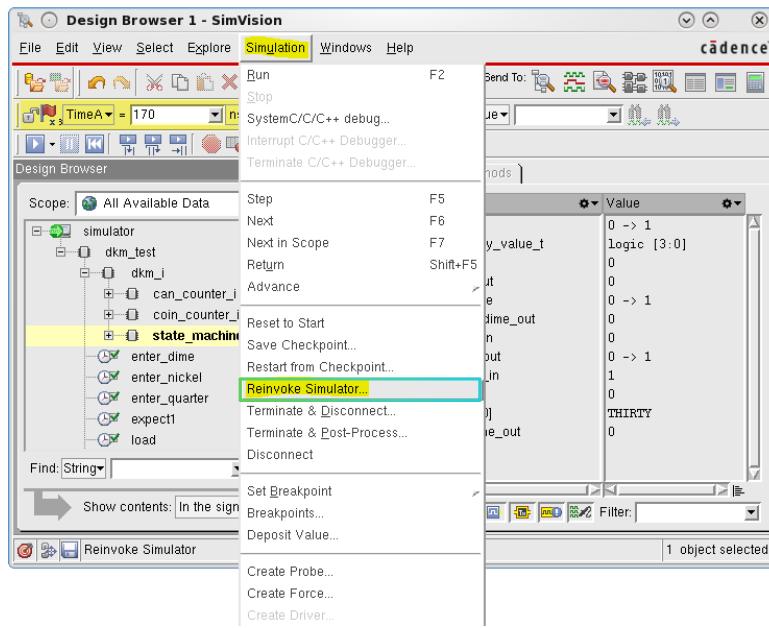
```
File Edit Tools Syntax Buffers Window Help
state <= TWENTY ;
else if ( quarter_in == 1 )
state <= THIRTY_FIVE ;
FIFTEEN;
if ( nickel_in == 1 )
state <= TWENTY ;
else if ( dime_in == 1 )
state <= TWENTY_FIVE ;
else if ( quarter_in == 1 )
state <= FORTY ;
TWENTY;
if ( nickel_in == 1 )
state <= TWENTY_FIVE ;
else if ( dime_in == 1 )
state <= THIRTY ;
else if ( quarter_in == 1 )
state <= FORTY_FIVE ;
TWENTY_FIVE;
if ( nickel_in == 1 )
state <= THIRTY ;
else if ( dime_in == 1 )
state <= THIRTY_FIVE ;
else if ( quarter_in == 1 )
begin
dispense <= 1 ;
state <= IDLE ;
end
THIRTY;
if ( nickel_in == 1 )
state <= THIRTY_FIVE ;
else if ( dime_in == 1 )
state <= FORTY ;
else if ( quarter_in == 1 )
begin
dispense <= 1 ;
nickel_out <= 1 ;
state <= IDLE ;
end
THIRTY_FIVE;
if ( nickel_in == 1 )
state <= FORTY ;
```

456 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Rerunning the Simulation from Any Window



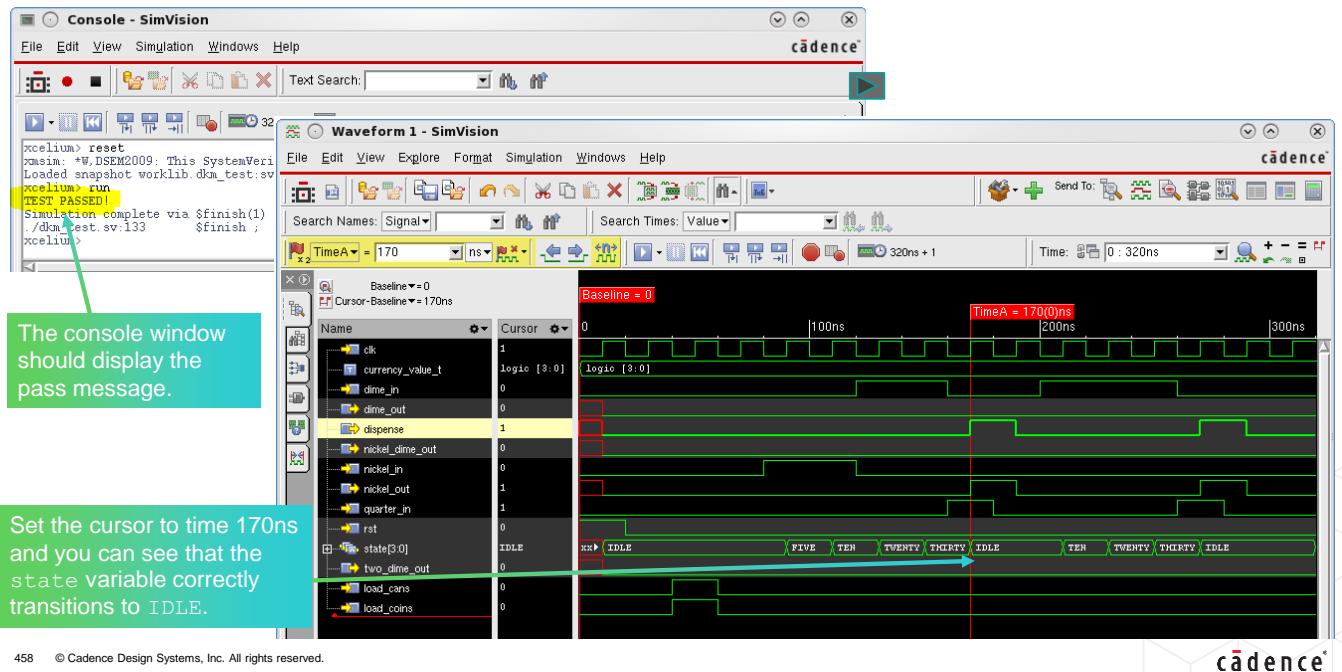
Select **Simulation -> Reinvoke Simulator**
to recompile with your modified code.

If there is compilation error, fix
the error and Reinvoke again.



This page does not contain notes.

Viewing the Passed Results



This page does not contain notes.

Summary of Problem and Solution

- **Error/problem**

- There is an error in the drink machine logic. It sometimes does not reset to the IDLE state when a drink is dispensed.

- **Given**

- Drink machine design and the testbench.
 - SimVision tool.

- **Goal**

- To use your preferred text editor or the one that SimVision opens to fix this error.
 - To rerun the simulation to ensure that the behavior is correct.

- **Strategy**

- Perform debug using the SimVision GUI, fix the code using your favorite editor, and rerun the design.

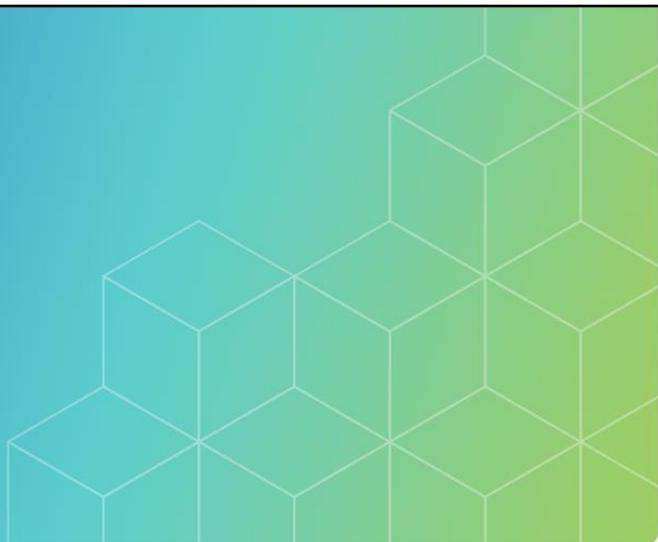
- **Results**

- Error fixed and the drink machine design works as it should.

459 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.



Submodule 5-2

Introduction to a Modern Verification Flow

cadence®

This training module introduces you to a modern verification methodology.

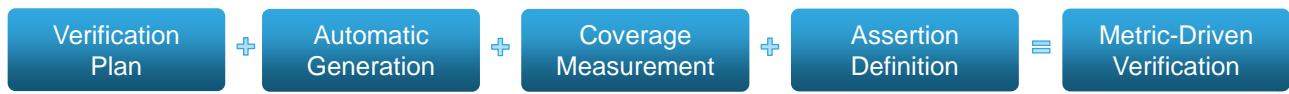
Submodule Objectives

In this submodule, you will:

- Identify the need for and nature of a Metric-Driven Verification (MDV) flow.

Topics include:

- Examining the verification problem
- Automatic stimulus generation with randomization
- Coverage to measure what has and has not been tested
- Assertions to check for correct and incorrect behavior
- Defining a Verification Plan



461 © Cadence Design Systems, Inc. All rights reserved.



In this training module, you explain the need for, and nature of, the MDV methodology.

What Is the Verification Problem?

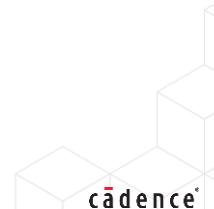
The **verification problem** is how to scale the verification effort to keep pace with demand.

The fundamental verification activity is to stimulate the DUT and check the DUT response.

- The problem is how to scale this activity to keep pace with continually increasing DUT size and complexity.

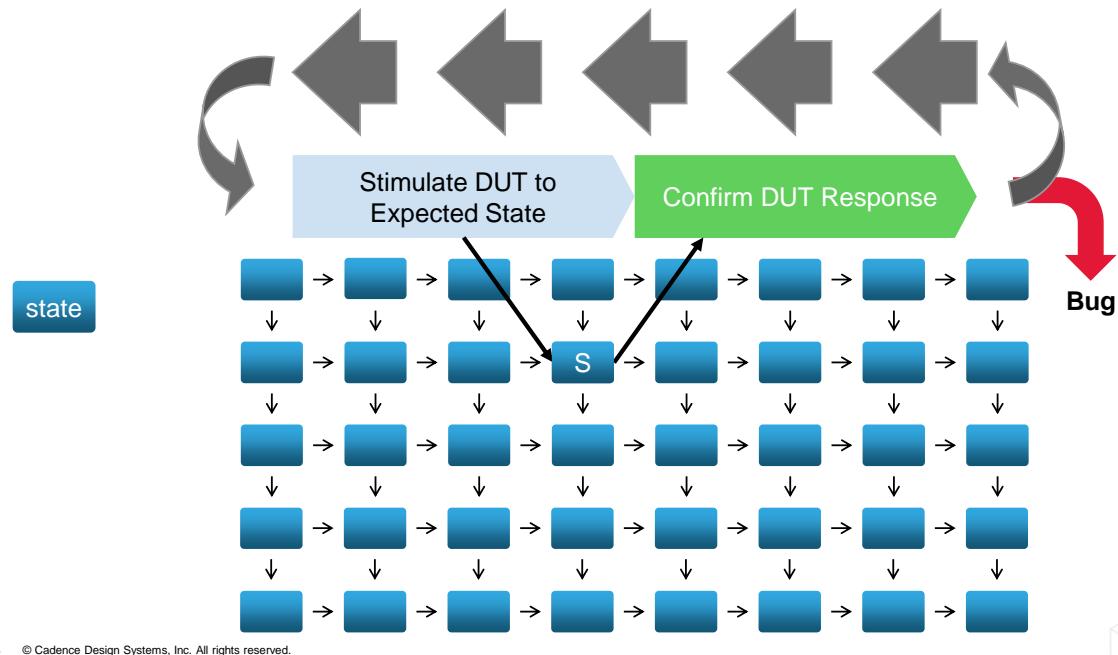


462 © Cadence Design Systems, Inc. All rights reserved.



The verification effort is at its most abstract level to control and observe, that is, apply stimulus and confirm the correct response.

Typical Verification via Simulation



463 © Cadence Design Systems, Inc. All rights reserved.



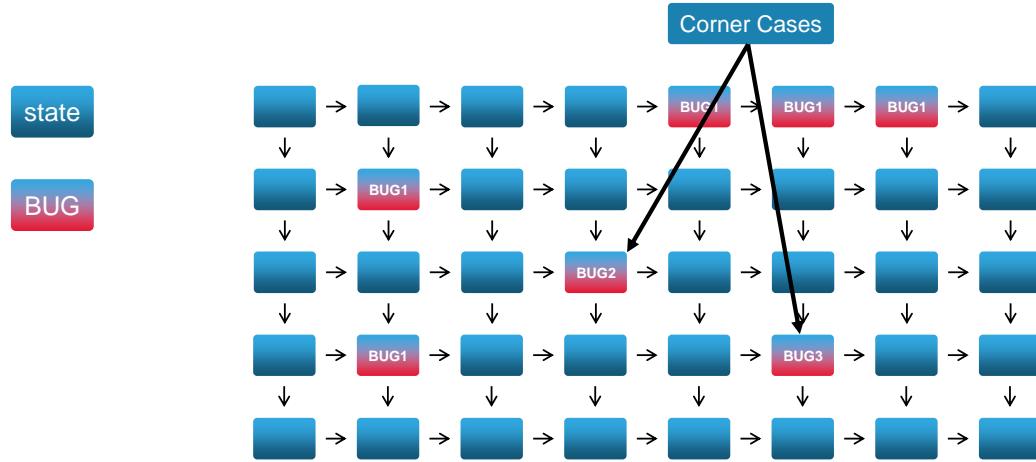
Typical verification via simulation stimulates the DUT to what we hope is a specific state and then confirms the DUT state and outputs. A design “bug” can prevent the DUT from achieving the expected state or producing something other than the expected output.

Where Are the Bugs?

Most “bugs” are quickly found and fixed.

The elusive bugs are revealed only by “interesting” states.

These are typically the “corner case” states.



464 © Cadence Design Systems, Inc. All rights reserved.



Where do the “bugs” hide? Most bugs are quickly found. Elusive bugs are revealed only by unusual states. These are typically the corner case states less frequently visited.

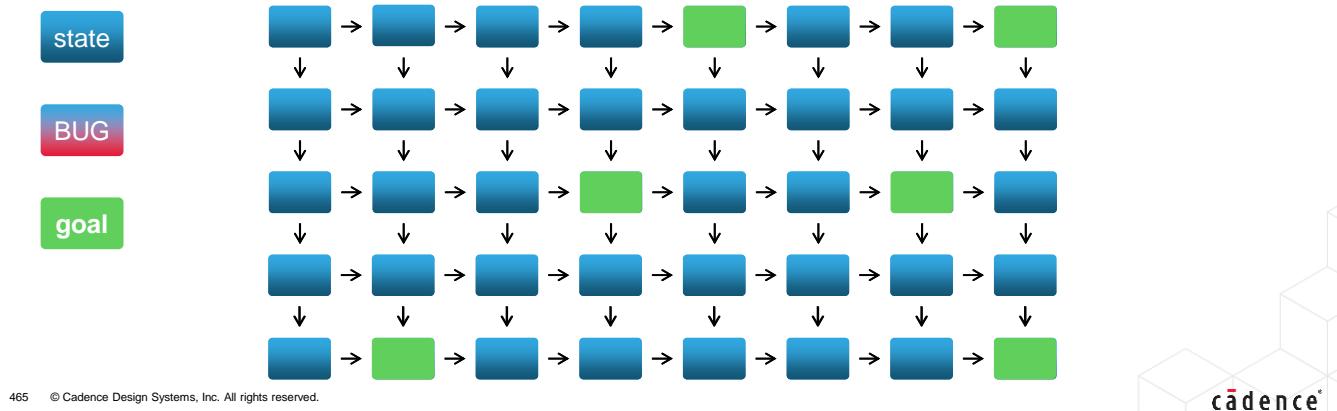
Illustrated here are one easily detected bug and two corner-case bugs.

Verification Goal States

The verification goal is to reveal the bugs.

To reveal bugs, the verification team establishes “goal” states to visit.

They do this based on analyzing the specification, identifying key features, and team experience.

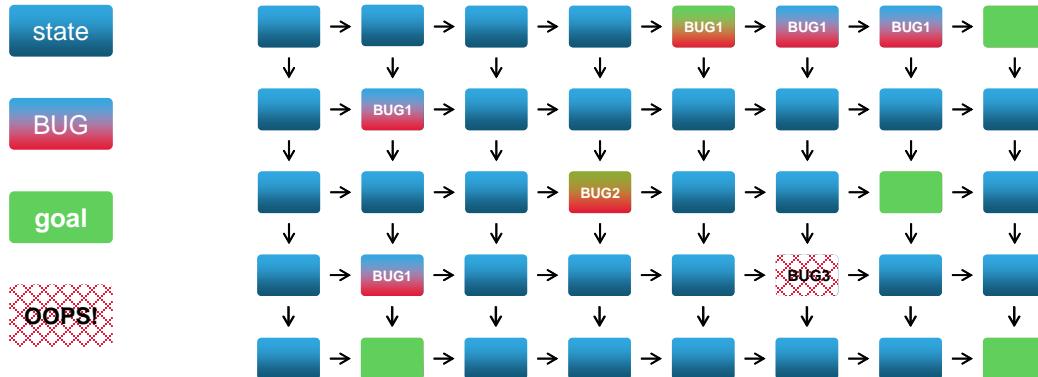


The verification goal is to reveal the bugs. To reveal bugs, the verification team establishes “goal” states to visit. It determines these states based on analyzing the specification, identifying key features, and applying team experience.

Here, the team has selected some goal states to visit.

Hit and Missed Bugs

- Verification goals may not reveal all bugs.
- Some bugs may be revealed only by states outside of the goals.
- Here, the verification plan missed a corner-case bug.
- We must not limit verification to only the *known* goals!



466 © Cadence Design Systems, Inc. All rights reserved.



The verification goals hit some bugs and miss some bugs. Some bugs may be revealed only by states outside of the verification goals. Here, the verification plan missed a corner-case bug.

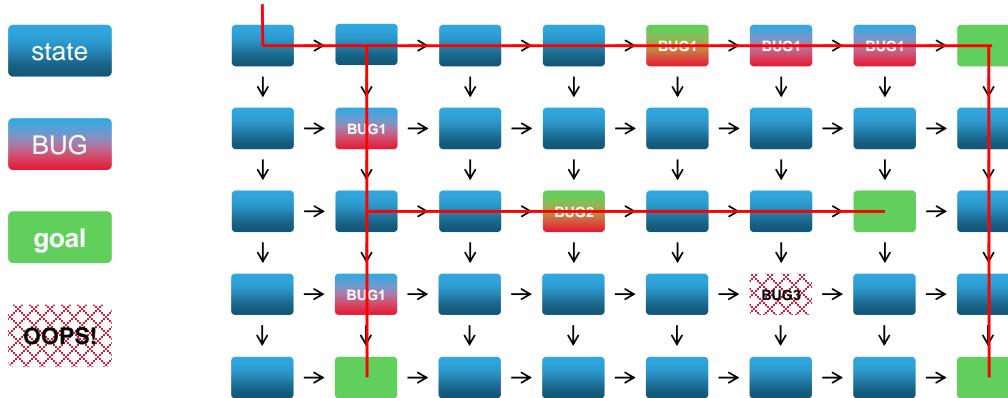
We must not limit verification to only the known goals!

Legacy Directed Test “Gate Banging”

The verification engineer wrote a directed test for each item in the verification plan.

- **No Automation** – Manual effort required to determine and follow the paths.
- **No Automation** – Manual effort required to verify that goals were reached.
- **No Automation** – Non-goal scenarios unlikely to be exercised.

Repeat for every design change!



467 © Cadence Design Systems, Inc. All rights reserved.



Traditionally, the verification engineer wrote a directed test for each item in the verification plan. The test would apply a stimulus and confirm the response. None of the efforts was automated.

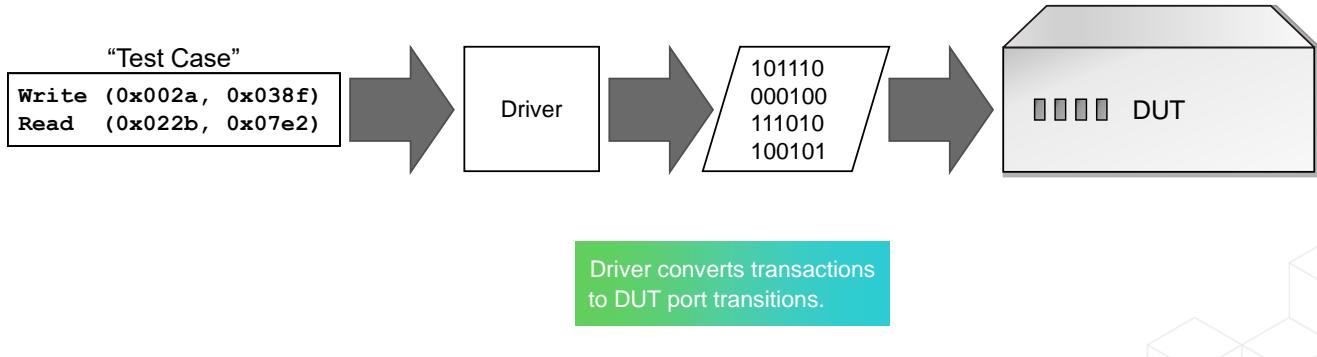
- The manual effort was required to determine and follow the paths;
- The manual effort was required to verify that goals were reached; and
- Non-goal scenarios were unlikely to be exercised.

Here, the test implemented the verification plan. It revealed the easily detected bug and a corner-case bug but missed the unplanned-for corner case bug.

The Case for Transaction-Based Verification

Moving stimulus generation to a higher level of abstraction improved productivity.

- The verification engineer wrote tests calling SystemVerilog tasks (VHDL procedures, C functions).
- The manual effort was required to determine and follow the paths (now via parameter selection).
- The test stimulus intent was still not well documented.
- The expected response was still typically built into the test.



468 © Cadence Design Systems, Inc. All rights reserved.



Moving stimulus generation to a higher level of abstraction improved test writing productivity.

- The verification engineer wrote tests calling SystemVerilog tasks (VHDL procedures or C functions).
- The manual effort was still required to determine and follow the paths, now via parameter selection;
- The test stimulus intent was still not well documented; and
- The expected response was still typically built into the test.

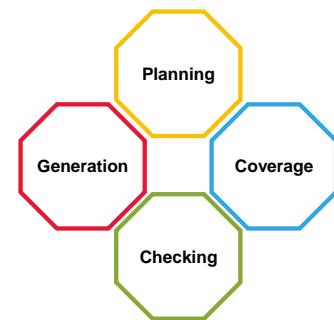
Here the test case calls a write routine, passing to it the address and data to write, and calls a read routine, passing to it the address and expected data.

What Is the Verification Problem Solution?

The verification problem **solution** is **Metric-Driven Verification** or **MDV**!

MDV is a powerful layer of methodology that sits above the Verification Testbench environment, which provides guidelines and tools for using metrics and automation to maximize the benefits of the verification testbench.

- It is a data-driven, decision-based flow that improves the Predictability, Productivity and Quality of the verification effort.
- MDV is a **closed-loop process** with the following stages:
 1. **PLAN** an Executable Verification Plan.
 2. **CONSTRUCT** a testbench and other verification infrastructure using Universal Verification Methodology (UVM).
 3. **EXECUTE** the regression on various engines such as Simulation, Formal Verification, Emulation, etc.
 4. **MEASURE AND ANALYZE** the results to see what else is left to be done to finish.



469 © Cadence Design Systems, Inc. All rights reserved.

The Metric-Driven Verification (MDV) methodology has four key components:

- Planning your verification goals and measuring your progress toward them.
- Generation of random stimulus constrained to meaningful values and ranges of values;
- Checking to confirm that the design reacted appropriately to the stimulus;
- Coverage to confirm that the stimulus is generated, and exercised design features as planned;

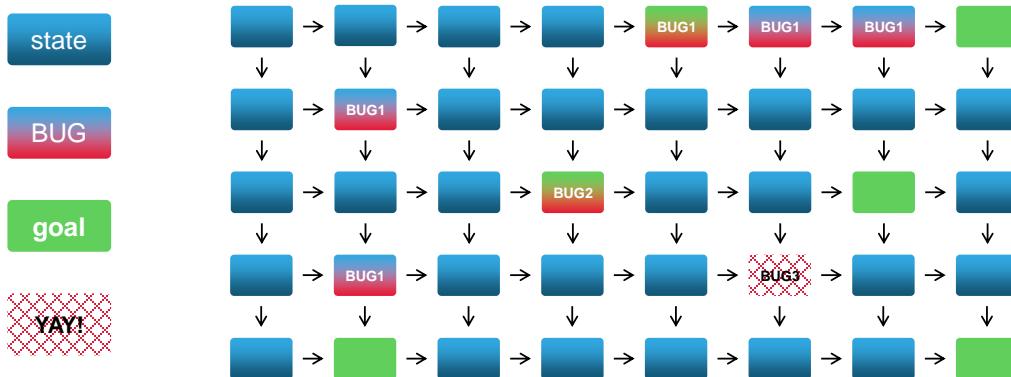
Generation with Constrained Randomization

Random stimulus (biased toward goal areas) partially automates test generation.

- Automation – Less manual effort required to determine and follow the paths.
- Automation – Non-goal scenarios more likely to be exercised.
- No Automation – Manual effort *still* required to verify that goals are reached.
- No Automation – Manual effort *now* required to check for expected results.



Repeat for every design change!



470 © Cadence Design Systems, Inc. All rights reserved.



To partially automate test generation, verification engineers began to incorporate random stimulus, biased toward the goal areas.

This automated or semi-automated some of the verification process.

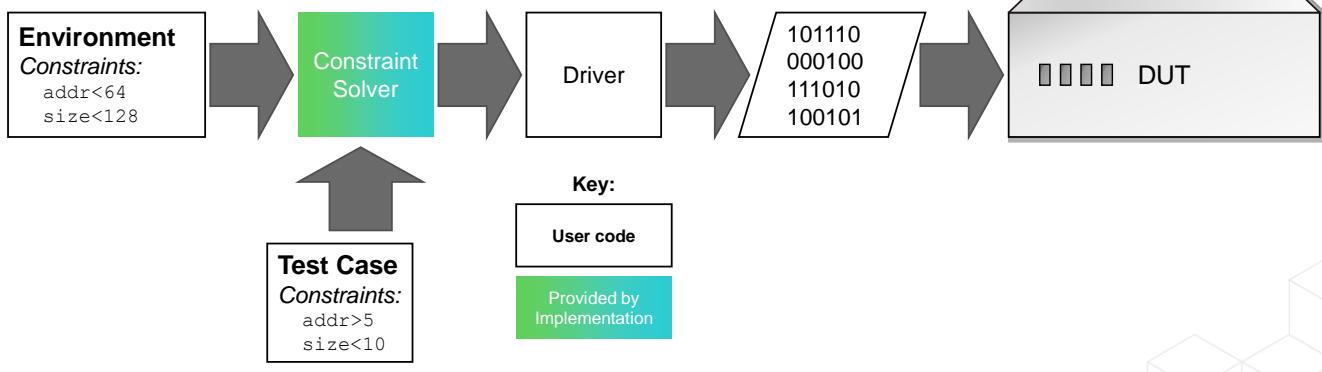
- The less manual effort was required to determine and follow the paths; and
- Non-goal scenarios were more likely to be exercised; but
- The manual effort was *still* required to verify that the goals were reached; and
- The manual effort was *now* required to cull the ineffective stimulus.

Here, the verification plan included biased random stimulus. Its implementation revealed the easily detected bug and both corner-case bugs, including the unplanned-for corner case bug.

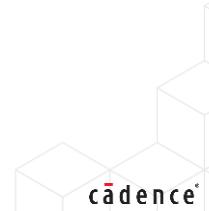
Constrained Random Stimulus

Your test case is constraint sets.

- The *Verification Environment* specifies the *allowable* value ranges.
- Each *Test Case* specifies the *desired* value ranges.
- The test stimulus intent is now better documented.
- The expected response is now removed from the test.



471 © Cadence Design Systems, Inc. All rights reserved.



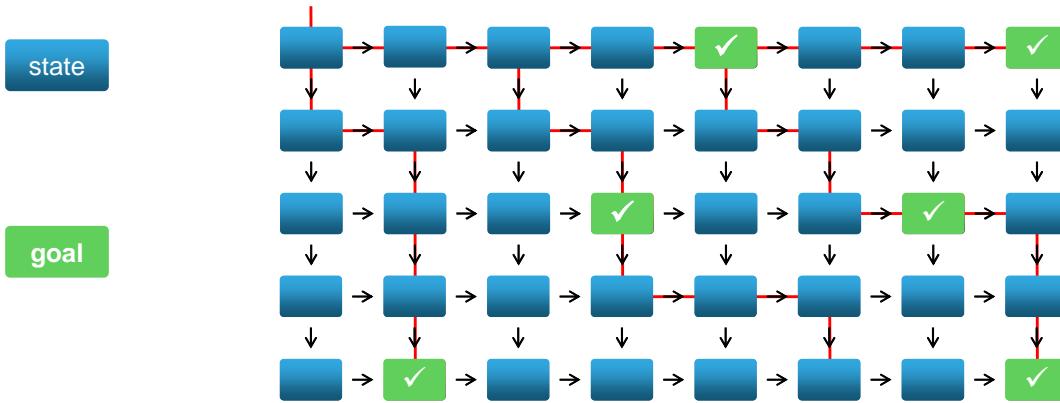
With constrained random stimulus, your test case is constraint sets.

- The *Verification Environment* specifies the allowable value ranges;
- Each *Test Case* specifies the desired value ranges within the allowable ranges;
- Constraint comments and names now better document the test stimulus intent; and
- The expected response is now removed from the test case.

Coverage Measures: What Is Tested?

Coverage partially automates verification that goals are reached.

- **Automation** – Coverage verifies what is and what is not tested.
- **Automation** – Coverage enables culling the ineffective stimulus.
- **No Automation** – Manual effort to add coverage to your verification environment.



472 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Coverage Makes Automatic Stimulus Generation Effective

Without coverage measurement...

- Lacks objective test goal achievement feedback.
- Many constraint sets provide little confidence that specific scenarios have been reached.
- No feedback on the effectiveness of constraint sets, leading to duplication of effort.

With coverage measurement...

- Has objective test goal achievement feedback.
- Fewer constraint sets reach most scenarios.
- Greater efficiency of verification through the removal of redundant or ineffective constraint sets.



Constrained random stimulus is most effective when paired with coverage.

Without coverage data, you have little or no idea what tests, if any, exercised what, if any, scenario.

With coverage data, you know which tests are useful and which design scenarios remain not exercised.

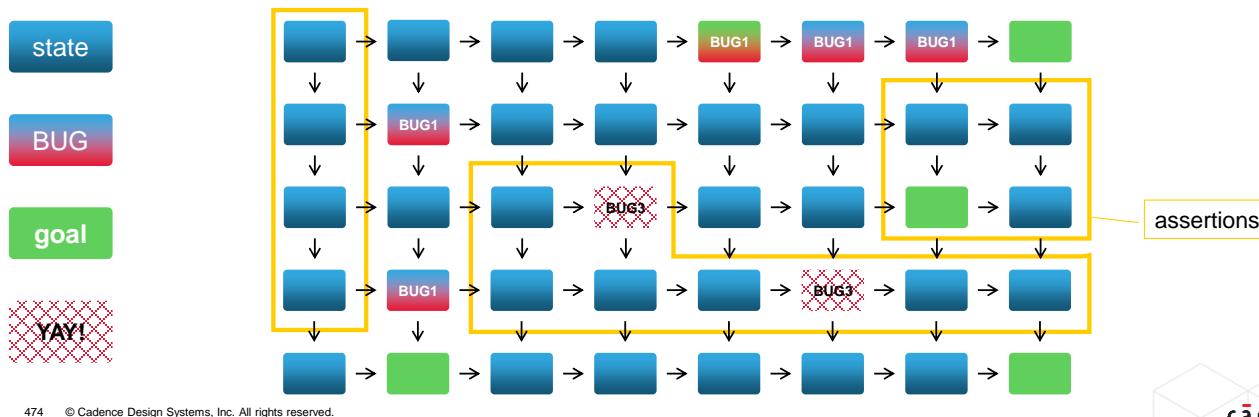
Assertions Check Behavior



Expected results for constrained random stimulus may not be known.

Self-checking can be embedded into DUT with assertions.

- **Automation** – Assertions check for good and bad behavior.
- **No Automation** – Manual effort to add assertions to your verification environment.



474 © Cadence Design Systems, Inc. All rights reserved.



With random stimulus, it may not be possible to construct expected data sets against which DUT values can be checked. The solution is to build self-checking into the test environment and the DUT itself using assertions. Assertions are embedded checkers, written in SystemVerilog, which can check for good behavior and detect bad behavior during the course of simulation. As well as checking, assertion syntax can also be used for coverage measurement and in formal verification.

Metric-Driven Verification Benefits: Automation

Design complexity continually increases....

...Therefore, verification complexity continually increases.

Manual effort cannot continue to increase at the same rate.

Application of manual effort must instead be at increasingly abstract levels.

- Providing stimulus goals.
- Analyzing coverage data.
- Analyzing checker-reported errors.
- Providing more direction when goals are not being met.

Automation

475 © Cadence Design Systems, Inc. All rights reserved.



Verification complexity continually increases.

The manual effort cannot continue to increase at that rate.

The manual effort must instead be made more productive by being applied at increasingly abstract levels.

- Providing stimulus goals;
- Analyzing checker-reported errors; and
- Providing more direction when goals are not being met.

This “providing more direction” should ideally be as automated as practical.

Metric-Driven Verification Benefits: Completeness

Verification complexity makes it impossible to imagine all failure scenarios.

We need a way to exercise failure scenarios other than those imagined.

Constrained random stimulus tests a DUT more thoroughly than directed tests.

Random stimulus is most effective when paired with coverage.

Completeness

476 © Cadence Design Systems, Inc. All rights reserved.



Verification complexity makes it impossible to imagine all failure scenarios.

We need a way to exercise failure scenarios other than those imagined. Constrained random stimulus focuses on goal areas but also as an unplanned side-effect exercises scenarios that the test plan, and thus directed test, might have missed.

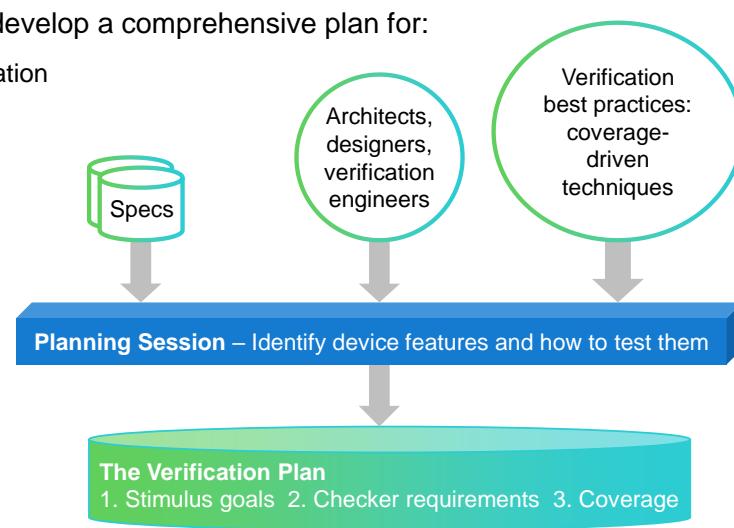
Constrained random stimulus is most effective when paired with coverage. Coverage confirms whether the goals are reached and enables ranking the stimulus sets and eliminating ineffective or redundant stimulus sets.

The Verification Plan

So, you want to use the MDV methodology but don't forget your *planning*.

Using a planning process, develop a comprehensive plan for:

- Automated stimulus generation
- Independent checking
- Coverage collection



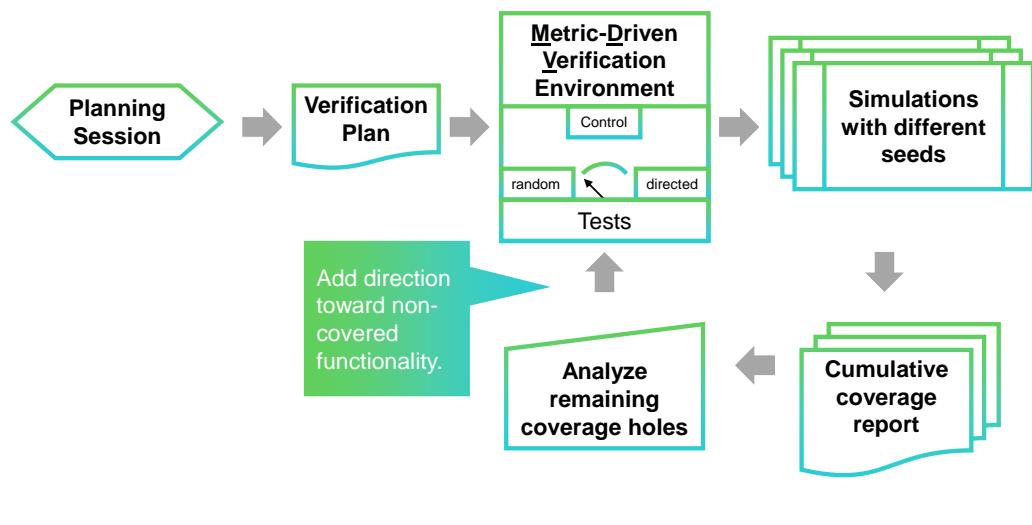
477 © Cadence Design Systems, Inc. All rights reserved.

So you want to use the MDV methodology, but don't forget your *planning*.

Using a planning process that draws upon appropriate resources, develop a comprehensive plan for:

- Automated stimulus generation;
- Independent checking; and
- Coverage collection.

The Verification Process



478 © Cadence Design Systems, Inc. All rights reserved.



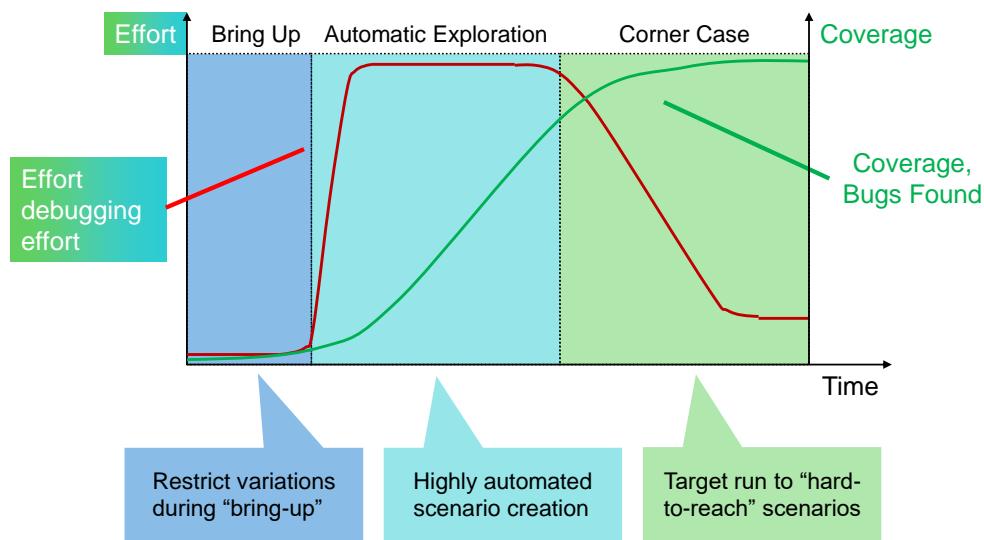
From the verification plan, you develop or reuse a verification environment and develop tests. Recall that tests are constraint sets. The tests can be as flexible, or not, as you want them to be.

You feed the verification environment and tests into multiple simulations with different seeds.

Each simulation generates coverage data, which you then analyze to determine where the coverage holes are, and also to cull the ineffective or redundant tests.

To address the coverage holes, you add tests with constraints modified to address those goal areas.

Verification Project Lifetime



479 © Cadence Design Systems, Inc. All rights reserved.



This sketch somewhat describes what you can expect to put into, and get out of, your test development efforts.

Every new verification project starts with a short interval in which you are “bringing up” the verification environment. At this point, your constraints have little variation and you are not really detecting and fixing too many bugs.

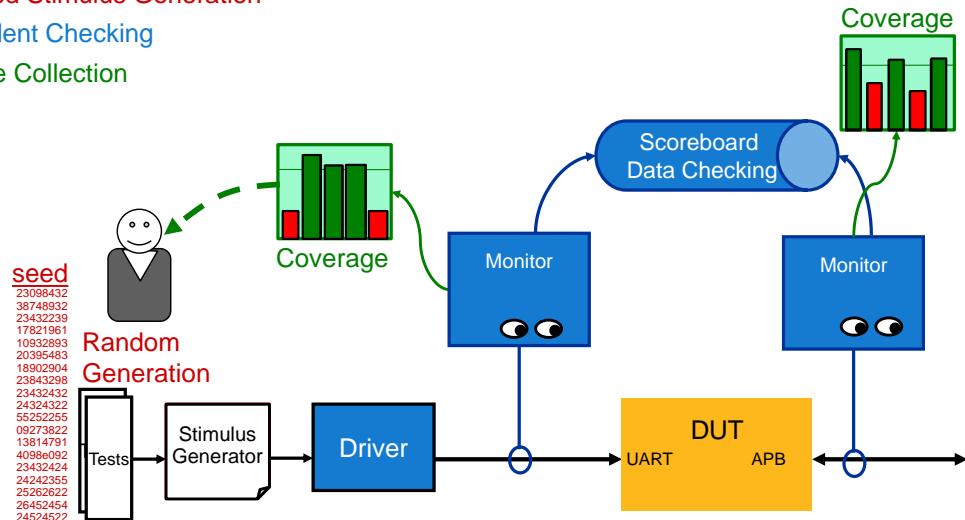
After “bringing up” the verification environment, a longer interval follows in which tests are automatically generated in accordance with your expanded constraint variations, while your team is spending much of its time discovering and fixing how neither the design nor their understanding of the specification really matches the specification.

Lastly, the project starts to wind down, the debug effort drops off, as some team members leave to bring up other projects, and the remaining team members spend sleepless nights in Seattle wringing out the last few bugs while total coverage asymptotically approaches 100%.

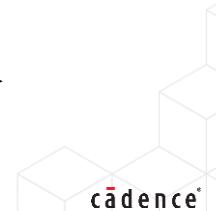
Metric-Driven Verification Review

Components of a MDV environment

- Automated Stimulus Generation
- Independent Checking
- Coverage Collection



480 © Cadence Design Systems, Inc. All rights reserved.



A Metric-Driven Verification methodology is composed of three main features:

1. Random stimulus to test the widest possible design functionality with the smallest stimulus code.
2. Assertions and independent checking to define behavior which should or should not happen, and so find bugs in the design.
3. Coverage so we can determine what has and has not been tested. Coverage feeds back to the test writer, who can then modify the stimulus to close coverage holes.

Class-Based Testbench

Traditional testbenches are written using a hierarchy of modules:

- Just like RTL code.

SystemVerilog adds classes:

- C/Java-like, Object-Oriented (OO) software-like programming constructs.
- Used for writing testbenches and simulation models.

Why do we use classes for testbench development?

- Easier code reuse.
- Configuration and customization.
- Powerful support for constrained randomization, coverage and checking.
- Pre-written class library of verification building blocks, simulation control, debugging features, etc.:
 - Universal Verification Methodology (UVM)



The SystemVerilog language includes C/Java-like class constructs for Object-Oriented (OO) design. The class features of SystemVerilog are not synthesizable, and therefore not used in RTL design, but are used for writing testbenches and simulation models.

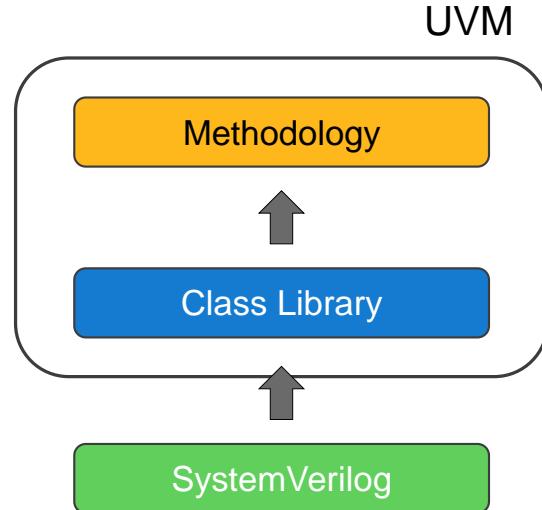
Using classes for testbench creation provides many significant advantages:

- Classes can be more easily packaged for internal and commercial reuse.
- Object-Oriented programming allows you to redefine almost any part of a testbench using inheritance and subclasses, while leaving the original code untouched. This allows easy, safe and convenient reconfiguration and customization of your testbench components, giving far better flexibility for reuse.
- Constrained randomization, coverage and checking features are far more powerful in a class environment rather than a traditional module-based testbench.
- But the most powerful advantage is UVM, a standard class library of pre-written verification building blocks, simulation control and debugging features (and much more). UVM is basically an Application Programming Interface (API) for verification, providing a set of SystemVerilog subprograms, wrapped in classes, with which a verification engineer can build a testbench environment.

What Is UVM?

A class library of verification building blocks written in standard IEEE1800 SystemVerilog. Supports features such as phasing, messaging, factory, etc.

- A proven verification methodology:
 - Defines how to use the class library.
 - Scalable from block-level to system-level verification.
- IEEE standard 1800.1:
 - Documented and maintained by IEEE.
 - Open source reference library from Accellera.
 - Apache license.
 - Does not lock users into a single vendor solution.



482 © Cadence Design Systems, Inc. All rights reserved.

UVM is a library of SystemVerilog classes that implement verification building blocks along with support features such as phasing, messaging, factory, etc. The library provides additional class automation, such as print, copy, comparison, cloning, etc., which are not available in SystemVerilog.

The library API is defined as an IEEE standard (1800.1), and Accellera (the industry standard body) provides a reference implementation of the library. By extending our testbench environment from the UVM library, we can exploit the built-in automation and support features.

However, UVM is also a methodology for using the building blocks and features of the library. By following the methodology, you can create scalable verification components which can be used from block-level to system-level verification.

By following the methodology and the architecture it dictates, we can create standardized, reusable verification blocks.

UVM has an Apache license, which allows commercial IP to be built and sold on top of UVM.

All this means that companies can create internal, external and commercial verification IP conforming to UVM, hence allowing the greatest possible reuse.

UVM Library Benefits

Library classes provide verification functionality and building blocks.

For example:

- Structural components
 - Verification blocks supporting a standard architecture
- Automation
 - Automatic implementation of print, copy, compare, etc. subprograms
- Simulation control
- Reporting

Benefits:

- Productivity
 - How much manual work can be saved?
- Predictability
 - Will I meet my deadlines on time?
- Quality
 - Will my testbench be free of bugs?



UVM is a library of SystemVerilog classes that implement verification building blocks along with support features such as phasing, messaging, etc.

Automation refers to the automatic generation of common class operations such as copy, print and compare, which are not available in SystemVerilog. By generating these for us, UVM saves us from having to write these ourselves. In general, UVM provides the building blocks for testbench generation, saving us considerable time.

Developing a testbench using UVM building blocks allows us to leverage tried and tested code and therefore minimizes bugs and errors.

UVM Methodology

Methodology defines how to use library classes.

Benefits

- Proven solution for verification success.
- Captures best-known practices with minimal improvisation.
- Provides consistency and uniformity in the way verification components are developed and used.
 - Internal and commercial reuse can dramatically speed up delivery and improve quality.
- Accommodates unanticipated requirements.
 - Supports easy reconfiguration, customization and modification.
- Ease of use.
 - How easily can a Test Writer develop tests for an existing verification environment?
- Supports key verification features required for Metric-Driven Verification (MDV).



Although the first version of UVM was only established in 2009, it leverages the verification methodology developed by Cadence's Specman® tool, which was released in 1996. So UVM has a proven success record and leverages decades of experience from prior tools.

A good methodology, based on a standard verification component architecture, increases productivity and predictability, as it saves much time otherwise taken in planning and partitioning the testbench design.

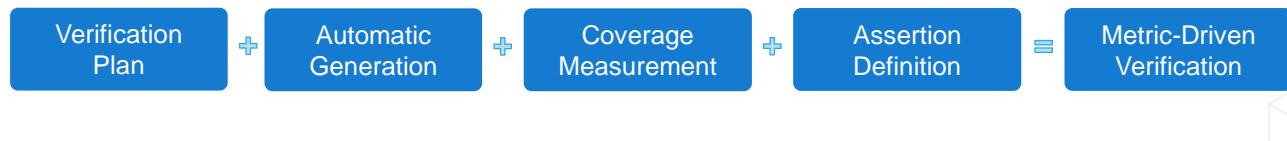
UVM recognizes that a reusable block or third-party verification IP may never completely match the requirements of a user and so provides powerful configuration and type-override capabilities to allow the user to customize and modify verification blocks.

A test writer is a verification engineer who develops tests, according to the verification plan, for an existing verification environment. UVM is designed to allow a test writer to have maximum control without requiring an in-depth knowledge of the environment. UVM also supports key verification features required for MDV.

Module Summary

You should now be able to explain the workings and benefits of the Metric-Driven Verification (MDV) methodology.

- Why you need an advanced verification methodology
- How the MDV methodology meets that need
- Implemented in your testbench using UVM

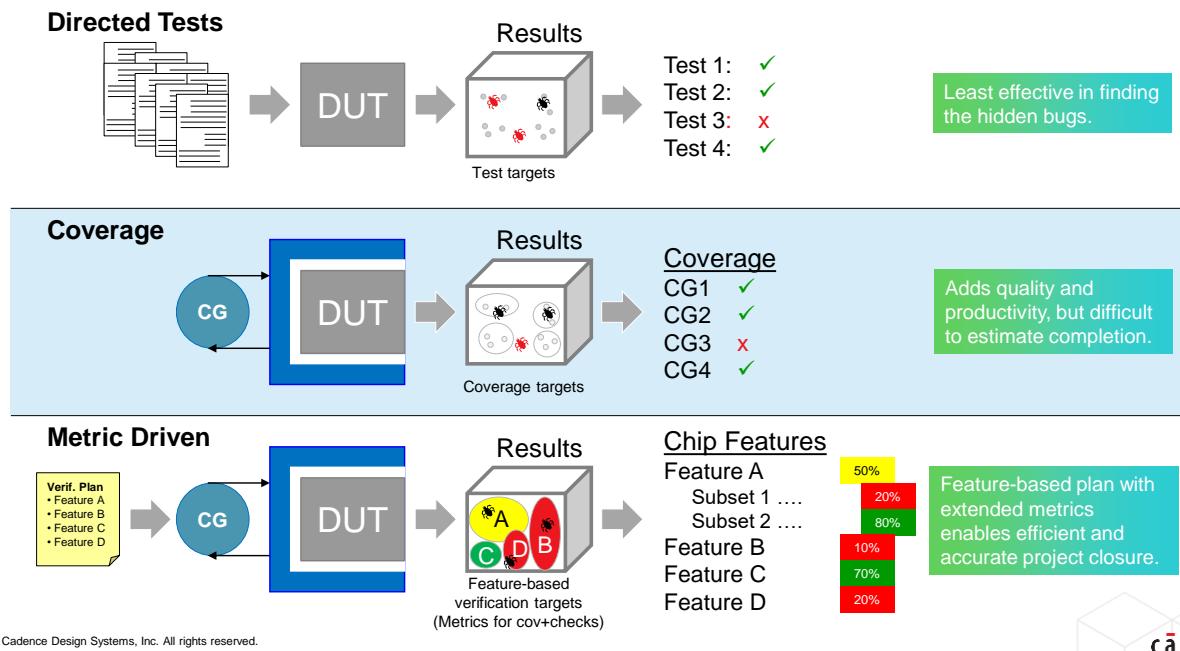


485 © Cadence Design Systems, Inc. All rights reserved.

cadence®

In this training module, you explained the need for and nature of the MDV methodology. You justified the effort that you will be putting into this training course. You examined why you need an advanced verification methodology, how the MDV methodology fills that need, and very briefly, to what you apply coverage metrics.

Functional Verification Methodologies Review



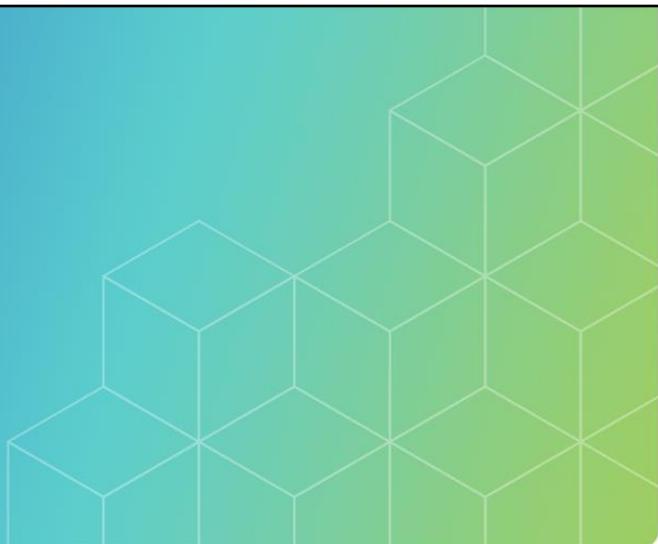
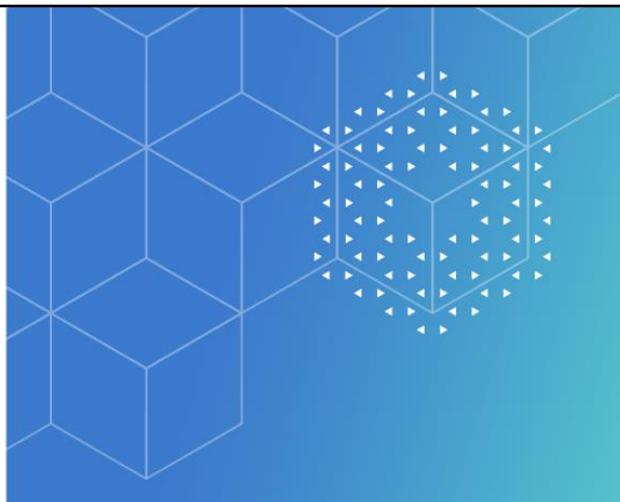
486 © Cadence Design Systems, Inc. All rights reserved.



This training module initially discussed the legacy directed-test verification methodology. This methodology focuses on test generation. This methodology is now not viable due to the complexity of modern designs and the competitive pressures that prevent us from merely throwing more manpower at the verification problem.

The Metric-Driven Verification (MDV) methodology utilizes automated constrained-random generation checkers to confirm design operation and coverage to confirm test completeness. All are controlled via the planning process with verification project management tools.

This methodology focuses on a higher level of design features. With the project management tools, we know not just how complete the test is but also how much more time we need to achieve a specific level of completeness.



Submodule 5-3

Introduction to MDV and Planning

cadence®

This training module introduces you to Metric driven verification or MDV and the concept of verification planning.

Submodule Objectives

In this submodule, you will:

- Define MDV
- Define the planning aspect of vManager
- Identify metrics
- Map metrics to your vPlan

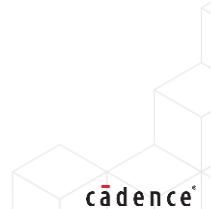


In this submodule, you will:

- Define MDV.
- Define the planning aspect of vManager.
- Identify metrics.
- Map metrics to your vPlan.

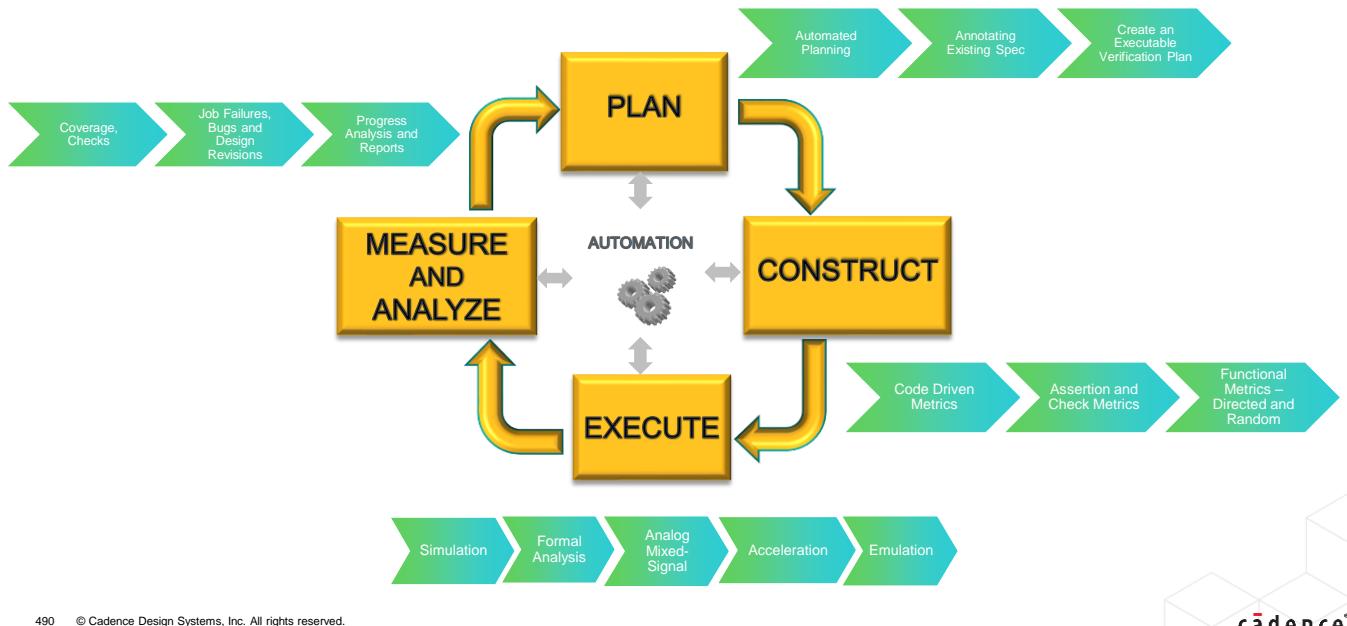
What Is Metric Driven Verification or MDV?

- MDV is a powerful layer of methodology that sits above the Verification Testbench environment, which provides guidelines and tools for using metrics and automation to maximize the benefits of the verification testbench.
- It is a data-driven, decision-based flow that improves the Predictability, Productivity and Quality of the verification effort.
- MDV is a **closed-loop process** with the following stages:
 - **PLAN** an Executable Verification Plan
 - **CONSTRUCT** a testbench and other verification infrastructure using Universal Verification Methodology (UVM)
 - **EXECUTE** the regression on various engines such as Simulation, Formal Verification, Emulation, etc.
 - **MEASURE AND ANALYZE** the results to see what else is left to be done to finish



This page does not contain notes.

MDV: A Closed-Loop Process



490 © Cadence Design Systems, Inc. All rights reserved.



- Planning:** The MDV solution utilizes the most powerful specific-purpose verification plan authoring software, the vPlanner tool, which is an integral part of the Cadence vManager™ solution. The MDV flow starts with automated planning, either annotating existing specifications with verification intent or creating an executable verification plan using team member inputs captured in a spreadsheet.
- Construction:** MDV spans from simply directed test methodologies through to register-transfer level (RTL)-based coverage-driven and software-driven approaches. For RTL verification, the Universal Verification Methodology (UVM) standard has been specifically designed to provide powerful constrained random testing. The power of constrained random stimulus is that vectors are automatically generated without having verification engineers spell out each set of input vectors manually. With the constrained random stimulus come constrained random changes in state. In this environment, it is imperative that the verification team is able to specify which states have been explored and that for those explored states, the correct response to stimulus vectors is observed. For this reason, functional coverage is utilized. Functional coverage is defined for and captures scenarios that are important for the verification environment to observe and capture.
- Execution:** Verification simulation, formal analysis, analog mixed-signal, acceleration, and emulation execution engines are dispatched and the convergence of results from multiple verification engines is managed within the Cadence vManager environment.
- Measurement/Analysis:** MDV is used to measure real-time progress with on-demand information as well as to determine when high-quality verification closure is achieved. Coverage, checks, and assertions provide the verification-specific metrics used to determine closure. Verification job failures, bugs, and design revisions all provide insight into the actual status of a project. Progress analysis and reports help the verification team make adjustments to their resource allocation (people and tools) to reach closure more efficiently and measure closure more accurately.

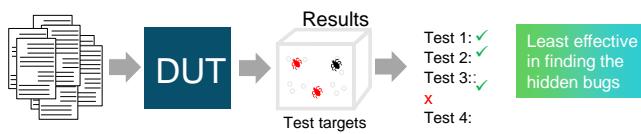
What Are the Other Prevalent Different Methods of Verification?

There are other prevalent Verification Methods in the industry, such as the Traditional Directed Testing Approach and Coverage Driven Verification (CDV).

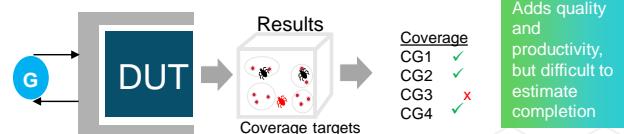
- **Traditional Directed Testing Verification** involves a planning process that usually documents what test cases need to be written, but the underlying reason for the said test cases should be due to design analysis.
- Manual testing is tedious and labor-prone.

- **Coverage-Driven Verification** methodology, or CDV, has been proven to be able to identify bugs faster.
- It can produce huge amounts of coverage data, which can be overwhelming when trying to analyze which design features belong to which coverage points. This overwhelming amount of data is why coverage-driven has many limitations that affect its usability and scalability.

Directed Tests Driven



Coverage Driven



491 © Cadence Design Systems, Inc. All rights reserved.

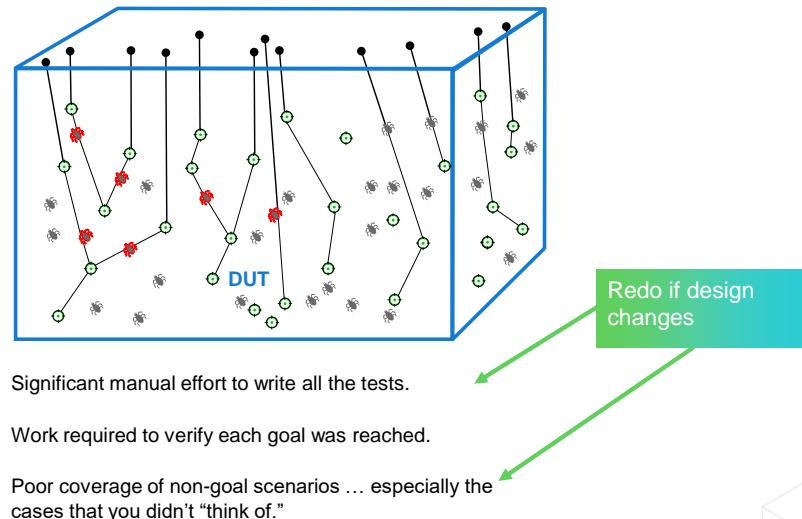
cadence®

The plan-based verification method, MDV, is the most successfully adopted method of verification as compared to the others. Let us see why. And then list the benefits of using the MDV Method.

Traditional Approach: Directed Testing

Verification Engineer:

- Defines DUT states to be tested based on specified behavior and corner cases.
- Writes directed test to visit and verify each item in the test plan.



- Traditional Verification Planning process usually indicates what test cases need to be written, but the underlying reason for the said test cases should be due to design analysis. Usually, the classical test plan will then just document what directed tests need to be written. The test plan should be the set of goals to achieve, but most plans document the means by which to achieve those goals.
- Since manual testing is tedious and labor prone, the typical method of copy-pasting snippets in test cases results in relatively low variability from one test case to the next.

Coverage-Driven Verification

Verification Engineer:

- Defines DUT states to be tested based on specified behavior and corner cases.
- Focus moves to reach goal areas (versus execution of test lists).

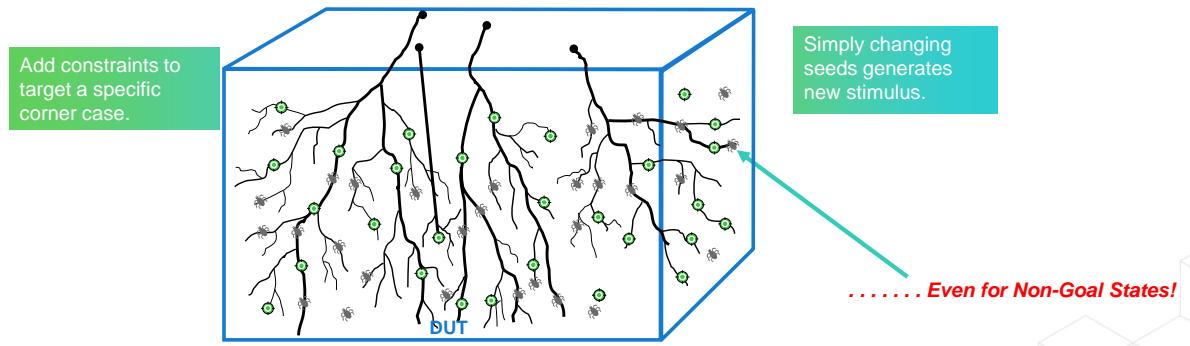
Constrained random stimulus generation explores goal areas (and beyond).

Coverage reports which *goals* have been exercised and which need attention.

Self-Checking ensures proper DUT response.

Automation:

- Constrained random stimulus accelerates hitting coverage goals and exposing bugs.
- Coverage and checking results indicate the effectiveness of each simulation.
- Enables many parallel runs to contribute in order to scale the verification effort.



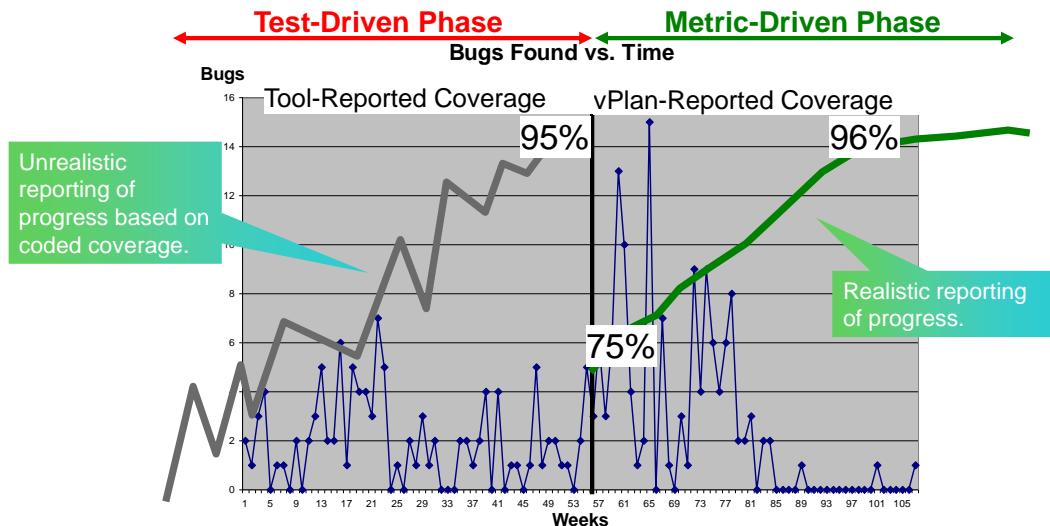
493 © Cadence Design Systems, Inc. All rights reserved.

Coverage-driven verification methodology or CDV has been proven to be able to identify bugs faster.

However, it is difficult to see what has been actually tested because the testbench is, by definition, randomized. Coverage driven is added to visibly see what is tested and what is not during this randomization process. While it is relatively easy to add functional coverage, it can produce huge amounts of coverage data, which can be overwhelming when trying to analyze which design features belong to which coverage points. This overwhelming amount of data is why coverage driven has many limitations that affect its usability and scalability.

Note: The limitations of CDV have been explained in detail in our new ES (Education Services) course, *Foundations of Metric Driven Verification*.

Tool-Reported Coverage Versus Plan-Based Coverage

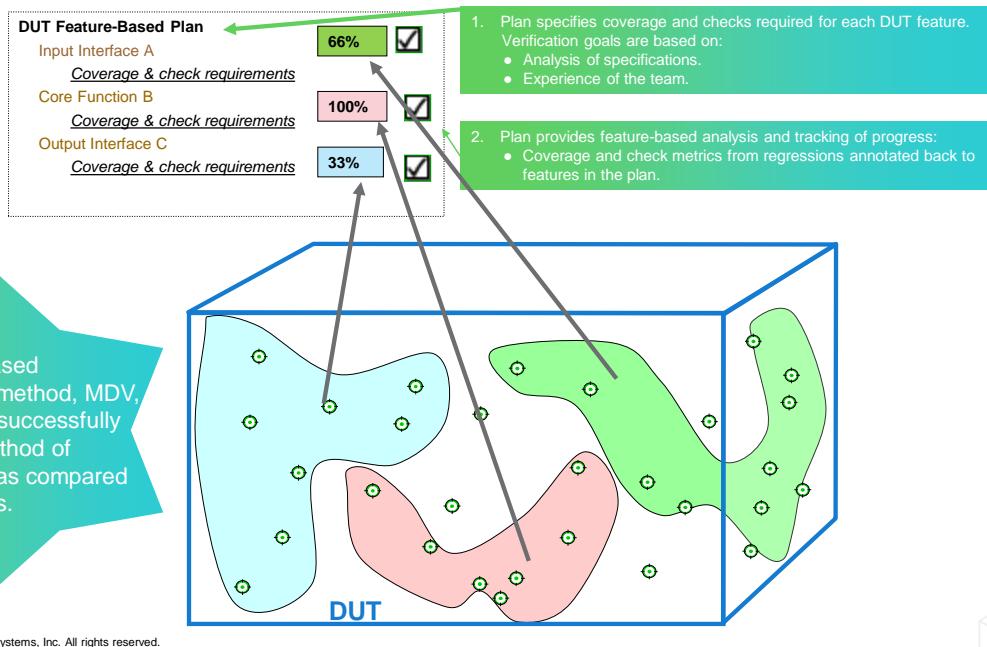


494 © Cadence Design Systems, Inc. All rights reserved.



- Here, we can see the problem of accurately estimating the true completion of the verification due to the lack of a vPlan.
- Before the CDV stage started, we estimated the functional coverage at 95% based on coverage reported by Specman®. As we formalized the vPlan for CDV, we were surprised to see that the actual coverage is only 75%.
- The difference is due to the fact that the normal flat coverage model cannot incorporate the complexity of different levels of importance, while the vPlan makes it possible, thus resulting in a more realistic result.

Planning Is Essential: Defines “What” to Include in CDV Environments

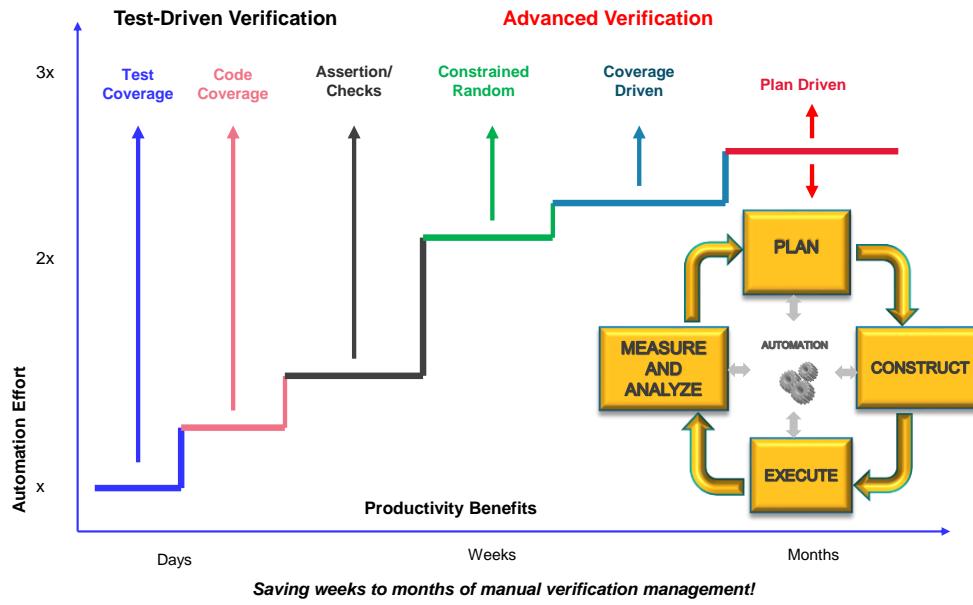


495 © Cadence Design Systems, Inc. All rights reserved.

cadence®

- We refer to the “what” as the “features” or “goals” that must be verified – the design intention. When planning a project, you should not presume the underlying tool by which a feature is verified. It will most likely come from several sources, with the results represented by various forms of coverage. Luckily, this top-down approach is not the only way to begin, and starting bottom up, with automation of the test plan, provides instant value in and of itself.
- A verification plan should capture the goals of the verification process in a way that the signoff criteria and milestones are clearly identified. An example of such a milestone is RTL code freeze, which a team might define as the point when 75% of all features are covered. In addition to capturing goals upfront, a verification plan should be executable.
- **Planning answers “what” you need in the verification environment – stimulus, coverage, and checks.**

MDV Productivity Benefits

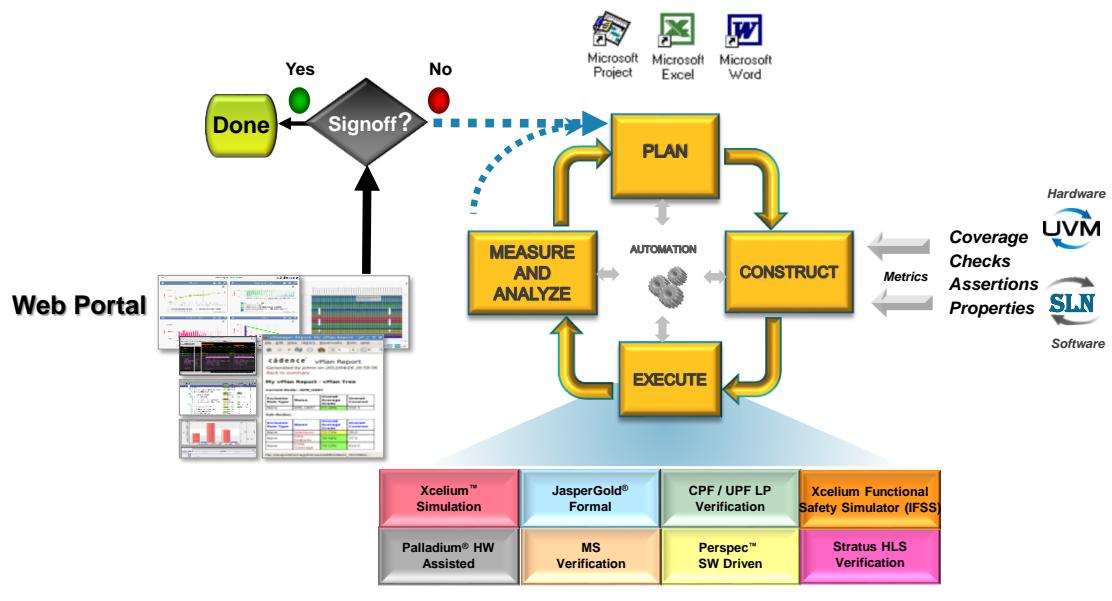


496 © Cadence Design Systems, Inc. All rights reserved.

cadence®

- This is a very interesting graph that shows the practicality and benefits of MDV. The metrics in MDV can start as simple “test case coverage” and extend to plan-based closure of design features using the notions of an executable verification plan. As one would expect, increasing investment in automation provides equally increasing productivity benefits from the perspective of time saved or product quality.
- Test-driven verification** is RTL based and design-centric. It is a very automatic and unobtrusive way to collect test coverage by measuring which tests have passed and which have failed and performing failure analysis.
- Next, adding **code coverage** is as simple as turning on a switch, but now you have visibility to see which parts of the code have been exercised, improving the overall quality of the Verilog or VHDL design.
- Finally, adding assertion-based functional testing with PSL or SVA is a natural fit for additional testing of functionality.
- Advanced RTL verification** techniques combine constrained random coverage and plan-based techniques to the RTL verification effort. Constrained random testing has proven to be able to identify more bugs faster. However, it is difficult to see what has been tested because the testbench is randomized. While it is relatively easy to add functional coverage, it can produce huge amounts of coverage data, which can be overwhelming when trying to analyze which design features belong to which coverage points. This is the reason coverage-driven verification has many limitations that affect its usability and scalability.
- Plan-based/plan-driven MDV** broadens the scope of what is captured and measured to include checks, assertions, software, and time-based data points that are encompassed in the term “metrics.” The plan-based notation means that you can organize your verification plans by features, by milestones, by design hierarchy, or all of the above including your own definitions.

Review of the Metric Driven Verification (MDV) Cycle



497 © Cadence Design Systems, Inc. All rights reserved.

cadence

MDV is a powerful, open and innovative approach to pre-silicon functional verification which Cadence pioneered over 10 years ago, and we continue to be the industry leader in this area. Our plan to closure process is represented by this diagram of planning, construction, execution and measurement and analysis of the results.

On the planning side, we connect to top-level plans, excel tests lists, requirements management systems and specifications. vManager provided a specific purpose verification planning tool which is used for plan authoring. Construction of your testbench is done through UVM for hardware testing, and SLN for software and system testing. This is where you define what metrics you wish to capture that are representative of the features you are trying to verify.

Next is the execution side. The vManager tool is a multi-user and multi-engine environment able to support the various Cadence verification engines for optimal productivity and performance. From Xcelium simulation and Jasper formal for IP-based designs to Palladium® with software-driven testing for a subsystem or SOC designs, the MDV solution is very scalable and flexible to accommodate digital and analog, hardware and software, IP to SOC. Measurement and analysis are done by the vManager tool, which allows many ways to drill into the results to identify holes and display the results. vManager uses a commercial SQL database to aggregate data such that real-time trends and web charts are visible and available to all. Reporting is done through text or as show, HTML reports, which are drillable down to the lowest levels of the design.

- The figure shows the complete MDV cycle starting from planning to the signoff phase.
- The vManager tool spans the planning, execution of the tests/plan, measurement of metrics and finally the analysis and management of the available metrics.
- The construction phase basically includes building the verification infrastructure such as test cases, testbench, assertions, etc., based on the UVM methodology.

Benefits of Metric Driven Verification

Higher Quality

- Find more bugs faster, combining formal ABV, constrained random simulation, and HW/SW emulation.
- Information on incomplete areas.

Better Productivity

- Automation for many everyday tasks.
 - Coverage and failure analysis, running regressions, reporting.
- Combine metrics across verification approaches to reduce effort duplication.
- Optimize human and compute resource utilization.

Improved Predictability

- Dynamic visibility into verification closure with respect to the plan.
- Ability to measure progress based on project-specific milestones.
- Answers the question – “are we done yet?”

498 © Cadence Design Systems, Inc. All rights reserved.



Hence MDV answers the top business concerns:

1. Schedule to DV completion
2. Cost of Tapeout (verification effort/people)
3. Functional bug(s!) causing re-spin or recall

Why Is Planning Important?

Any process that needs clear and measurable goals and verification is no exception and definitely needs planning!

- You can gauge improvement only by what you can clearly measure.
- Failing to capture goals at the beginning of a project means no definition of what needs to be done.



MDV defines a verification plan – a list of “what to verify” or the “features” of the design intention.

The traditional test plan is a list of “how to verify” the methods of a design.

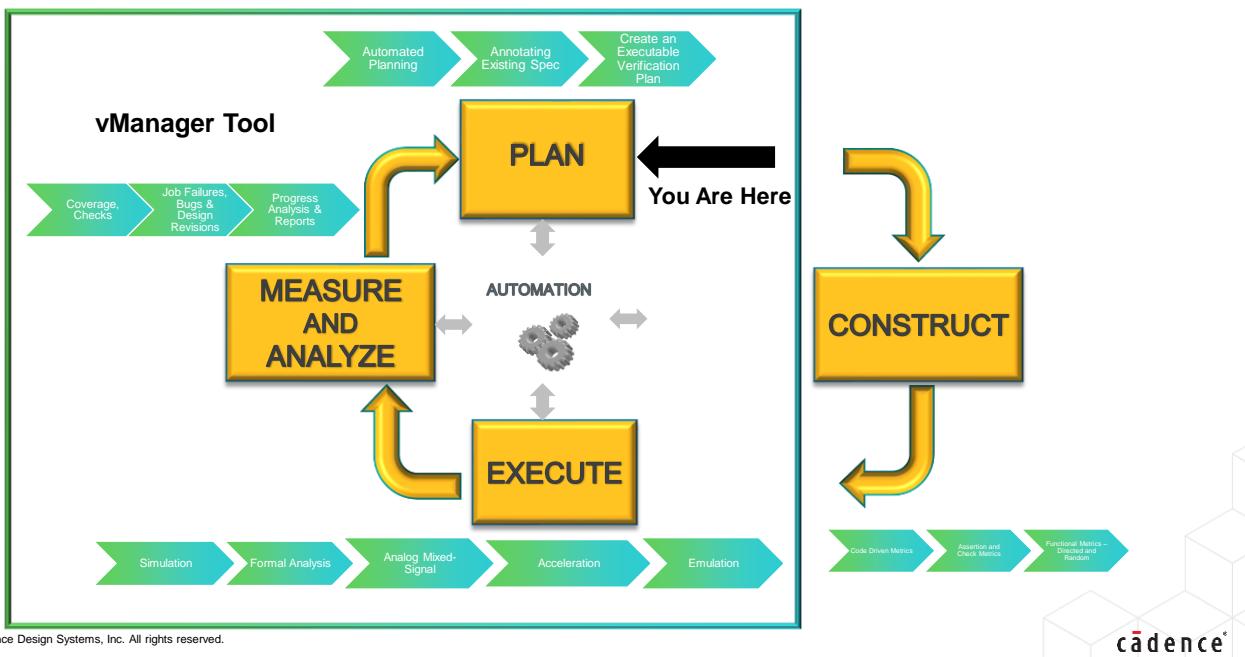
Verification plan – closed and executable process – is measurable.

Test plan is an open process and not executable.

The MDV vManager™ flow starts with a verification plan.



Planning Phase



Motivation for Verification Planning

Given limited resources, we would like to:

- Maximize quality and quantity of verification achieved.
- Mitigate project risk.
- Efficiently execute the task of implementing the verification environment:
 - Understand requirements before writing a bunch of code.
- Forecast and track verification resource requirements.
- Achieve transparency of verification progress.
- Give early warning of missed schedules and design/verification issues.
- Assure an effective plan is produced.
- Measure progress relative to the plan throughout the project.



This page does not contain notes.

What Is Verification Plan?

A Verification Plan captures the goals of the verification process, defining the signoff criteria and milestones clearly.

- **Example:** RTL code freeze
- This plan should be executable.
 - Progress toward the project closure is easily and automatically measured.
- This plan is a living document that matures over the lifetime of a project.

Verification and Validation Plan

Template Version X • Document ID: 000000000000

Over the company, model is developed and maintained. It is used to validate the software design and verify its correctness.

KLARITI.COM

[Company Name]
[Project Name]
[Revision Number]

3 Verification and Validation Plans

Describe the main Verification and Validation tasks to be achieved throughout the project lifecycle. The Verification and Validation Plan can be started during the Concept phase. However, as Verification and Validation planning may be dependent on specific tasks in your current, Verification and Validation planning may be performed in conjunction with the overall software development planning effort.

NOTE: Evaluate proposed changes for effects on previously completed V&V tasks.

3.1 Management

Management will be comprised of general responsibilities for monitoring, controlling, reporting, and managing the V&V plan. Provide details of the following management tasks:

- Baseline Change Assessment
- Management Review
- Review Support
- Verification and Validation Plan Generation

The Verification and Validation Plan begins during the concept phase of the project. Other areas to observe include:

- Comparing V&V task results to project management reviews.
- Evaluating proposed changes for effects on previously completed V&V tasks.
- Identifying key review requirements.
- Reviewing other V&V items, technical accomplishments, resource utilization, future planning, and risk management.

3.2 Concept Phase

The major V&V activity in this phase is to develop an approach for reviewing and testing the software. Testing may involve a single generation or has been by a user or the development of elaborate simulation and needs generators.

Evaluate the concept documentation to determine if the proposed concept satisfies the user's needs and objectives. Identify the main constraints posed by interacting systems. This phase establishes the basis for the proposed system.

.....

502 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

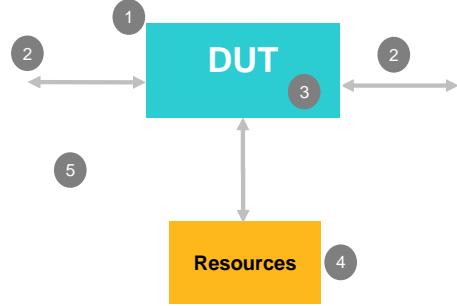
What Is the Feature Planning Approach?

Feature planning focuses on “what” to verify versus “how.”

- From specifications.
- From experience of the project team.

Some examples of design features include:

1. Device mode and configuration options. Traffic or protocol handling.
2. Protocol or device exception handling.
3. Internal device properties.
4. Arbitration for external resources.
5. Typical and critical use cases.



Feature-Based Design Analysis

Prerequisite: Design intent is understood by the stakeholders.

- A stakeholder is a participant in a verification project, such as Design/Verification Engineer, Program Manager, etc.

Review the design specifications and consider the following:

- What functionality does the design provide?
- What expected behavior is triggered by a scenario? Or a test case?
- Do not brainstorm as to how the features should be verified.
- Write down a description for each feature. Use words like “The design...”
 - ...supports...”
 - ...is able to...”
 - ...handles...”



This page does not contain notes.

Feature-Based Design Analysis (continued)

- Avoid phrases (because they have a “Directed Test” implication) like:
 - “check that...”
 - “verify...”
 - “if <this> then <that>”
- Organize the identified features in a hierarchical fashion:
 - Structure should consider the need for tracking progress.
- Granularity and fidelity of identified features depend on:
 - Stakeholders involved.
 - Design complexity.
- Create separate categories for:
 - Architectural details.
 - Implementation details.
 - Specialized scenarios (next slide).

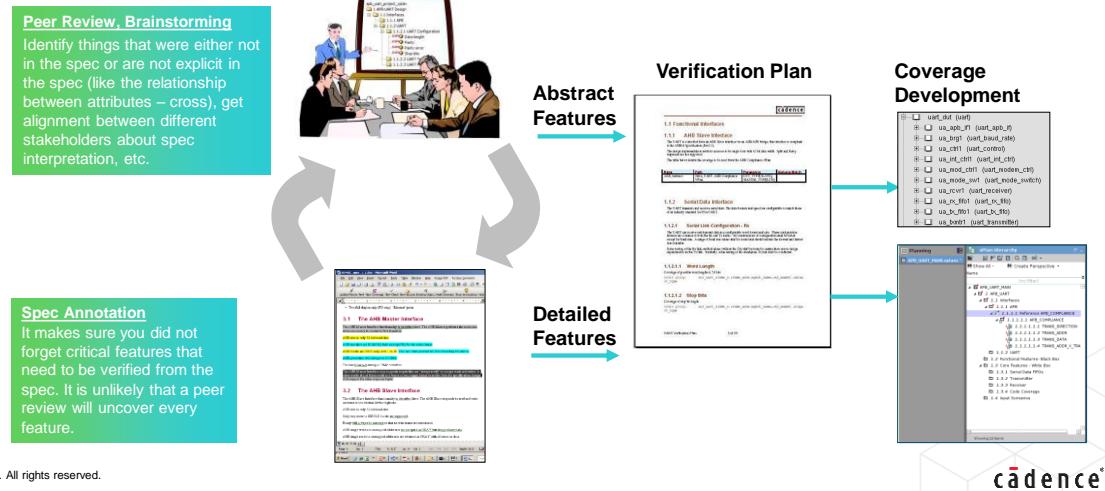
505 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

Feature Planning (*What to Verify?*)

- A verification plan is needed to capture the features that need to be verified.
- Peer review sessions, as well as a thorough spec analysis, are good techniques for extracting the main features and structure of the verification plan.
- Spec annotation or peer review. Start with one and continue to the other...



This page does not contain notes.

Example of Feature-Based Plan: The “WHAT”

Focuses First on “What,” Not “How,” to Allow Involvement of the Entire Team

Feature-Based Plan

1. System Bus Subsystem features to be verified:
 - 1.1 Transfer Types: The bus supports both single transfers and burst transfers.
 - 1.2 Transfer Directions: The bus supports both read and write transfers.
 - 1.3 Transfer Targets: The bus supports transfers from the master to all slaves.
2. Packet Interface
 - 2.1 ...
3. Graphics Engine
 - 2.2 ...

Benefits

- Plan-specific measurable goals for what must be verified.
 - For example, it can be measured with functional coverage.
- Plan expresses the exact intent of “what” will be verified (not “how”) – can be understood by **all** stakeholders.
 - For example, the System Architect who doesn’t know SV syntax.
- Plan provides tracking of exactly what has been verified.
 - “Your latest plan-based coverage status shows that you have not tested burst transfers yet.”



Example of a Feature-Based Plan: The “WHAT” Focuses First on “What,” Not “How,” to Allow Involvement of the Entire Team.

Attribute Elaboration: The “How”

Translating Feature Requirements into Concrete Metric Goals

The verification plan features are translated into a set of quantifiable metrics by asking **HOW** they will be measured.

Which attributes and values are important for each feature?

Where should each value be observed?

When are the values valid to be sampled?
Produce results that can be measured.

Feature-Based Plan	WHAT?
<ol style="list-style-type: none">1. System Bus Subsystem features to be verified:<ol style="list-style-type: none">1.1 Transfer Types: The bus supports both <u>single</u> transfers and <u>burst</u> transfers.1.2 Transfer Directions: The bus supports both <u>read</u> and <u>write</u> transfers.1.3 Transfer Targets: The bus supports transfers from the master to <u>all slaves</u>.2. Packet Interface<ol style="list-style-type: none">2.1 ...3. Graphics Engine<ol style="list-style-type: none">2.2 ...	

Implementation Guidelines	HOW?
<u>Coverage Items</u>	
transfer_types : single, burst	
transfer_direction : read, write	
transfer_targets : (all slaves) 0xa000, 0xa004, 0xa008, 0xa00C	
cross : transfer_types, transfer_direction, transfer_targets	
<u>Observed</u> by the system bus monitor	
► Sampled	when transfer_completes



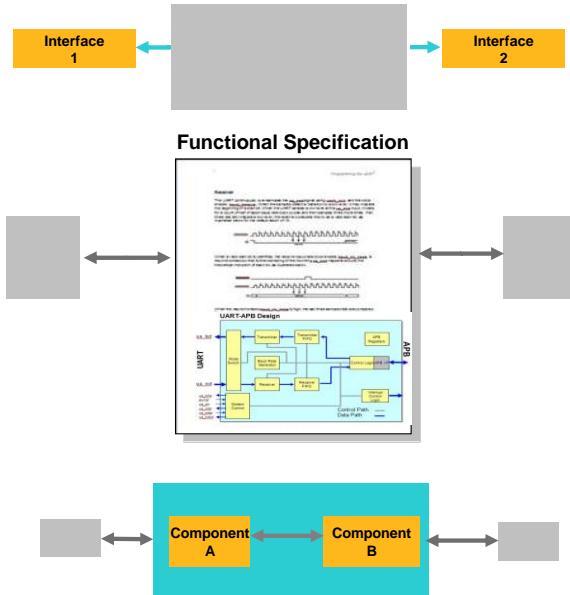
508 © Cadence Design Systems, Inc. All rights reserved.

The slide clearly shows the difference between the verification plan and the test plan.

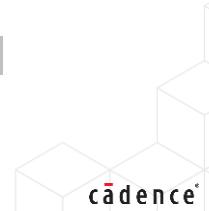
The vPlan or the verification plan describes the “what” features to be verified. The test plan describes “how” these features have to be verified.

vPlan Template

- Interfaces
 - <interface name>
- Functional Features – Black Box (*features measured outside of the design*)
 - Legal Behavior
 - <feature>
 - Exception Handling
 - <feature>
- Core Features – White Box (*features measured in the design*)
 - Components
 - <component name>
 - Interfaces
 - <interfaces>
- Input Scenarios (*results from test cases*)

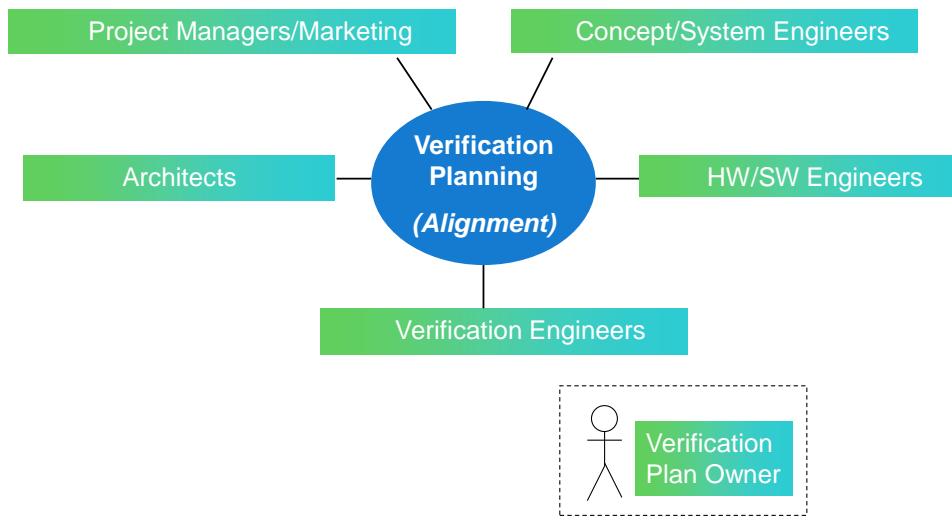


509 © Cadence Design Systems, Inc. All rights reserved.



The vPlan for the UART_APB design is shown.

Verification Planning Stakeholders



510 © Cadence Design Systems, Inc. All rights reserved.

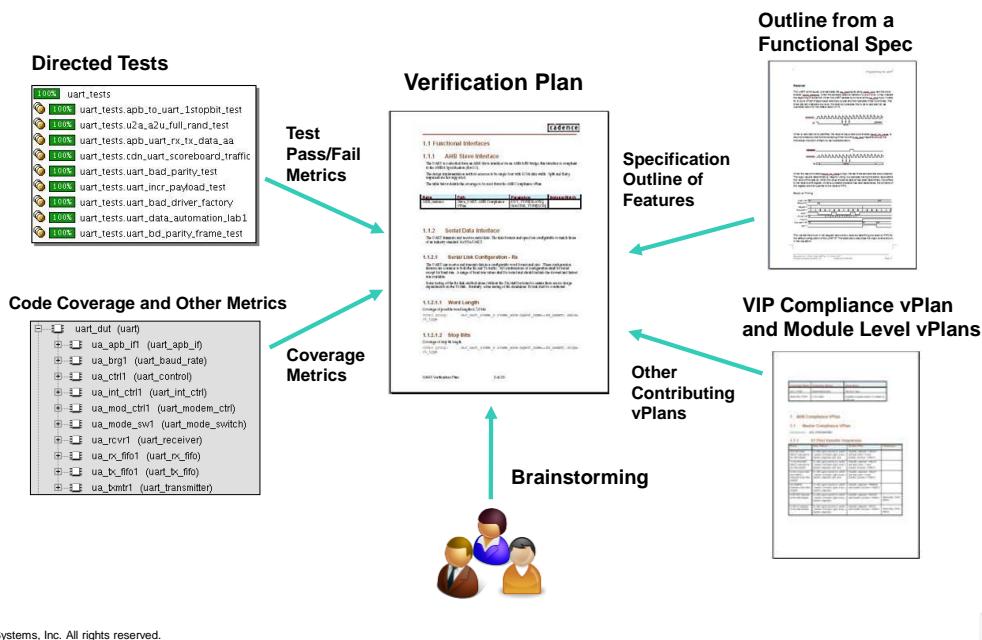


These are all the stakeholders that should be involved in planning for simulation-based verification.

Each of these stakeholders is important, first, for the perspective they can offer to a holistic understanding of the device under verification as described in the module on Verification Planning. Remember that design intent is translated differently by each stakeholder, and the planning session is our opportunity to align these translations.

In addition to contributing their unique viewpoint to the team, each contributor needs to specify and discuss which aspects of each feature are important to them. In some cases, they will help to define metrics that will objectively measure verification closure vs. their concerns. In other cases, they will note metrics that have already been defined and add them to their individually defined “view” of the verification plan.

Verification Planning: Many Sources



511 © Cadence Design Systems, Inc. All rights reserved.

cadence®

There are many sources that constitute the final vPlan and make it a complete one.

The Top-Down Approach

The top-down approach is a generalized brainstorming approach that works for any product regardless of whether written product specifications exist. In this approach, using a block diagram of the design for reference, you list two categories of features – product requirements and design requirements. In each of these categories, you list interface and core requirements. As the requirements are defined, a team member enters them into the vPlanner's verification plan (vPlan) editor.

The Bottom-Up Approach

The bottom-up approach is a more systematic approach that focuses on the specifications for the current project. The team reviews the specifications paragraph by paragraph and identifies the features that are described. Using vPlanner, a team member selects the appropriate text in the specifications document and creates a new section or another element in the verification plan. The selected text appears in the verification plan as part of the description of the new element.

The bottom-up approach is more systematic, but it creates a verification plan whose structure reflects the functional requirements. As a result, the plan might not be easily reusable by other projects.

For best results, use a combination of the two approaches. For example, brainstorm key features and then annotate the specification to ensure nothing is missing from the plan.

Verification Planning: Main Challenges

- How do I know whether my plan is complete?
 - No missing pieces
 - Up-to-date
 - Reflects functional and design specifications including spec changes
- How do we write a plan that can be reused later?
 - From module to system
 - Between projects
 - Among different owners in the team
- How do we track our progress against this plan?
 - Dynamically track completeness of each verification plan feature using measured verification metrics (coverage driven and assertion-based)
 - Accurately predict verification project schedule and track progress *versus* goals established for each project milestone

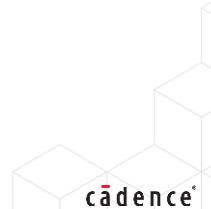
512 © Cadence Design Systems, Inc. All rights reserved.



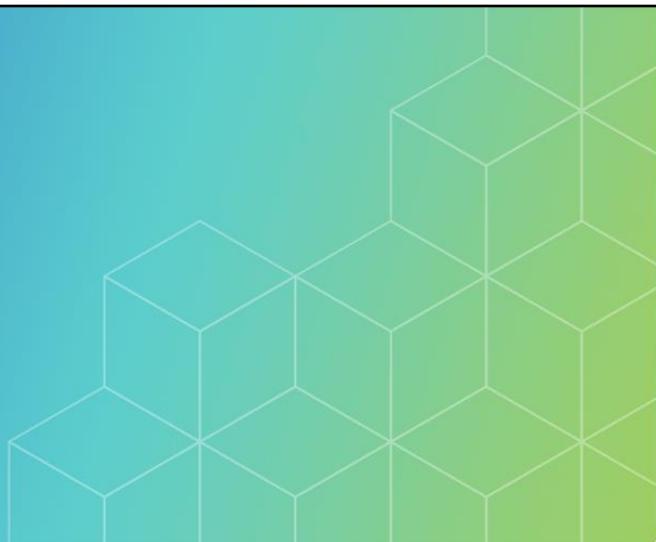
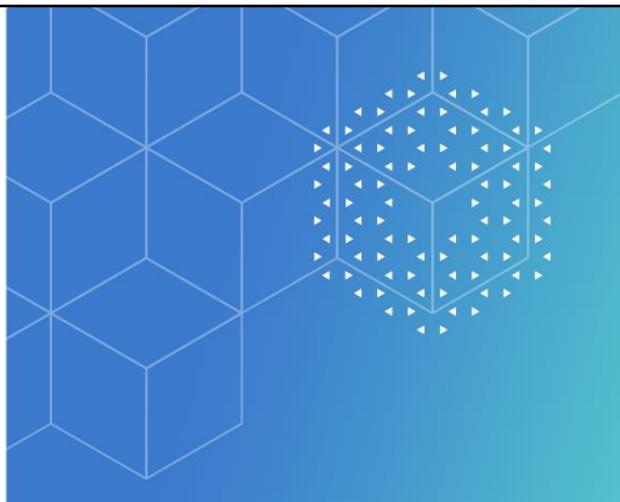
This page does not contain notes.

Human Factors

- Stakeholders who come up with or influence the feature-based plan are humans.
- It is in human nature to make mistakes!
- These mistakes should be accounted for upfront while planning as much as possible.
- Changes/modifications happen inevitably during the course of projects because of this.
- Verification plan should be flexible and mature enough to include these changes on the fly.
- MDV methodology mainly automates most of these processes.



This page does not contain notes.



Submodule 5-4

Overview of Formal Analysis Use Models

cadence®

In this module, we're going to introduce formal analysis, what formal does, how it works in general principle and the things you have to take care of in order to run the tool, and how you analyze the results.

Submodule Objectives

In this submodule, you will:

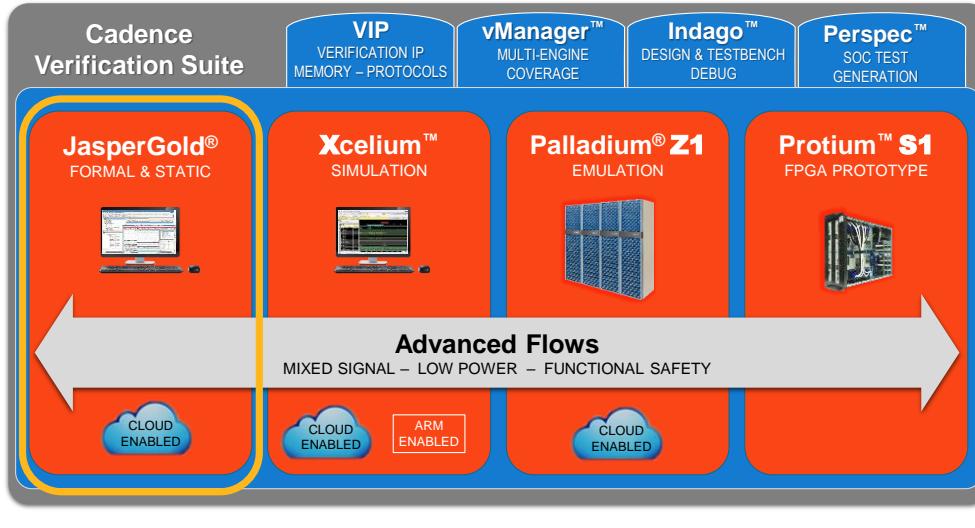
- Differentiate Formal Verification from simulation
- Use Formal Analysis for design exploration
- Use Formal Analysis to prove user-defined functional properties
- Use Formal Analysis to perform Deep Bug Hunting (DBH)
- Apply each of the different JasperGold® apps to obtain verification closure



This page does not contain notes.

JasperGold Formal Apps

Cadence Verification Suite



516 © Cadence Design Systems, Inc. All rights reserved.



The purpose of this slide is to show that formal analysis is just part of the solution to a verification problem. It does not directly replace any other methodology or tool, but it can make significant savings; for example, the use of more can drastically reduce the amount of simulation which needs to be done.

It is another tool that the Verification Engineer has to solve today's complex verification problems.

So JasperGold is part of the Cadence Verification Suite and each of these tools has to be used together into operatively in order to verify a complex chip. So JasperGold is just one of the things you will have to do to verify a chip along with all these other things. So JasperGold is not a replacement for all of your simulation. There are certain things JasperGold is good at and there's certain things simulation is good at and understanding what those are is key to being effective with it. So just as not a standalone tool, it's not more work. It saves you a lot of work in simulation. So that's the way in which to view this.

What Is Formal Verification?

Why is it called `formal`?

- This derives from `formal` logic

Reasoning based on manipulation of formulas:

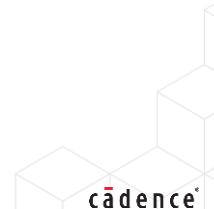
- As opposed to intuition

The name `formal` can mean many things, e.g., any of the following:

- Logic equivalency checking (LEC)
- Model Checking – Functional formal verification (Formal Analysis)
- Theorem proving

This course deals only with Formal Analysis (FA)

517 © Cadence Design Systems, Inc. All rights reserved.



Formal techniques have been around for decades.

It has lately become a mainstream verification method for these reasons:

1. Industry standard languages for defining properties.
2. Improvements in algorithms can yield results in practical timeframes
3. Software tools have become much more user friendly
4. CPUs become much more powerful and less expensive- and memory too
5. Widespread availability of commercial Distributed Resource Management software, e.g., LSF, Sun/Oracle Grid Engine, etc.

Firstly, then formal verification, why is it called formal? This is derived from formal logic, which is based upon manipulation of formulas. So basically, it's just algebra. And this is opposed to simulation where what you're doing is you're guessing. A lot of people don't like admitting this, but when you're doing a simulation, you're just guessing that if you apply certain kinds of stimulus and observe the results, that you've verified your design, but you never know with simulation because you can't simulate every possible scenario. I think most people doing simulation understand that. You can't possibly test everything in simulation because there's too many different cases to test in any reasonable kind of time frame. Now in itself, the name `formal` can mean many different things to different people. Logic Equivalence checking, which has been around for a long time, this is an example of `formal`, so this is an algebraic comparison of one design against another. For example, two versions of an RTL design, one of which has been optimized with Clock gating, for example, they're improving, which is there are no commercial tools which do this. This is more of an academic thing. There are tools, but they're all academic ones. This requires a lot of expertise and knowing where you want to end up with. With there and proving basically you propose a theorem and then you propose an addition to that theorem which you try and prove is correct. And if it is correct, then that becomes your new theorem, and you just carry on like that building one thing on top of another until you prove what you really intended to. You have to know where you're going, and it takes a lot of expertise. Now, model checking. This is what all commercial tools do. JasperGold is a model checking too. We'll talk about what that means as we go through. In order to distinguish model checking from other different kinds of `formal`, we're going to call model checking `formal analysis`.

Formal Verification Practical Issues

Writing formal (symbolic) specifications is difficult

How do I know whether I've specified:

- What do I want?
- Everything I want?
- The environment correctly?

Tools and recommended methodologies can help

These are general verification problems

- Not limited to formal analysis



Practical issues are it's difficult to know whether you specified what you really intended, everything you should have, and whether you've set the environment up correctly. However, the tool itself and any recommended methodologies help you through that. And this is what this course is about, really showing you that methodology. And these aren't really problems that are specific to formal that you could apply this to any kind of UVM Testbench as well, for example. These aren't new verification problems. They're problems that any formal verification will have.

SVA Verification Directives

Property

- A design behavior defined as logical and temporal relationships between boolean and sequential expressions.

Expression of design behavior.

Verification Directive **Keywords**

- assert** – An assertion
 - The tool must check that the property is always true.
- Assume** – An assumption/constraint
 - In Formal – The tool must only allow the DUT and inputs to behave in ways that never violate the property.
 - In Simulation – Constraints have no meaning in this context, so it is automatically converted into an assertion.
- cover** – A cover expression
 - In Formal – Demonstrate that it is possible for the expression to be true.
 - In Simulation – Count how many times the expression is true.
- restrict** – A readability intent statement
 - In Formal – Exactly the same as **assume**.
 - In Simulation – Has no meaning, hence, is ignored.

What the tool should do with the expressed design behavior.

The stated design behaviors in the SVA properties do not care if we are a simulation or formal user.



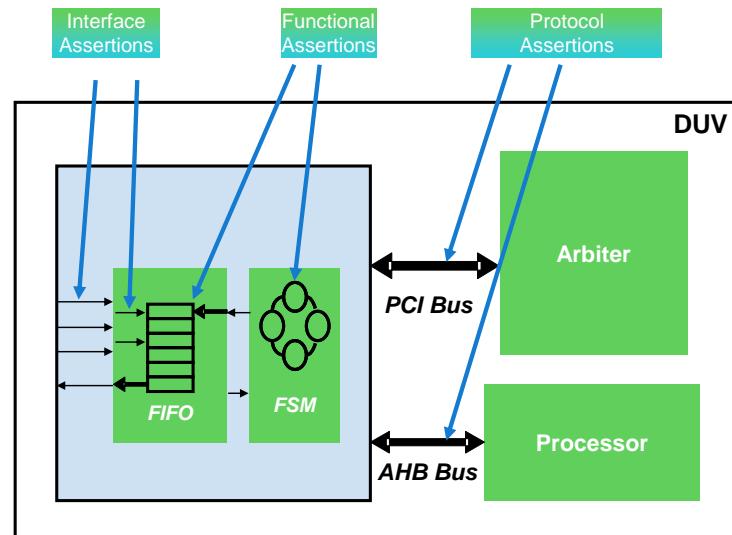
So that's all a property is a definition of behavior, and the verification directives we have in SVA or ASSERT, which means the tool must check the properties are always true. Notice none of these verification directives are specific for formal or simulation. What happens depends upon the tool you use in the course, but there is no definition in the language reference manual of whether you're doing a simulation or whether using the form. The assumption is a constraint, so in formal, what that means is the tool is only allowed to drive the inputs in ways that don't violate that property. Remember, these directives are applied to the property itself in simulation has no meaning because the stimulus is determined by the testbench. What happens in simulators? The Cadence Xcelium simulator is that these will be automatically converted into assertions. A cover in formal is an example of something being true. So normally when you write a cover, you're writing it on a sequence. In formal, when you say to cover a sequence, you're saying to the tool, show me an example of that sequence occurring, if it's possible. In simulation, it means count how many times that expression is true. If the cover was on a sequence, you would count how many times you observe that sequence during the simulation you did. Restrict is not widely used. It's a readability intent statement. In formal, it means literally exactly the same thing as assume. In simulation it has no meaning, so it's ignored. So basically, a restriction is there to use as an assumption in formal for reasons of verification efficiency in terms of throughput, it's not expected design behavior in real life. So really you can view a restriction as a temporary restriction which is only there for convenience of verification, nothing to do with the design behavior required.

Dynamic Verification (Simulation)

Assertions monitor and report:

- Expected behavior
- Forbidden behavior
- Signal protocols

Assertions are dependent on the effectiveness of stimulation vectors.



520 © Cadence Design Systems, Inc. All rights reserved.



If we remind ourselves what happens in simulation, then we write these assertions about things like, for example, interface behavior and our protocols being followed correctly. Functional assertions, like all my FIFO pointers working correctly, are my state machine making transitions through the states as I expect. And for this kind of process, which are industry standards, you will probably be using verification IP because someone's already taken the time and trouble to create assertions and assumptions and covers for those protocols. So basically, you just bind that verification IP to that bus, and then you observe the results. If you use a simulation, then how effective your assertions are dependent upon the stimulus. If, for example, you have some assertions written about AHB bus and your stimulus never causes any transactions on the AHB bus, you've never learned anything from that. You've had no failures. But that doesn't mean there's no problem because you never exercised it. In order to know you exercised it, you also need coverage, which we talk about in more detail.

Formal Property Checking

Properties are created from the design specification and implementation decisions:

- *Assertions* are properties that the tool must prove.
- *Assumptions* (a.k.a. Constraints) are properties that inform the tool how the design inputs can behave.

Formal engines verify whether the assertions hold under all input conditions allowed by constraints.

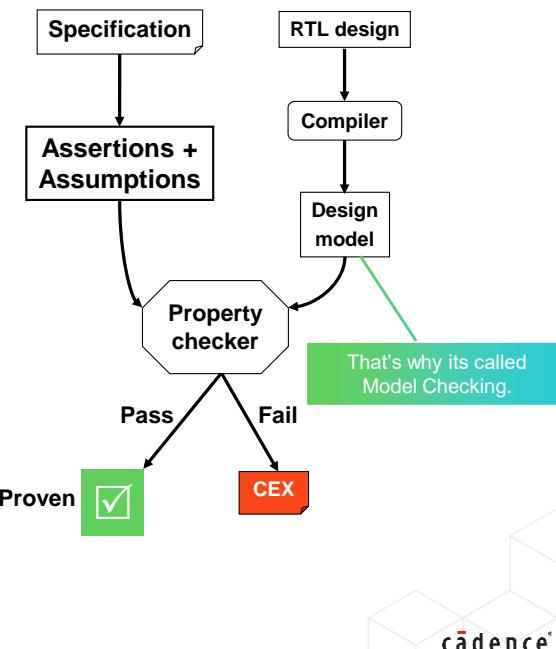
- There is no testbench driving input stimulus.

On a failure, the formal engine should provide vectors to demonstrate failure.

- Known as a Counter Example (CEX).

Properties can be defined with SVA, PSL or OVL.

- Although the vast majority (98%+) of people use SVA.



521 © Cadence Design Systems, Inc. All rights reserved.



If we're to draw a kind of flowchart very simplistic, one of which is formal verification, we will do this. From the specification we derive the assertions and assumptions we need. We create our RTL design. This has to pass through the compiler in the JasperGold tool, and from that it creates a model of your design, and then on a one-by-one basis, each assertion is proven against the design model. Is the assertion always true against the design model, using the assumptions we have? And there are two outcomes either. Yes, that assertion is always true, and that's called a proven result, or no, it's not true, and what you get is a counter-example, a waveform showing you a way in which the assertion is violated by the design and the assumptions. There's no testbench there. There's no random stimulus or anything like this. It's an algebraic proof. Is that true under all conditions? And the answer is yes. Whether it is a proven result or not, it's not. Here's a counter-example and we have to debug that counterexample because it's a waveform to understand how does my design get into a state where the assertions violated the properties that are normally defined using SVA? There are other languages that do the same thing. But most people, I'm talking about almost a hundred percent of people use SVA. That has become the dominant language. This creation of the design model here this process is actually known as synthesis in JasperGold. Not like the synthesis you might know as a designer. It doesn't mean you turn it into silicon gates from a library. It means you create a mathematical model. And how would you break the model of the design? And you measure each assertion, whether it's true or not, against that design. So that's why it's called model checking because you have that design model.

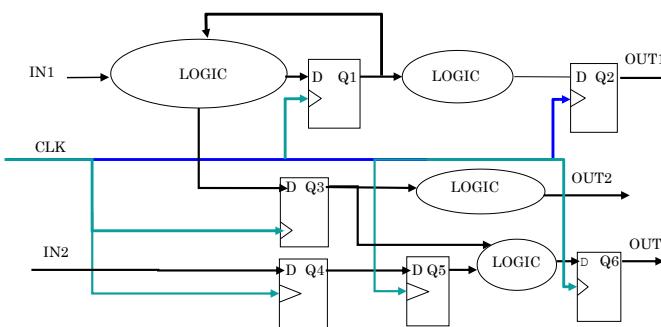
What Is Your Design?

One big state machine

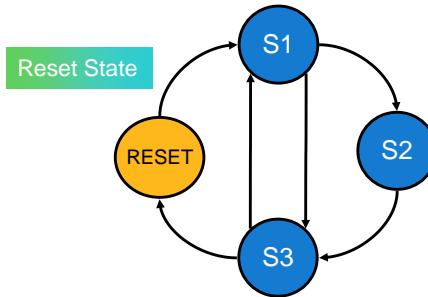
Output of memory elements determines state of the design

State on next clock depends *only* on:

- Inputs
- Current state
- The design



522 © Cadence Design Systems, Inc. All rights reserved.



State	Q1	Q2	Q3	Q4	Q5	Q6
Reset	1	0	0	1	1	1
S1	0	1	0	1	0	0
S2	1	0	1	0	0	1
S3	1	0	0	1	1	0



- You can view your entire design as being one large state machine. Each memory element, either a latch or a flip-flop, remembers a value.
- If you take an aggregate of all of the memory elements in the design, then this represents the state that the design is in.
- In a design with only flip-flops, the state of the entire design can change on every active clock edge.
- The state that the design goes into next is determined by only three things:
 - The state that you are currently in.
 - The values of the inputs.
 - The design itself.
- The combinational logic is always going to produce the same output, given the same inputs. Outputs change in response to inputs immediately as far as we are concerned for functional verification. Remember, formal analysis is about functionality, not timing.

When this model is built, then how does this viewed by the tool as far as the JasperGold tool is concerned, your design is just one giant state machine and it's the output of the memory elements which determine the state of the design. The memory elements will be the flip-flops, the latches and the memory cells. And for this giant state machine, the state on the next clock depends on what the inputs are doing, what the current state is, and how the design is defined. If you wished and there's no point doing it, but if you wished, you could draw a giant state map of the entire design, which you can imagine will be absolutely massive for a complicated design, and there'll be no point doing it anyway. But you can imagine that that's how your design is viewed by the tool.

How Many States?

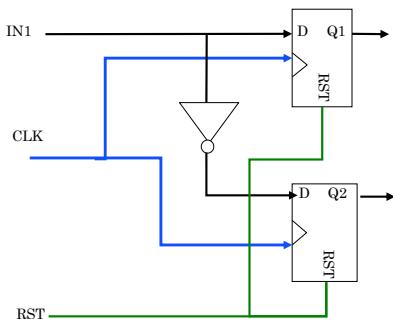
Most probably << $2^{(\text{Number of FFs})}$

Depends on the design

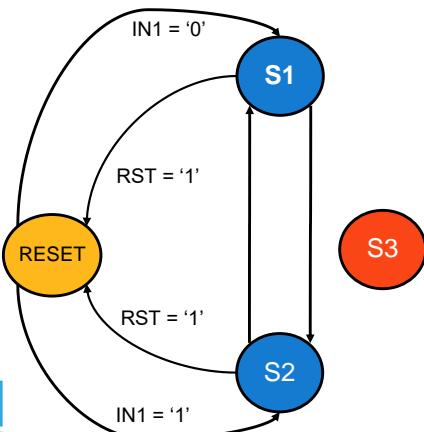
Some states may never be reached

S3 is always unreachable

- If you start from S1, S2 or RESET



State	Q1	Q2
RESET	0	0
S1	0	1
S2	1	0
S3	1	1



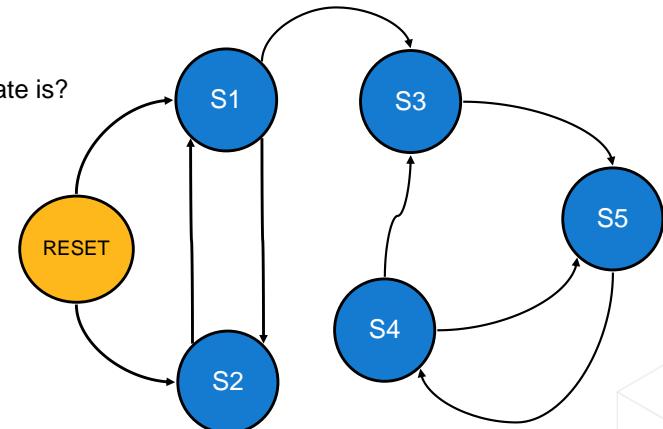
- It is highly unlikely, though possible, that the number of states that your design can be in, is two raised to the power of the number of FFs in your design.
- Your design itself determines how many states it can be in.
- In the example, Q1 can never equal Q2 after the first active clock edge has passed unless RST is active.
- The state where Q1 = '1' and Q2 = '1' can only occur if you initialize the FFs to those values. In verification, that can be done by way of initial signal assignments. In hardware, this can be if your specific technology has FFs that can power up with the Q output set to '1'.
- After an active RST or clock edge occurs, that state is never reached again.

How many states are there? Well, it's most likely that there's a lot less than two to the power of the number of flip-flops in your design, because it depends on the design. And this example shows here an example of a design where there's only two flip-flops here and there's an inverter between the inputs of these two. If we imagine that we start from some known state, which is a reset state where both outputs of the flip-flops are zero, then there is no way you can ever observe both outputs being one. So not all states are reachable depending upon the design, and there's less states than two to the power of n. This state three here, this is orphaned. We can't ever reach there.

Unreachable States

- Which states are unreachable may depend on the initial state.
- Which state you decide to start from:
 - If initial state is S3, S4, S5, then S1, S2, RESET is unreachable.
 - If initial state is RESET, then all states are reachable.
- How does the verification tool know what the initial state is?

We have to decide what the initial state is.



524 © Cadence Design Systems, Inc. All rights reserved.

cadence®

- It is not necessarily true that a state is always unreachable. It can depend upon where you start from.
- In the example, if the design ever enters state S3, then it never goes to state RESET, S1 and S2, again.
- If your initial state for verification is S3, then this means that RESET, S1, and S2 are never reached during that verification run.
- However, if you start from state RESET, then all states are reachable during that verification run.
- The verification engineer has to inform the verification tool of the initial state of the design for a particular verification run.
- <<states, reachable>> <<states, unreachable>>

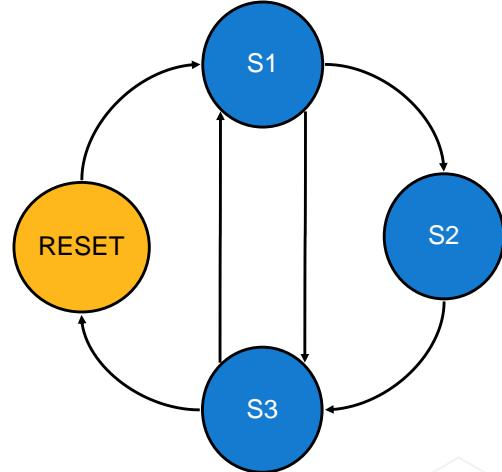
Which states are unreachable and which are not depend upon where you start from, which is called the initial state in formal. If in this example we start from the reset state, all of the stakes are reachable. There is a path to each other state. If we decided to start verification from state S3, for example, then only states three, four and five would be reachable. There's no path back to these two states, S1 and S2. The question is then how does the tool know where to start from, what the initial state is? And the answer is we'll tell it. We need to tell JasperGold what the initial state is. It's not always the reset state, by the way, either. It could be and often is, but it can be any state you wish.

Changes of State

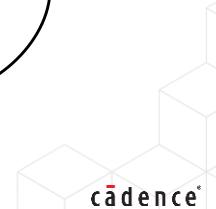
For every state a finite number of states can be reached on the next cycle.

In this example:

- All states are reachable.
- From state S2, only S3 can be reached on the next cycle.



525 © Cadence Design Systems, Inc. All rights reserved.



- In most cases, only a finite number of states can be reached on the next clock edge.
- The example shows all states are reachable.
- This does not necessarily have to be true. There can be unreachable states.

For every state that we have in this giant state machine, there are a finite number of states we can reach on the next clock. So, in this particular example, all states are reachable. If I'm in state two, I can only reach state three on the next clock.

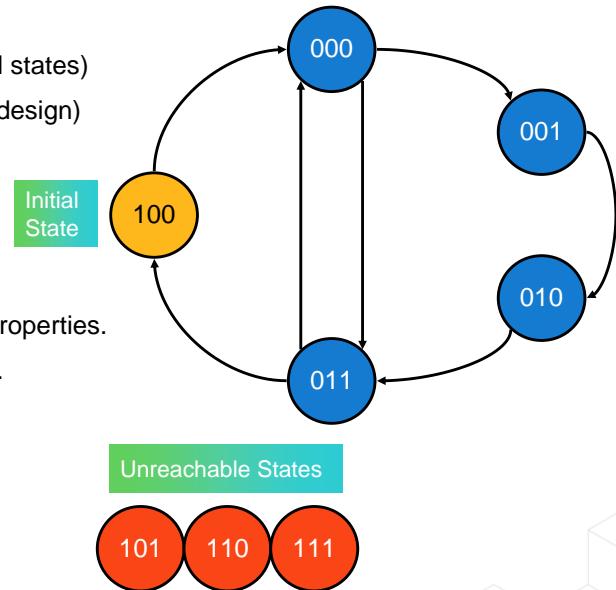
Terminology

Key terms

- State space = 8 (As there are 3 state elements, 2^3 total states)
- Reachable state space = 5 (5 being determined by the design)
- Diameter = 3
 - How many steps from initial state to all reachable states.

Observations

- Reachability and diameter depend on the initial state.
- Verification complexity depends on the design *and* the properties.
- Assertions pass if they are true in every *reachable* state.



526 © Cadence Design Systems, Inc. All rights reserved.



State space. How many possible states that the design can be in? Determined by the number of storage elements in your design.

Effectively, you can view your entire design as being one big state machine.

Reachable state space is how many of the possible states our design can ever be in.

Some states may be unreachable by design; that is, the number of states that your design can ever be in is less than two state space states.

Diameter definition: number of steps it takes to reach all the states in the system starting from the initial set of states.

A good estimate of the diameter of a design is the largest value a counter in the design can count to.

Get to the initial state by the constraints you give the formal analysis tool.

For example, typically, the design is put into the RESET state in a tool-specific way.

There is another design methodology called *hybrid verification*, where the initial state of the design considered for formal analysis is achieved by running some simulation vectors. This is useful in situations where describing constraints for a static design is extremely difficult.

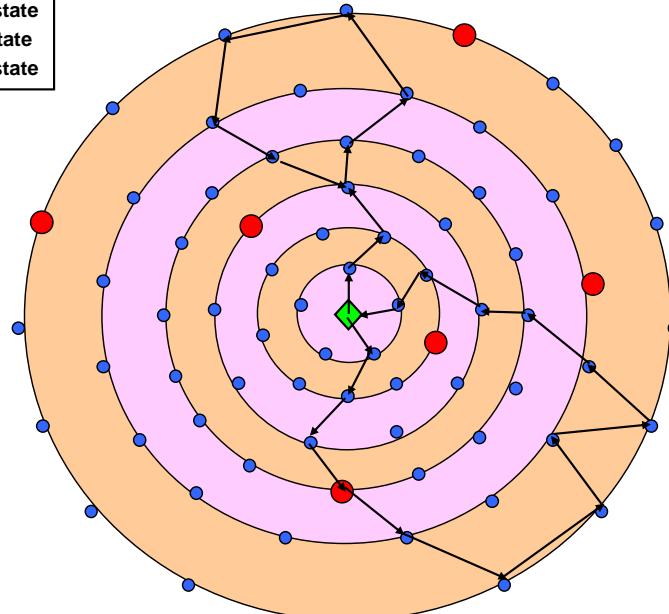
<<state space>> <<diameter>>

Key terms about this model, then are state space as if there are three state elements. Then that means that I've got two to the three total states which are possible by design for the reasons we saw on previous slides. There's only a much smaller number of states which are actually reachable. In this case we're just arbitrarily choosing the value. Five. And another crucial thing is the diameter. What the diameter means is from the initial state, which we choose, how many steps does it take to reach every other state? In this example, the answer is three, because if the initial state is here, it takes me one, two, three clocks to get to every other state I can reach every other state in three clocks or less.

What we can see from that is reachability and diameter depend upon what the initial state is, which we decide how hard it is for the tool which we're calling verification complexity depends on both the design and the properties, both of which we write. Assertions pass if they are true in every reasonable state, bearing in mind in this model we have three states we can't reach. We're never going to verify anything in those states because they're unreachable.

Formal Analysis Versus Simulation

- ◆ Initial state
- Bug state
- Good state



527 © Cadence Design Systems, Inc. All rights reserved.

Formal Analysis

- No testbench required
- Even small diameters equivalent to millions of simulation vectors
- Exhaustive
 - Uncovers all possible bugs
 - Reached max depth/diameter
- Undetermined
 - Inconclusive result
 - Reports verification depth

Simulation

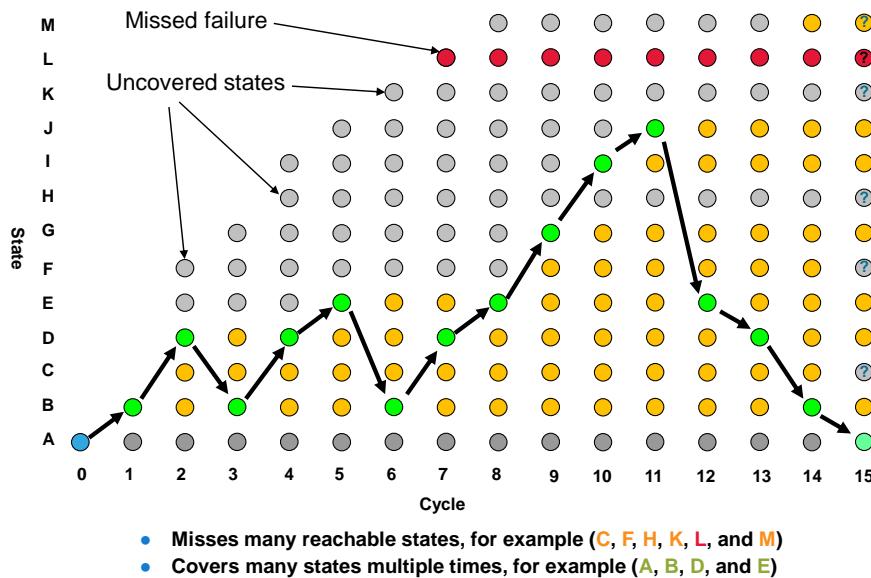
- Follows specific paths
 - Might not hit a bug
- Depends on testbench quality
- Limited controllability
 - Difficult to hit all states
- Applicable later in the design cycle



Showing a diagram, which nicely shows how formal works and affects. You start from some initial state which we decide, and on the next clock, there will be a finite number of states we can reach on the next clock based upon the design and constraints. And on the next clock, we can reach some more states. There'll be a finite number of states we can reach on the clock after that and on the next cycle there'll be more states we can reach. And this goes on and on and on until we've explored the entire state space.

The benefit of this form analysis is we don't need any testbench. There's no stimulus. Even small diameters. You even making ten clocks might be the equivalent of millions upon millions of simulation test factors. It's exhaustive. If there are any bugs in there, we find them. There's no such thing as corner cases. We will find them. And the problem with this, of course, is that at some stage the state space becomes too big to evaluate in any reasonable kind of time. You might get undetermined results. These red dots represent the bugs; all of them are going to be found by formal contrast with simulation, where what you're doing is starting from some initial state, which is almost certainly the research state. You are making a path through the state space based on what your random stimulus is doing in your testbench. You don't really know what you're doing. Even if you could see this map of states which you can't but if you could, it's still probably not controllable. You couldn't follow that specific path if you wanted to because you wouldn't know what stimulus you needed to do that. Notice you're reaching states more than once so in formal. You only need to visit each state once you know everything about it. Once you've been there in simulation, you tend to keep coming back to the same state and learning nothing new. In these simulations, we run here, two simulations, we followed a path through state space, and we might have hit a bug like we do here. We got lucky here. We hit this bug, but all these other bugs we didn't hit. So that's why people run tens of thousands of simulations runs, of tests and simulations in order to maximize the chances of hitting bugs, but you've got no guarantee. And that's the problem with simulation. You can't test everything, every scenario, and you may or may not hit a bug. And what you hope is that by writing random stimuli and having coverage, you maximize your chances of hitting states that reveal a bug.

Formal Compared with Simulation



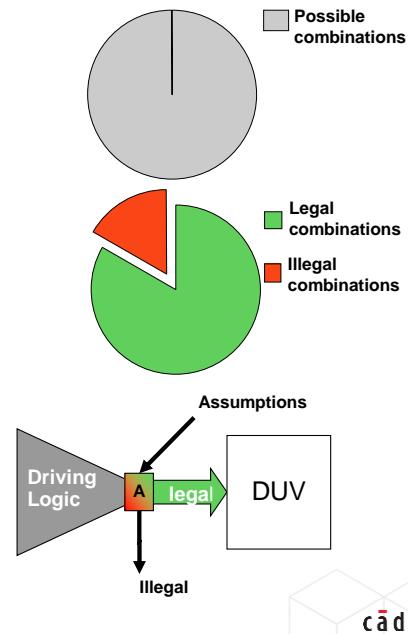
528 © Cadence Design Systems, Inc. All rights reserved.



Another way of viewing that same information is like this. We start from the initial state here, the blue dot, and on the next clock, we see how many new states can we reach. We can reach one more new state here. We don't need to go and visit the state again because we already know everything about that. On the next clock, we can reach four more new states, and on the clock after that, we reach one more new state, and so on, until we've explored all the state space. Within the simulation, what you find is you start from here. You take a path through this state space and you revisiting the same state. It's a state to be visited it one, two, three, or four times. We learned nothing new. After the first time we've been there, we learned nothing new. And notice it's really hard to get to these deep states. It's a really convoluted, specific set of stimuli that will be needed in order to reach these bugs. So that's what's known in formal as bug hunting. When you're reaching these deep parts, that simulation really struggles with. Even if you use lots of different random seeds in tens of thousands of simulations, you still can't get here other than by pure luck.

Assumptions

- Without assumptions, an FA tool checks all possible input combinations of a DUV over time:
 - Likely to get false negatives.
- Arbitrary input combinations are not legal for the proper operation of a design.
- Assumptions model the expectations of the driving logic in a larger context.
- Assumptions specify the legal input combinations of a DUV.
 - Model the verification environment.
- Assumptions exclude illegal input combinations from the verification.
- Assumptions intentionally reduce the State Space.
- Formal tools will **NEVER** violate assumptions.



529 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Legal inputs are those that we expect to see during normal operation. It is not realistic to expect your design to behave in the correct way when all possible input combinations are being applied unless you explicitly define every possible set of input combinations that your design can theoretically see.

Now for formal, we don't just need the design and our assertions, we also need assumptions. And the reason we need them is because if we didn't have them, we'd get lots of false negatives. These are counter examples which aren't valid ones. If we can imagine all possible input combinations over time with zero assumptions,

what we do is we write assumptions in order to exclude illegal input combinations. In real life you're not expecting your design to work with the inputs driven purely randomly. You know, there has to be some order, you know, inputs have to follow certain protocols in order for the design to behave correctly.

What we're doing with assumptions is modeling the circuit, the environment under which the design is actually working. A formal tool will never violate assumptions. This is crucial to understand this. Those assumptions are taken as axioms by the tall truth which cannot be questioned. Your tool will never violate the assumptions.

When you think of this as regard our state model, then assumptions are there to intentionally reduce the state space. We don't want to see all these illegal states, which should never happen in real life, because by definition they will never occur.

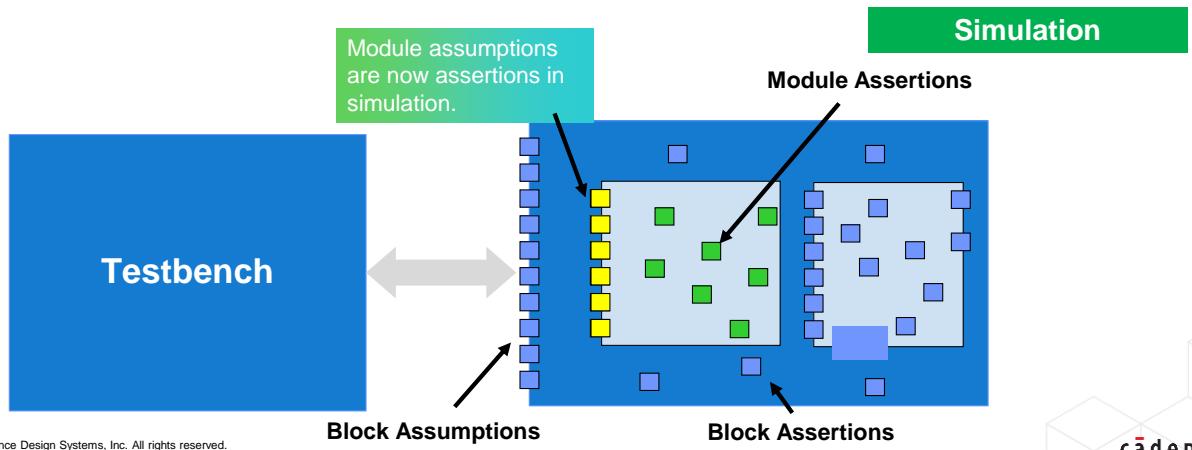
Formal Verification and Simulation of a Block

In Formal Verification, we need both assumptions and assertions, but only assertions are verified.

In simulation, both assertions and assumptions are verified.

The simulator automatically changes assumptions into assertions.

- Allows us to check the validity of assumptions we use in Formal Analysis.



530 © Cadence Design Systems, Inc. All rights reserved.



When we think about this, when we're integrating one block inside of a bigger block and so on until we have the whole system integrated, the assumptions here when we verify this block in formal, we had to write assumptions about what the inputs were doing, how we expect the inputs to behave in real life.

And when we put that block now into a bigger system, what was an assumption here is now converted to an assertion because we need to validate that these properties are still true throughout any simulation test we do. If you can imagine, if we go back here a second, if any of these assumptions are wrong, then all of my formal results could be wrong. If your assumptions are incorrect because they're taken as axioms by the tool, then all your results might be invalid. So that's the reason we need to have another mechanism for verifying assumptions. There are other things we can do as well. Assume guaranteed reasoning, for example, that's beyond the scope of this course.

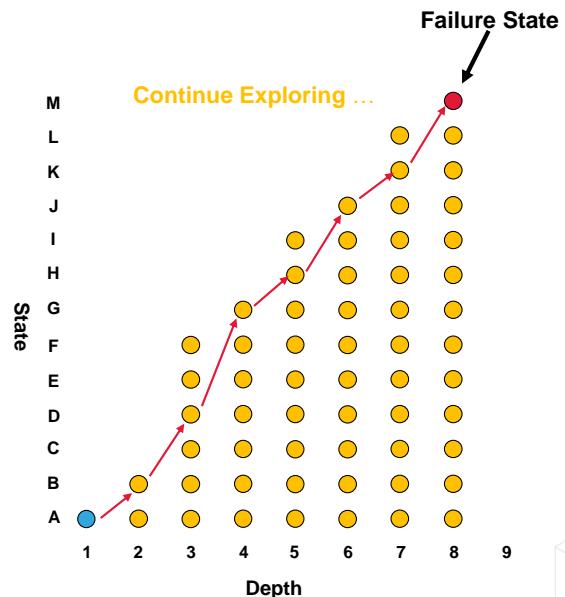
Formal State Space Exploration

Formal visits DUT states as it walks through stimulus

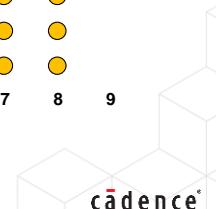
- State = value of all registers/inputs

Formal may reach states that hit properties

- Assertion failures, cover hits
- Converted to waveforms
- Usually, the waveform is the shortest possible path



531 © Cadence Design Systems, Inc. All rights reserved.



What does formal do? Going back to the similar diagram I saw earlier; we start from some initial states, and then we continue exploring all different states. How many new states can we reach on the next clocks?

As we go, we might find a state where we observe a failure. What the tool will do is it will draw back the shortest path from the failure back to the initial state and demonstrate that as a waveform to us. So, we have something to debug.

We can also continue exploring and get to the state where we find no failures at all. We know we've visited every state. We know for sure there cannot be a counter-example. So that's an exhaustive proof, a foolproof.

Or continue exploring states, and then the time limit comes in, and we know we've got more states to reach. So that's where the undetermined result comes from. No counterexamples so far, but there may be some in states we haven't yet reached.

Coverage in Formal Analysis

Cover statements express desired reachability of states of the DUV and the environment.

- In formal, with a cover statement, we are asking, “is this cover statement possible to observe?”
- Enables us to detect an unintentionally overconstrained environment.

In simulation, the same cover statement states, “count how many times this cover statement was observed.”

As users of simulation are aware:

- Checks (assertions) without coverage are of little value.

Coverage is as vital in FA as in simulation.



It is important that constraints and properties considered during formal analysis are complete and correct. Check for over-constraining or under-constraining. Check that all inputs and outputs are covered by properties, that is, that there are no holes in the coverage.

It is highly recommended to have more than one person create the properties and constraints. It is also highly recommended that sanity checks be added during the verification.

JasperGold has several tools to automatically check the validity of the user input: Vacuity checks find contradictions in the design or properties and find inputs and outputs that are not covered in formal proofs. This identifies holes in verification coverage. Incisive Formal Verifier looks at cover directives and says if the sequence that is the subject of the cover directive will never happen.

If the sequence can never occur, then the sequence, constraints, or the design must be incorrect.

<<complexity>>

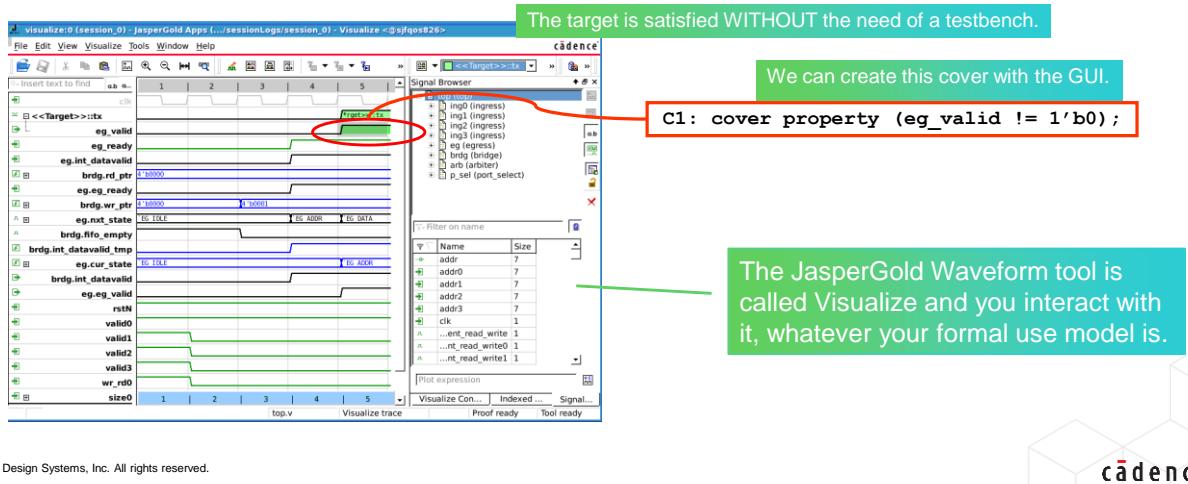
And that's why coverage is as important in formal analysis as it is in simulation. Coverage is essential. Cover statements designed to express the reachability of states, you know, the kind of behaviors we're expecting to be able to see the tool demonstrate to us. So, in formal, a cover statement means Is this cover statement possible to observe and show me a waveform of it happening? This allows us to detect unintentionally over constrained environments. And there are also some tool features which do that as well, which we discuss later. In simulation the same cover, meaning exactly the same code means how many times that cover statement was observed. The same code has different kind of information given to us depend on what kind of tool we're using, although it's the same code. So just like simulation, having checks without coverage is of no value.

Design Exploration

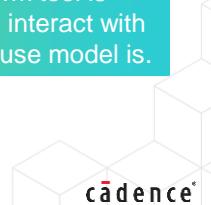
We don't even need to write any properties for JasperGold to be useful.

A cover in formal means "show me an example of this happening."

One does not need to create a testbench or think what stimulus is needed for the cover to be hit.



533 © Cadence Design Systems, Inc. All rights reserved.



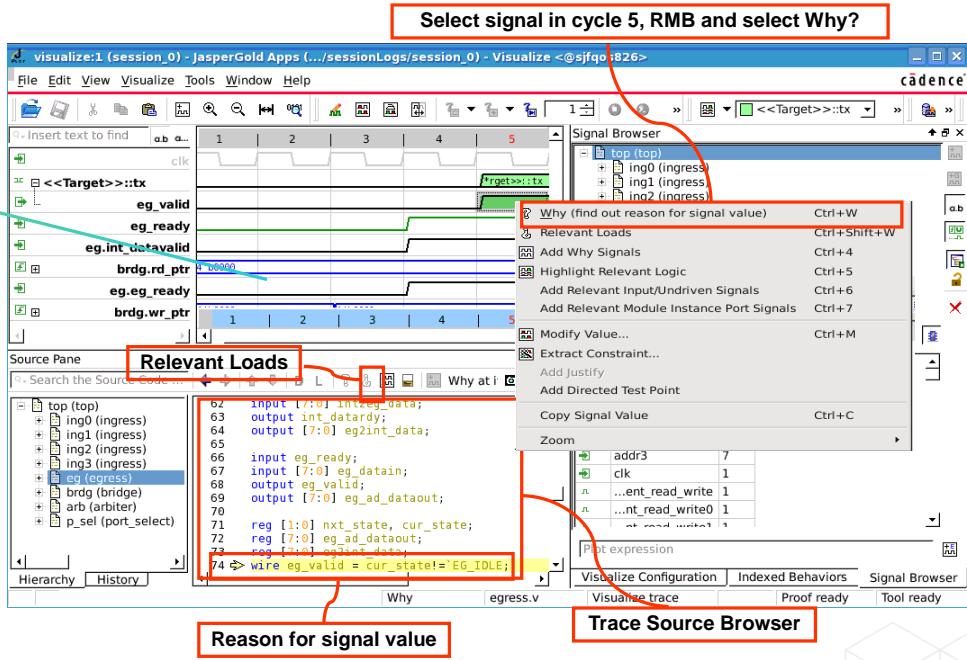
The first thing is design exploration. This is essential. If you, for example, inherit a new design from a company you've just acquired or you've changed positions in the company, and the first thing you worry about is how can I get up to speed on how this really complicated design works. Well, the answer is

You can use formal. You don't even need to have written any properties for this. If you want to observe particular behaviors of the design, like, for example, producing a burst on the outputs, you don't need to think of what corner case or stimulus do I need to exhibit that behavior. The tool will do it for you, of course, because in formal if you write a cover even manually or using the GUI, what this means is show me an example of this happening. The tool will do whatever it takes in order for this to happen. And if it can happen, then it means your designs are either not capable of it, or your constraints are over-constraining the design inputs in such a way that a burst cannot occur.

Essentially, we can satisfy some given target without the need for a testbench. And if we use the GUI to do this effectively, it's as if we wrote that in our code without having to go through the compilation and editing of files we've already seen Visualize, we talked about that on the previous section and use it in the labs already. This is how you interact with any of the formal apps that we're going to discuss. All the use models, you will be looking at tracers. These are called tracers. These waveforms commonly known as tracers.

Visualize Features: Why?

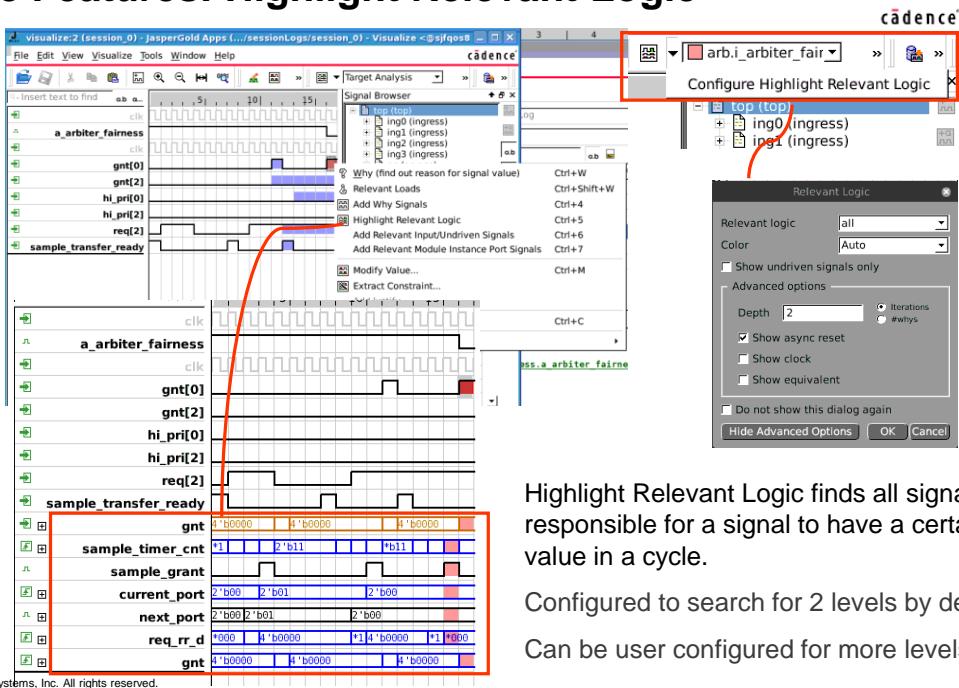
One can even modify the waveform observed by dragging signal values with a mouse pointer.



534 © Cadence Design Systems, Inc. All rights reserved.

You see those, and you interact with them via all the standard things that we've seen so far and described in the previous section, like, for example, the Y feature where you can select any signal in any cycle and see which signal assignment caused that. I won't dwell on that because we talked about this in the previous section. And what it's mentioned in there is wave at it, so you can actually modify what you see to see, can my design exhibit that behavior, or can I still reproduce a counterexample effectively with wave when you're applying constraints to the waveform?

Visualize Features: Highlight Relevant Logic

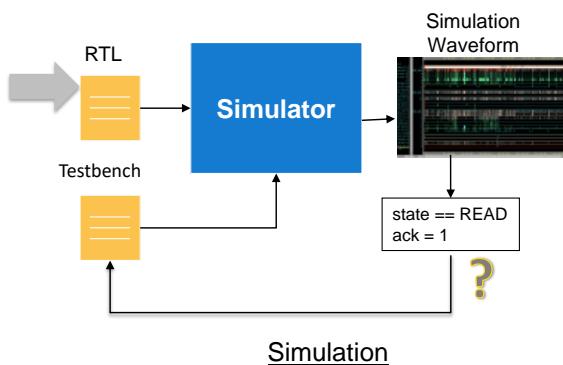


We can also know which signals actually contribute towards a particular signal having a certain value in a certain cycle via analysis of the relevant logic.

This means the logic which drives that signal is in the funding code, basically going back to some number of hierarchical boundaries, which are flip-flop boundaries.

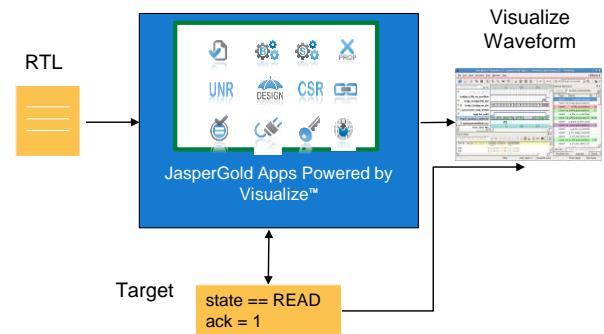
This number of hierarchical boundaries defaults to two. We can of course change that to whatever we all wish because most people using formal have come from doing verification using simulation previously.

Benefits of the Visualize Approach



Pain Points

- “Input driven” trial and error approach to producing the desired behaviors.
- May not exercise desired states.
- Need all elements of the design or must manually create stubs.



Visualize Flow

Benefits

- Specifies the target and lets the formal engines automatically generate the stimulus (“output driven” method).
- Proof (no trace possible) is exhaustive.
- Interactively adds constraints to construct desired waveform(s).



536 © Cadence Design Systems, Inc. All rights reserved.

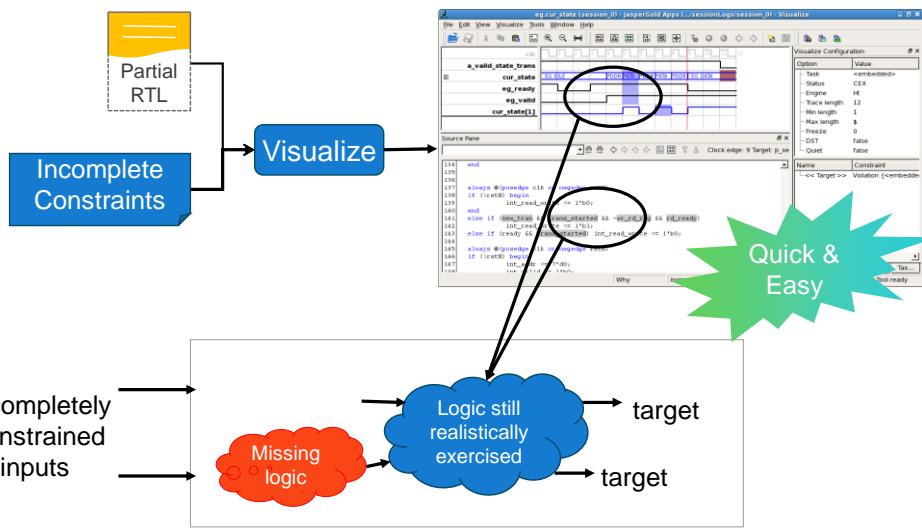
In simulation what we do, whether we like it or not, is we guess what might be a good stimulus. It's all trial and error. We create random stimuli normally in a UVM testbench and apply that to our design and observe what the outputs do in response to that.

The problem with simulation is the first thing you accept if you're doing simulation is you can't test everything and you need the entire design, or you need to go manually and create some stubs or models to act in place there. You can't do it, do it on a partial design. So that's what you're doing in simulation. Basically, these are all the difficulties with it.

With Jasper, however, it's the other way around. It's an output-based, driven method in that what you're doing is debugging counterexamples, which are failures of the properties of your design or inspecting covers in order to explore your design and work out how you ended up in the state where the cover is reached or the counterexample is seen. So basically, we get an end result and then we work backwards to work out how did we get the benefits of this method are that you get the shortest trace possible because that's what formal tools will do, in general. You can interact with the waveforms you're seeing in order to get a different one, to make debugging easier or to experiment with what you think might be RTL fixes and you can know which signals actually affect the result or not.

Partial Designs Can Be Verified

Incomplete RTL or constraints is not a show-stopper for visualize.



537 © Cadence Design Systems, Inc. All rights reserved.



You don't need the whole design because, as we've already seen, each assertion has its own unique COI, and the only part of the design you need in order to verify that property is just what's in the COI. And in many cases, you don't even need the complete COI in order to get a proven result from a property.

You can exercise your design partially, which is important because what this means is you can use formal as a design aid, not just a verification tool. You can use this while you're designing to make sure that what you've just designed literally minutes ago actually does the things, you're expecting it to do without having to think of what corner cases are or without having to think what a testbench is and how that's required and what stimulus you would generate.

What Is Functional Property Verification (FPV)?

FPV is a JasperGold app that most people think of when they hear the term “Formal Verification.”

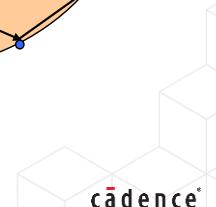
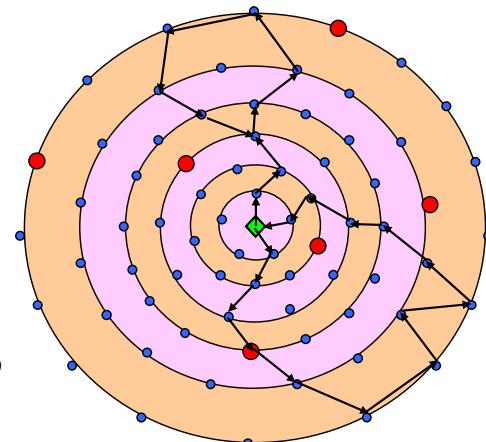
Simply, FPV:

- Writes user-defined SVA functional properties and covers.
- Gets JasperGold to prove exhaustively that all properties are always true and all covers can be hit.
- Debugs any CEXs and fixes them.
- This FPV use model is known as “Classic Formal.”

However, life and formal use models are not that simple.

At some point, the state space becomes too big for formal to prove end-to-end, full-chip, user-defined SVA functional properties.

- Without expert techniques.



538 © Cadence Design Systems, Inc. All rights reserved.

Now, virtually everything we've talked about so far and what we've done in the labs, and what we will continue to do in the labs is FPV. This is functional property verification. Basically, this means it's user-defined properties, i.e., properties that you wrote and then you wish to prove those inside of Jasper.

So simplistically, this is what we do. We write our functional properties and covers. So, the functional properties mean the asserts and the assumes. We get Jasper to exhaustively prove that all those properties or is true under all circumstances and all covers can be hit. And if we observe counterexamples or unreachable covers, then we go and fix them. This is what's known as classical formal.

This is almost like a pejorative term, meaning that this is kind of a simplistic way of using it and more advanced users today are using formal for more things like that, particularly what's called deep bug hunting. Getting an exhaustive proof on every single asset you have is not normally a requirement of every formal user. Although it would be nice if you can achieve full proofs on every proper thing.

At some point, your designs will get bigger enough. And remember, the number of transistors you can get in any given area virtually doubles every 18 months. The thing you'll never catch up with is that curve of complexity versus how hard it is to actually verify the state space will always be too big for end-to-end formal checks without some radical discovery in the processing of these formal algorithms. Of course, you can negate that to some extent with expert techniques.

But however, what we're saying here is if you have an entire chip, a big SOC and you write somewhere the assertion that says, if I put these inputs in the design, then this is the output, so I should get out the design. without expert techniques, it's going to be very difficult for the tool to prove that in any realistic time frame. Because the state space is so big, the CLI is likely to be huge. In this use of the classical formal we started at the initial state here and then we jumped out one cycle to all the states that could be reached the next clock and then we jumped to all cycles that can be reached on the clock after that and so on until we've exhaustively covered the state space.

JasperGold FPV Use Models

Property proving is exhaustive.

Bug hunting is not exhaustive – aka Semi-Formal – one can find bugs but not prove properties.

		Property Proving		Bug Hunting	
Use Model		Classic Formal	Analyze exhaustively one cycle at a time	Cycle Swarm	Run formal search at specified cycles
1. Formal to complement simulation	Goal	Prove some functions	Find corner-case bugs (counter-examples – CEXs)	Bugs found (number and “quality”)	Find deep bugs/deadlocks plus increase coverage
	Metrics	Proof core coverage			
2. Formal signoff	Goal	Prove functionality		Bugs found plus combined coverage in vManager	Bugs found plus combined coverage in vManager
	Metrics	Combined cov in vMgr			

539 © Cadence Design Systems, Inc. All rights reserved.

cadence®

It doesn't mean you can't use formal at the whole level as we've seen the property proving we talked about there previously

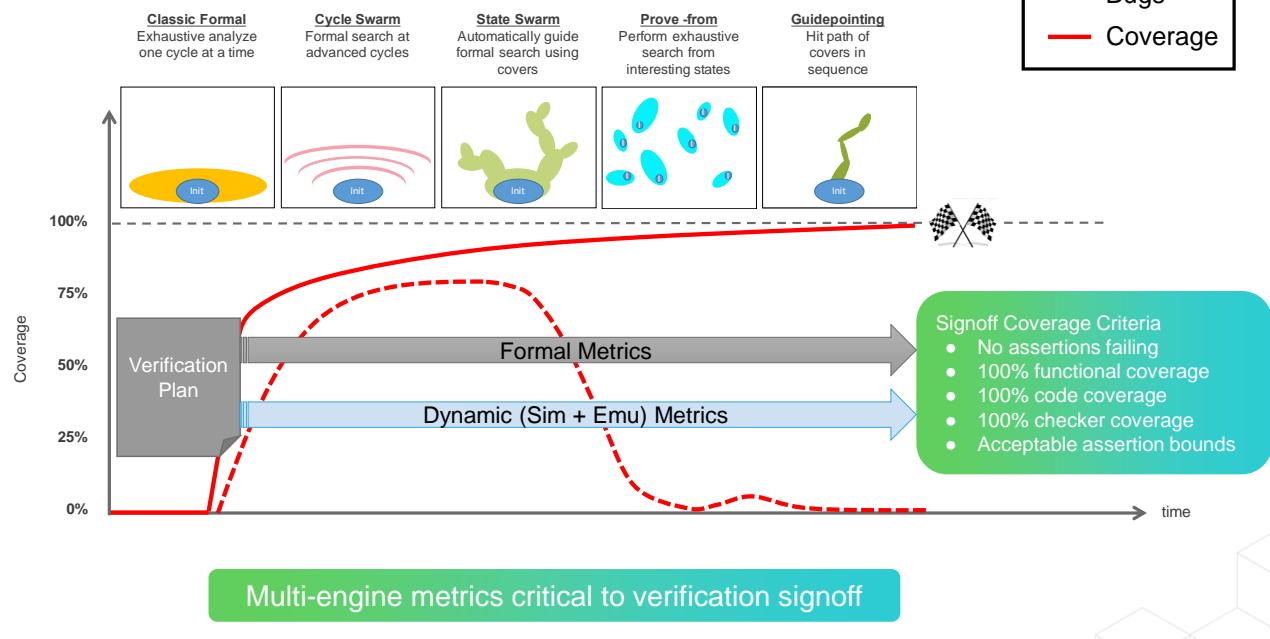
was the classic formal as we discussed, we exhaustively verify something, so we know it's true under all circumstances. However, what we can do instead, because we accept that the whole design is too big to exhaustively prove we can adopt these different strategies

like bug hunting is what this is called. So, our intent is we realize we can't exhaustively verify everything because we're applying these different techniques, which means that's all reason why we get results better and quicker because we're no longer exhaustively verifying. The kinds of things we can do, we can jump over cycles. Remember we had that concentric diagram of rings back a few slides ago now shown here. With Cycle Swarm, we're jumping more than one ring at a time basically to try and find bugs quicker. For state swarm, we start from some initial state and then what we do is we guide the formal search using covers. On this path through the state space that is following a cover that we write, the tool is doing exhaustive proofs but only for a limited number of cycles from that path, and guide pointing allows us to step from one kind of guide point to another to another following this trail of breadcrumbs in order to reach some really deep and hard to reach state. There are all these various techniques that we've got, all of which are automated, using this hunt command. If you search the support site for the hunt Tcl commands for Jasper, you'll see various videos showing you how to use that.

The use model here is for formal to complement simulation. We want to prove some functionality; the rest will be proven in simulation. The same is true for coverage. Find corner-case bugs, which are really hard to find in simulation because they're so deep. It requires such a convoluted sequence of stimulus in order to reach these deep states that in simulation, even if you do tens of thousands or even hundreds of thousands of simulations, you're very unlikely to ever get there. That's why a lot of formal experts use this bug hunting technique.

For formal signoff, what we have to do is prove all functionality. And for the coverage, we combine the coverage using VManager from other verification methodologies like emulation simulation and so on.

Technology + Methodology to Achieve Signoff



540 © Cadence Design Systems, Inc. All rights reserved.



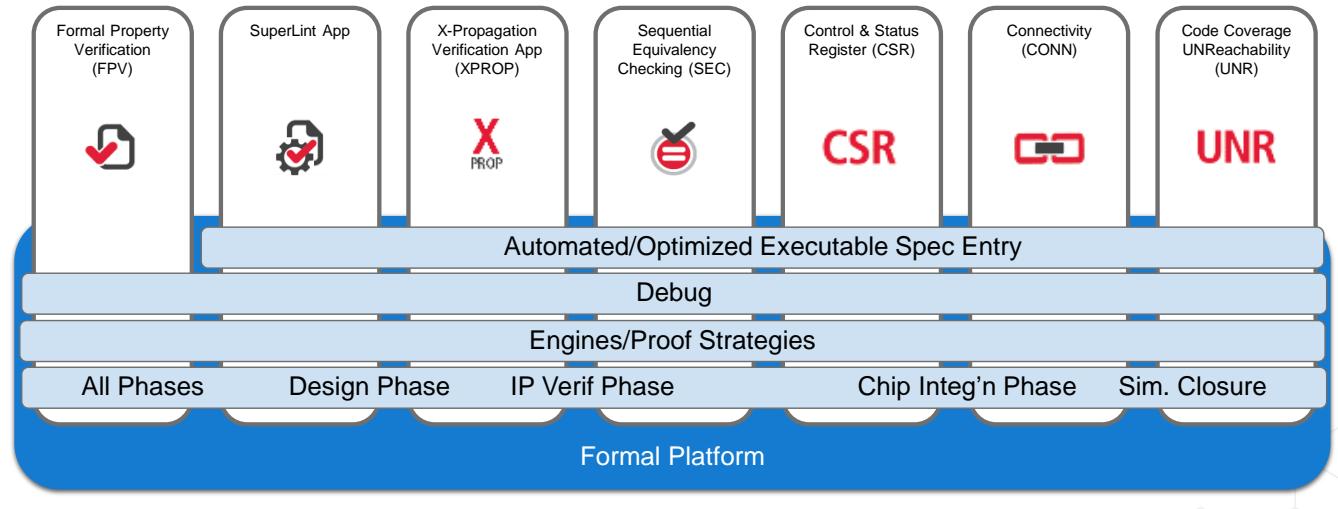
This is our goal in any kind of verification,

we require that we achieve 100% coverage without any checks failing. What's likely to happen in all projects is that you start from some initial stage where you start verifying against your verification plan, and you start hitting the wall on these bugs. As you do more and more complex kinds of tests and this bug peaks and then eventually it comes down and as you keep adding new features and correcting or removing features or enhancing the design, and somehow you get a peak of bugs again until this goes away, and hopefully over time, this kind of gets as close to zero as possible. We can use a combination of all of these things from using formal, and this one here proved from this is an option in the prove command. What this allows us to do is once we've reached some interesting state, i.e., by a cover, we can exhaustively prove from those each of those interesting states. If we had a whole bunch of covers, for example, we could say prove some assertion from each one of those covers. And this is what this diagram is representing here.

The crucial thing here is we can't do everything with just one single kind of engine. We need, as in verification tool. We need metrics from formal metrics from simulation, metrics from emulation, and hardware acceleration.

JasperGold Formal Verification Apps

Automating Common Verification Tasks



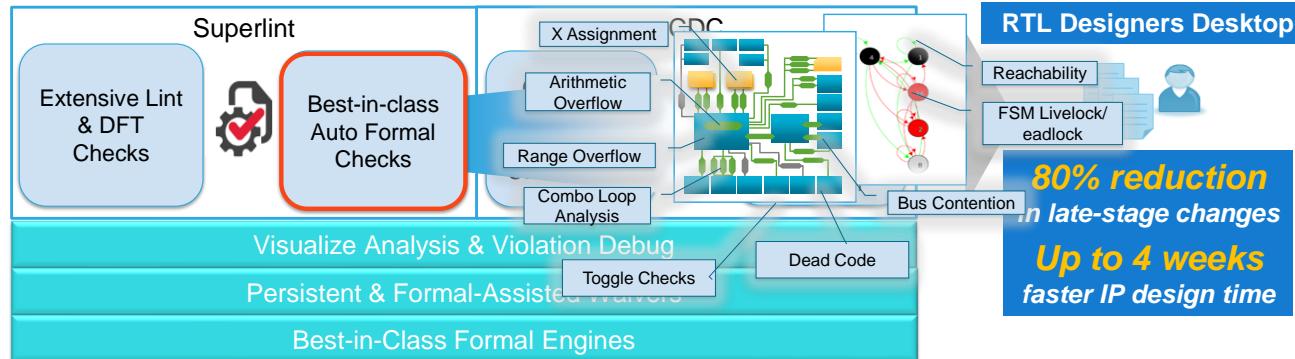
541 © Cadence Design Systems, Inc. All rights reserved.



Jasper has a variety of apps. And each app solves one problem that everybody has. It's a common problem everybody has, like, for example, X-Propagation. I'm going to go through all of these ones now and just give you a very, very brief, the briefest of overviews of what they actually do.

JasperGold RTL Designer Apps

Shift-left: remove bugs before handoff to verification and implementation.



- Common JasperGold front end with leading SystemVerilog support and capacity.
- Leverages formal technology to reduce noise and automate waivers.

542 © Cadence Design Systems, Inc. All rights reserved.



80% reduction – an endorsement from Arm when we launched

Up to 4 weeks per IP – an endorsement from STMicro

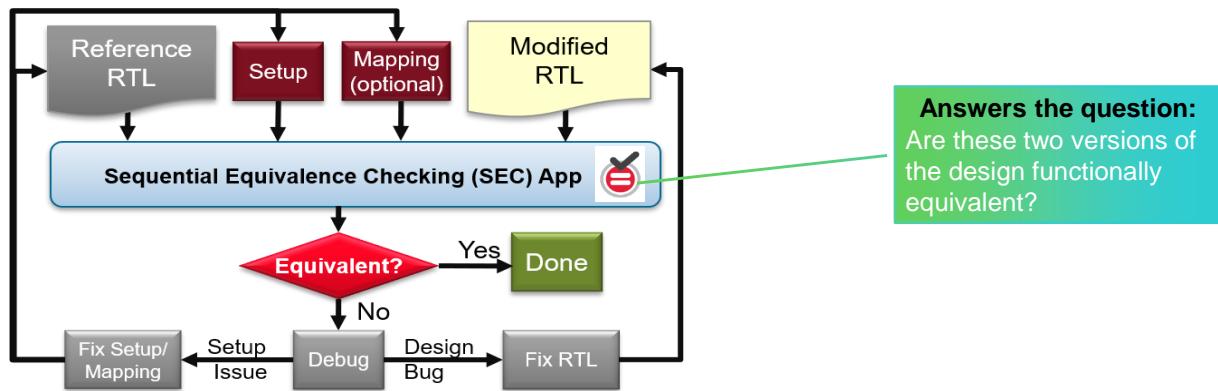
First thing is the apps which are intended for designers. Superlint is one of the apps we will actually talk about in detail and do a lab in this course. What Superlint does is getting rid of bugs before we hand off the design to verification and implementation. And we also use this as a baseline for regressions every time we do a check in on the project. Normally, companies will run super lint overnight to make sure no bugs got introduced. The kinds of things this does is it's like a static linting tool, just like how in fact it is how as in the Xcelium Cadence simulator. So that's the structural checks, for example, naming conventions and these kinds of things. Also, the DFT checks, design for test, and also the part that's of interest to us is these auto formal checks. Basically, the tool will read your design, this superlint app, read your design, and it will analyze what the kind of structures exist in the design I create automatic formal checks based upon them and run them for you.

The kinds of checks you get are, can you reach an assignment? Is it possible to get arithmetic overflow? Can you get range overflow when indexing arrays? Have we got any combination of loops? Toggle Checks are a vital part of verification. Can every signal change from 0 to 1 and back to zero again, for example? That code is every branch of code executable, bus contention, Can I get a floating bus? I'm driven or can I get a contention on the bus? And for state machines, you have all different checks like reachability and deadlock and live lock.

You don't need to know anything about formal or just for that matter, in order to run this, the tool will create these checks for you. So, it's very simple to do as we're going to see.

Sequential Equivalence Checking App

Accelerates design convergence



Sweet-spot use cases:

- Clock Gating Optimization
- Pipeline Retiming

543 © Cadence Design Systems, Inc. All rights reserved.



Another very commonly used app is Sequential Equivalence Checking. The difference between Sequential Equivalence Checking and Equivalence checking, i.e., without the sequential on the front, is that this checks the behavior is equivalent at I/Os rather than mapping each internal state element. So that's what sequential means in this term kinds of things people use this for is, when they introduce clock gating to reduce power consumption. The introduction of the clock gating changes the functionality of the design. If they're trying to retime a pipeline, for example, or implement things like multipliers in a different way, is a design still functionally equivalent to what they had before? You compare an implementation with a reference design in effect. You might compare, for example, RTL against gate level and on other kinds of things as well.

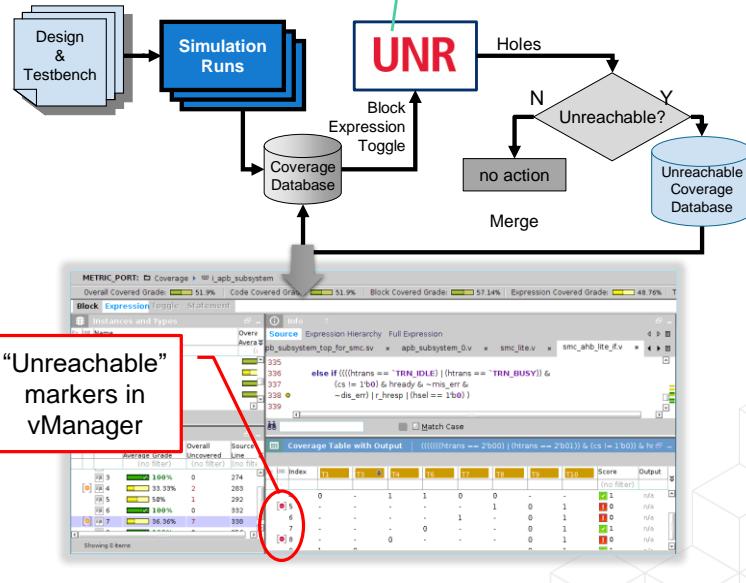
So basically, we're trying to check that there are two versions of the design that are functionally equivalent under all circumstances.

Coverage Unreachability App

Saves simulation users weeks of time and effort for verification closure.

- Inputs: Simulation coverage database and RTL.
- Output: Unreachable cover points database.
- Run by simulation users without formal expertise.
- Integrated with vManager to clearly show unreachable coverage points.
- Resilient compilation with Xcelium.
- Supports all Xcelium modeling languages and setup.

Answers the question: Do I have simulation coverage holes because my testbench doesn't exercise the design or because the coverage is unreachable, ever?

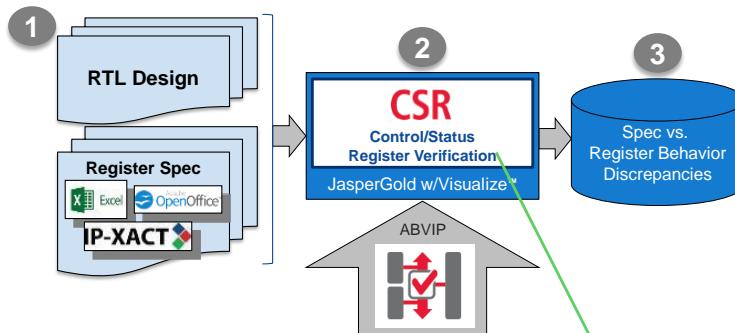


544 © Cadence Design Systems, Inc. All rights reserved.

Another one is another app, and you noticed a feature that a lot of these apps is you don't need to know much about SVA or formal or JasperGold to use them. This app is intended for simulation users. And the reason people use it is because it saves weeks of time typically. As you can imagine, you've got a big suite of simulation tests, and your idea is to get to 100% coverage, and yet you aren't filling your coverage fast enough. Then what you can say to yourself is, is the reason I'm not getting this coverage because the design is never capable of it or that testbench is never capable of driving the stimulus to reach that coverage? Or do I have a bug that prevents the coverage? What happens here, then, is after all your simulation tests, you can dump a database of your coverage, load that into Jasper, and Jasper will create cover statements for the unreachable. And what it will tell you, what Jasper will tell you is, is it possible ever to reach these covers? Because formerly, of course, you can prove that. And if the answer is yes, they are reachable, then you can do two things. You can either sign it off and say yes, it is reachable. I've verified that in formal, or you can modify your testbench, which is not exercised in design in enough ways to reach that coverage. So, either way, you make the decision on what you do next. But the important thing is, you know, whether it's reachable at all or if it's never reachable, this is what Jasper will tell you. So that's the question it's answering. Do I have simulation coverage holes because of my testbench, or do I have some kind of bug that prevents it from ever happening?

CSR Verification App

Exhaustive verification with reduced setup and runtimes.



1. Inputs: Register spec and your RTL design.
2. Run the JasperGold Control/Status Register Verification app.
 - App automatically derives and generates all properties.
 - Automatically runs formal engines to prove all properties.
3. Output: CEXs show discrepancies between the spec and RTL.

Answers the question:
Do all of my register accesses always obey the access policies in my spec?

545 © Cadence Design Systems, Inc. All rights reserved.



The Control Status Register Verification app is commonly used as well. At the highest level where you're trying to verify your kind of registers, there'll be probably tens of thousands of them.

So normally, you specify these in either some kind of spreadsheet or in IP exact, which is a kind of language for describing these things. And what the tool will do for you is once you've got this exact description; for example, this will guide the access policies. For example, some registers should be readable but not writable, and so on.

What the tool is proving here is generating these transactions in order that you can say, do all of my register accesses, and always obey the access policies defining my spec, which comes from either the IP exact model of that or an Excel spreadsheet.

Connectivity Problem: A Tangled Web

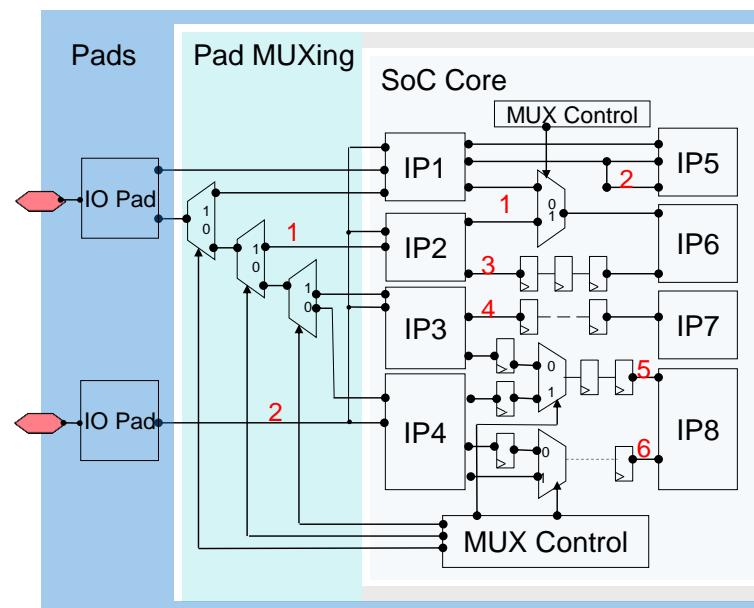
Connection Types

Combinational

1. Direct
2. Conditional

Pipelined

3. Fixed latency
4. Variable latency
5. Fixed latency and multiplexing logic
6. Variable latency and multiplexing logic



546 © Cadence Design Systems, Inc. All rights reserved.



Connectivity is a big problem because in reality you've got so much functionality you could get inside of a chip, but you've only got physically so many IO parts that you can actually have. In order to get signals out of the chip, you've got to do lots of switching between IP blocks and the outputs. And what you need to confirm is that the output of an IP block ends up at the correct IO pad under the correct circumstances. And as you have thousands of these connections to check tens of thousands possibly, you may have something like, you know, one or 2000 different pads on your chip, you know, as high as that. It's a very exhaustive task to do inside of simulation because not only is there a large number of them, but you also need the controller's ability to be able to know what the switches on these MUXes are set to on any given cycle. This is overwhelmingly difficult to do. And the kinds of connections, there's a direct connection, ones which are made under conditions pipeline with a fixed or variable latency. These are the problems.

The Key: The 3 Cs

Proving all connections in the chip are correct.

- In a typical chip there may be 10,000+ connections to verify.

Traditional simulation methods lack an efficient way to exercise the connections.

- Significant manual effort to apply stimulus.  **Controllability**

Very difficult to assess coverage in a simulation environment.

- Have you really exercised all combinations?  **Coverage**

Finally, it's a big challenge to identify whether all connections are tested by the testbench.

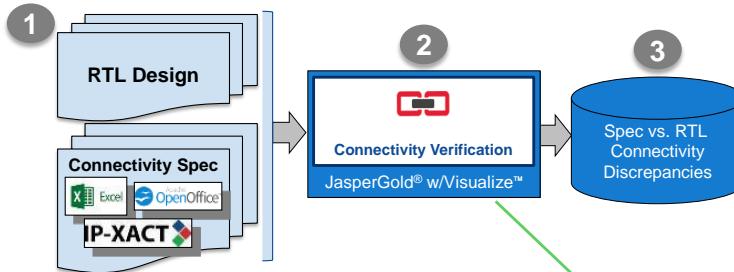
- Do you have checks for all connections?  **Completeness**



Then in that connectivity, prove all connections are correct. Controllability is the problem in simulation as well as the sheer volume of the number of connections. Very difficult to assess coverage because how do you know you've really exercised all possible combinations? You probably got it wrong if you define it manually. And to know whether you've actually tested them all as well, completeness, did you actually test all the connections? You could write some coverage, but is your coverage complete? So those are the problems.

Connectivity Verification App

Fast, exhaustive verification with lower setup and maintenance.



1. Inputs: Static and temporal connectivity spec and SoC RTL.
2. Run the JasperGold Connectivity Verification app.
 - App automatically derives and generates all properties.
 - Automatically runs formal engines under the hood to prove all properties.
3. Output: Any CEXs are connectivity errors.

Answers the question:
Do all of my connections
match my specification, and
did I specify all connections?

548 © Cadence Design Systems, Inc. All rights reserved.



This is what the connectivity app does.

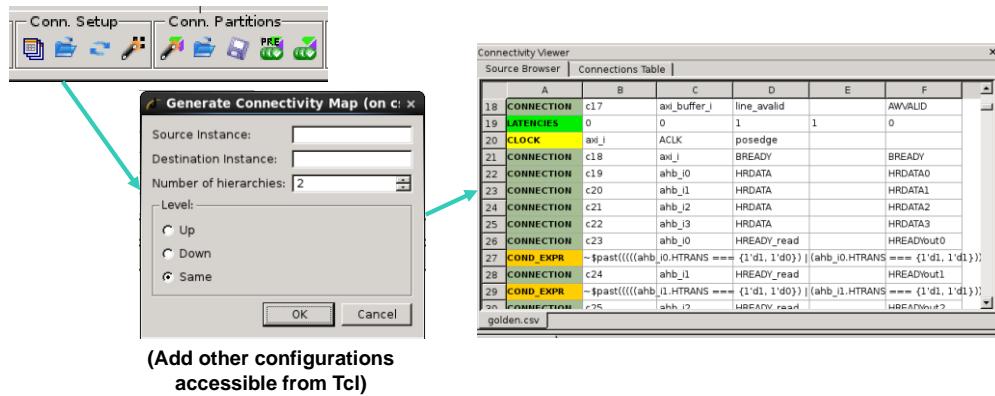
What you have is some kind of connectivity spec that comes from either an Excel spreadsheet or IP exact.

You run this connectivity app, and it will create checks for all of these connections to make sure they're made under the correct circumstances, and you get reported and your discrepancies into as a schematic.

This is answering the question do all of my connections match my spec? And also, an equally important thing is, did I specify all the connections? Are there so many of them? It's easy to miss one.

Built-In “Reverse Connectivity” Capability

Generate connectivity map from a golden RTL and re-verify it in a revised RTL.



549 © Cadence Design Systems, Inc. All rights reserved.



So the second aspect, did I specify all connections dealt with in what's called reverse connectivity? You can generate a table showing you what all of the connections actually are in your RTL.

X-Propagation Verification App

Automatic formal checks with X-Aware analysis and debug.

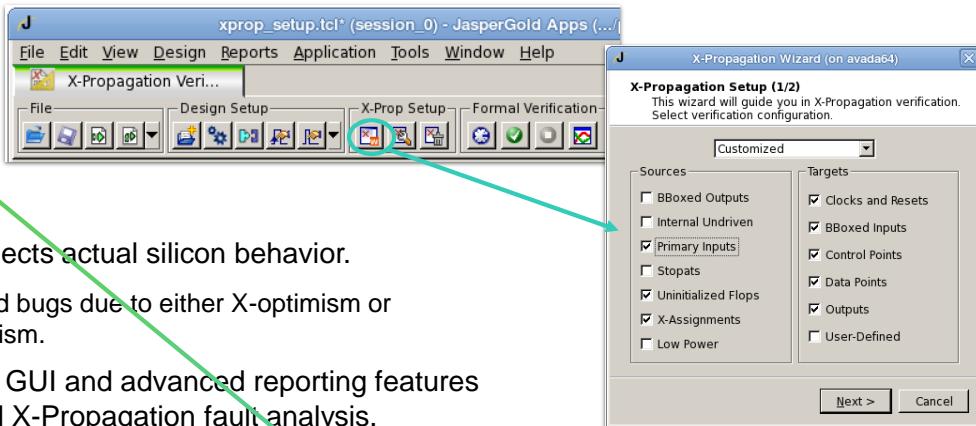


Analysis reflects actual silicon behavior.

- No missed bugs due to either X-optimism or X-Pessimism.

Customized GUI and advanced reporting features enable rapid X-Propagation fault analysis.

Requires no knowledge of formal or SVA.



Answers the question:
Can X's reach destinations in my design where I don't want them?

550 © Cadence Design Systems, Inc. All rights reserved.

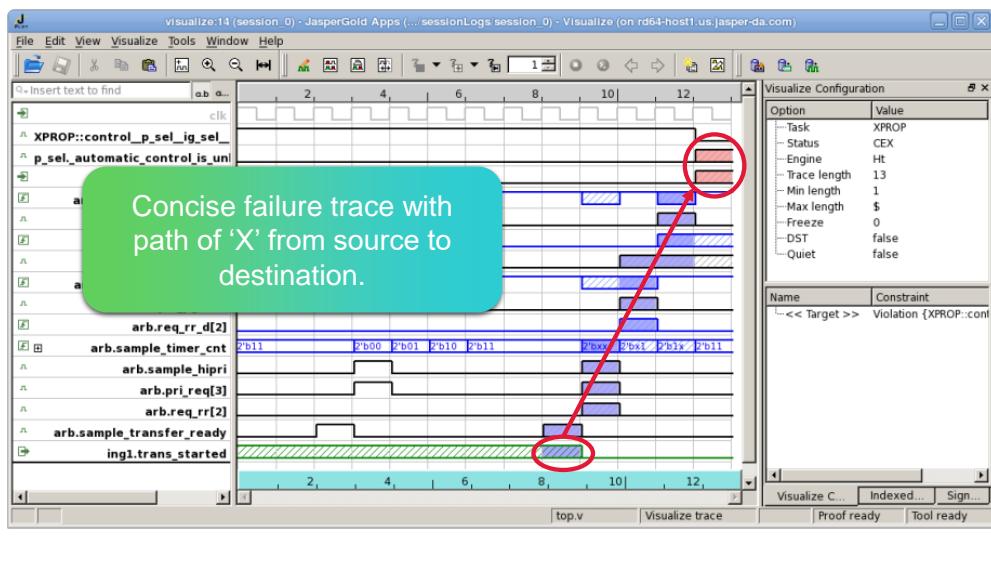


This is an app that we talk about in more detail and also do a lab on.

This is a problem all people can access; reach places in my design I don't wish them to reach.

And the problem with this kind of thing in simulation is you have this concept of pessimism and optimism on X, depending on how X-Propagates, which you need to decide which one do I choose? You don't get this problem in formal because, literally, the value X does not exist. It's either zero or one. It's a set of values that it could be, so formal deals with this as the hardware would. We just select the sources of X that we wish to consider and the destinations we don't want it to reach. And then, the tool will create these checks for us.

XPROP CEX Debug Using Visualize



551 © Cadence Design Systems, Inc. All rights reserved.

cadence®

And give me a waveform in visualize. And the shading here shows me this is an X value, so I can see an X is reaching this output here, and I can follow back all of the signals that all just and in all the signals where the X is propagating through in order that I can debug this and understand if it's a real problem or not and how to fix it.

Security: Functional Versus Data Propagation Requirements

FPV App or SEC App

Functional requirements specified by assertions (SVA)

Security Path Verification (SPV) App

Data Propagation requirements & restrictions (high-level rules)

Examples

- CPU must not be interrupted if running secure code.
- System must be reset if an environment monitor trips.
- FSM must never transition to SECURE after reaching TEST or DEBUG states.
- Other JasperGold apps relevant to security:
 - CSR app verifies integrity of register access policies.
 - FSV app models direct attacks on internal HW circuitry (laser, radiation, electrical, thermal, overclocking, etc.).

Examples

- Data in a secure area must not be visible to the CPU if it is not in a secure mode.
- Secure register must not be written by a non-secure agent.

552 © Cadence Design Systems, Inc. All rights reserved.



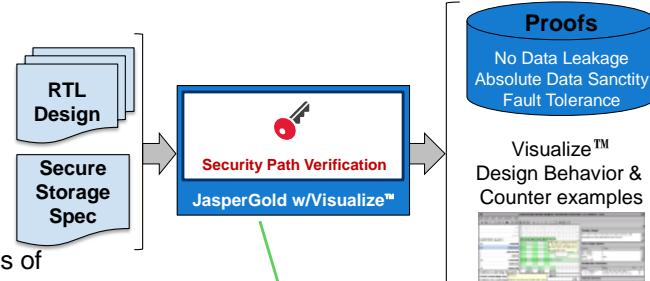
What's becoming increasingly important for people is security as well. There are various aspects to security. For example, a CPU cannot be interrupted if using a secure code system and must be reset if some environmental monitor trips. The state machine should not transition to secure after reaching test or debug states and thus functions as regards data propagation.

So, if we have secure data like keys for encryption, these must not be visible to the CPU if it is not in a secure mode. And for example, any secure data should not reach an interface that is not secure. These are the kinds of things people check for with the SPV app. Security path verification is what SPV stands for.

Security Path Verification App

Formally Prove Secure Data Cannot Leak

1. Inputs: RTL and specification of the secure storage element.
2. Run JasperGold Security Path Verification app:
 - App automatically derives and generates all properties.
 - Automatically runs special path analysis, optimized formal engine under the hood.
3. Output: CEXs show data leakage, violations of data sanctity, or vulnerabilities to tampering/faults.



Answers the question:

Can secure data reach destinations in my design or interfaces where I don't want it to?

The inputs are the RTL and the spec of which things you consider secure. You run the app, and it will automatically derive the properties, and you can run them. And the counter-examples are showing the data leakage and any violations of the secureness of the data. The essential question being answered is, can I secure data, and reach places in my design where it shouldn't? Or can my design exhibit behaviors that it shouldn't?

JasperGold Functional Safety Verification Use Models



FSV
batch

Fault Testability Analysis

- Structural Fault Testability, Activatability and Relation analysis
- Automated pre-qualification flow for Xcelium Safety, no user intervention
- Reduces the number of fault simulations

Fault Propagatability Analysis

- Formal Propagatability and Activatability analysis for Xcelium Safety
- Interactive debug, schematics and visualization of propagation
- Assists fault analysis sign-off with Xcelium Safety

Fault Safety and Security Analysis

- Custom strobes and faults specification to model hacker attacks
- Advanced formal checks and multiplicity of faults
- Addresses safety and security hardware qualification

Formal Fault Testability

Fault Sim Xcelium Safety

Formal Fault Propagatability

FSV Standalone

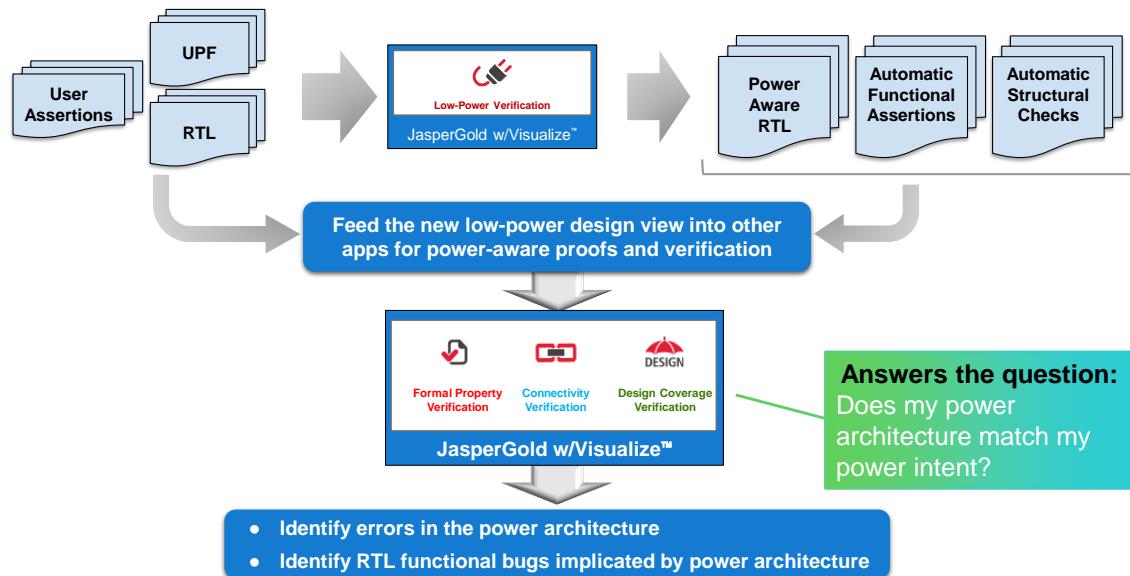


Answers the question: Can all faults be detected?

554 © Cadence Design Systems, Inc. All rights reserved.

What's also very popular or essential rather, especially in the automotive industry, is functional safety. What we need to know is whether can we actually prove that faults will propagate to where they can be detected. Is it possible to have a fault that cannot be detected? The FSV app will answer those questions for us. Can we detect and propagate all faults to places where we can observe them? And we're doing that formula, of course. So, we don't need to think of all these different simulation test cases, which we would have to do otherwise. This number of false simulations is not just an RTL simulation. Still, during the simulation, you have to corrupt certain values in order to see, i.e., insert errors in order to see if you can detect that fault. So that's what you can do formally a lot more effectively.

Low-Power Verification



555 © Cadence Design Systems, Inc. All rights reserved.



Low-Power Verification, given that you define your power domain architecture using a language called UPF, that's another IEEE standard. Does your actual implementation match that description? This is what the FPV app does. Does my power architecture match my power intent?

Formal Coverage

What Formal-Specific Questions Need to Be Answered?

What code or functionality has been “exercised” by the formal testbench?

- During classic formal or bug hunting
- Covers may be reached, undetermined, unreachable



Answers the question:
Is every part of my design exercised meaningfully by an assertion or a cover?

Is my formal checking complete?

- Properties may be proven, but how much of the design do they cover?
- Complete checks, but not to be fully proven – have the engines achieved sufficient design coverage to detect a bug?

Use JG-COV in conjunction with FPV (or other apps) to answer these questions!



This page does not contain notes.

Formal Coverage Types

Stimuli Coverage

- Measures the ability of the formal testbench to **reach** all code and functional covers.
- Formal testbench is exercising the entire design code and design behaviors that both are expected to happen and are likely to be in the path of a possible CEX.

Checker Coverage

- Measures how much of the design is observed by assertions.
- Gives confidence that formal **checking** is sufficient to detect a bug.

Formal Coverage

- Consolidation of Stimuli and Checker Coverage results to a single coverage number.
- Cover item is “covered” if it is **reachable AND checked** by the formal environment.
- Provides a single-metric view of formal verification coverage.

Signoff should be a function of the verification plan.

- Coverage metrics are useful indicators of not done.

557 © Cadence Design Systems, Inc. All rights reserved.



Formal coverage as well; this is a complicated business. Of course, we've discussed this before that when you have a kind of influence for some assertion, if the assertion is proven, it doesn't necessarily mean you tested everything inside of that COI. Therefore, if that's the case, how can we be sure that our formal checking is complete while we use the JG-coverage app to do this? This answers the question in every part of my design exercise meaningful by some assertion or cover. And if the answer is no, then this tool tells us, and we can decide what we do about that. There are all these listed different kinds of coverage types here, stimuli coverage, much as the ability of the testbench to reach all code and functional covers.

Checker coverage measures how much of the design is observed by assertions and covers.

Checker Coverage Modes

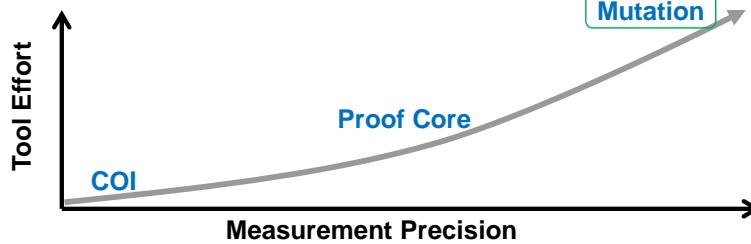
Three selectable modes for Checker Coverage Measurement:

1. COI (Cone Of Influence) – structural fan-in analysis
2. Proof Core – engine-based region abstraction
- 3. Mutation – engine-based single point mutation**

Increasing Checker Coverage measurement precision

- Requiring higher tool effort

Answers the question:
If I mutate (corrupt) my design, then can my verification environment detect this change?



558 © Cadence Design Systems, Inc. All rights reserved.



For the coverage app, there are different levels; if you like, it's a tradeoff between how long it takes and accuracy.

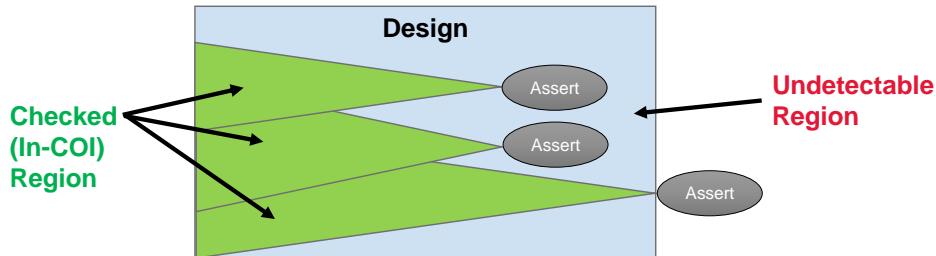
The simplest kind of coverage is COI coverage. For example, you have any parts of your design which do not appear in the COI of any assertion or cover, you know, that it never got exercised. You know, there's no argument about that. So that's quite clearly that is a problem. Now, that is not to say you are complete. If the answer is yes, everything is in the COI assertion. It doesn't mean you are complete because you don't know whether you use the entire COI for proof.

This is where the proof core comes in. That recognizes what was actually needed to prove the properties. And we can increase the effort more and do this thing called mutation. Mutation will modify the design in ways that should cause an assertion to fail. The idea here is that if parts of your design can be changed and none of your assertions fail now, then it means clearly that there is no assertion detecting that change in behavior. So now this is very time-consuming.

This takes a lot of effort and more effort than the actual proof itself, but it gives you more accuracy or more confidence. So, it's always a trade off on how long you have available in your project versus how important these things are to verify.

Example: Cone-Of-Influence (COI) Mode

- Design fan-in is computed starting from assertion, traversing back to inputs.
- The union of COIs from all assertions is reported.
- Anything outside the COI region *cannot* influence assertion status.
- Anything inside the COI region *may* influence assertion status.
- Fast measurement – no formal engines are run.



559 © Cadence Design Systems, Inc. All rights reserved.



Here's an example of a COI. This is the simplest thing. All the bits in red here are regions where we're not verifying anything for sure.

Submodule Summary

In this submodule, you learned:

- Formal is a very different paradigm from simulation.
- One needs to be prepared for a complete change in mindset.
- Formal does not have a simple “one size fits all” use model.
- New use models are always being created or modified as the technology progresses.
- Apps exist to fix common design and verification problems.
- Many apps are designed with the intent to minimize the amount of SVA and Formal expertise required.
- Each separate app is a specialism in its own right.
 - No one knows all of them in detail.



Obviously, you probably still have a lot of questions from that, but at least you're aware of all the different kinds of things which can be done with formal now.

Remember that formal is a very different paradigm to simulation, and for some of these problems like SPV, equivalence checking, and connectivity checking, you simply can't do them in simulation in any real kind of timeframe.

You need to be prepared for a complete change in mindset.

There's no one size fits all use model. Different customers use different apps, and they're always using different use models anyway.

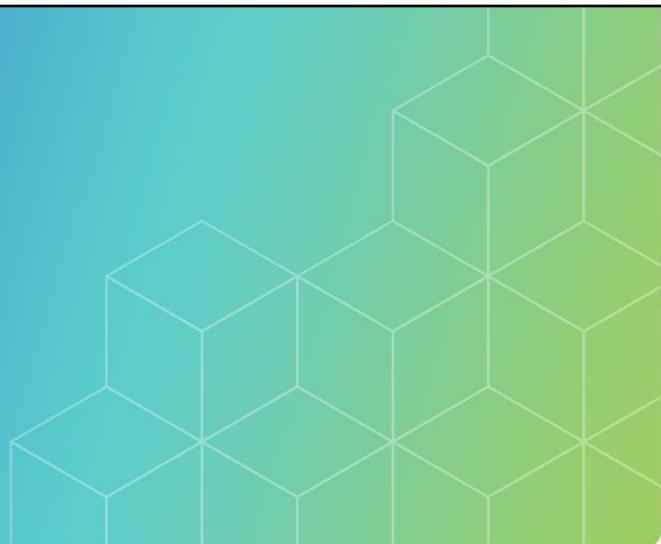
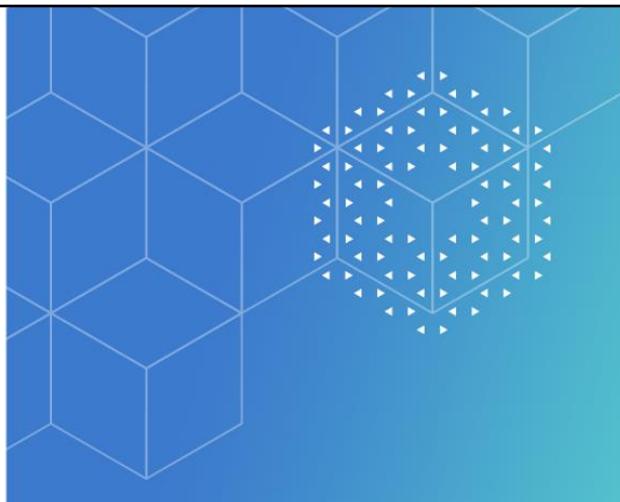
As time goes on, people find different ways of using these things. Apps are there for the simple reason that they fix common problems people have, and many of these apps are designed with the express intent to minimize the amount of SVA and formal knowledge you need in order to use them, which is why Jasper was so successful.

Each app can get very detailed in its own right, so you can't expect to know how every single app works in great detail.

And in practice, you'll find that you don't need in any given company doing any similar kind of projects. You won't use all the apps necessarily,

There's no right or wrong answer on which apps you should or shouldn't be using; it all depends on what you're doing and how you wish to approach your verification.

This concludes the section on form analysis and models. Now proceed to the next section.



Submodule 5-5

The Verification Completeness Problem

cadence®

It's important to understand what the term verification completeness means and why it's such a big problem. And in many ways, it's something that can never be achieved. It's important for any verification course that we cover these kinds of concepts.

The example we're going to follow through is about formal verification, a very simple example just to show you how hard this is to achieve completeness.

Submodule Objectives

In this submodule, you will:

- Review verification methodologies.
- Examine a case study that highlights the verification completeness problem.
- State techniques for minimizing the risk of verification holes.
- State the merits of each verification technique.



We will start off with a review of verification methodologies, and the case study will be specific to formal. I minimize the risk of missing a bug during verification.

Scale of the Functional Verification Problem

The most complex processor IC designs may have, for example, 500 thousand flip-flops (FFs).

Those FFs could be in any of 2^{500000} different states at any particular time.

The actual number of states will be much less because:

- By design, not all states will be possible.
- There will be many states we don't particularly care about.

However, the number of states is still absolutely massive.

It is essential that you accept this truth.

You can't possibly test everything!

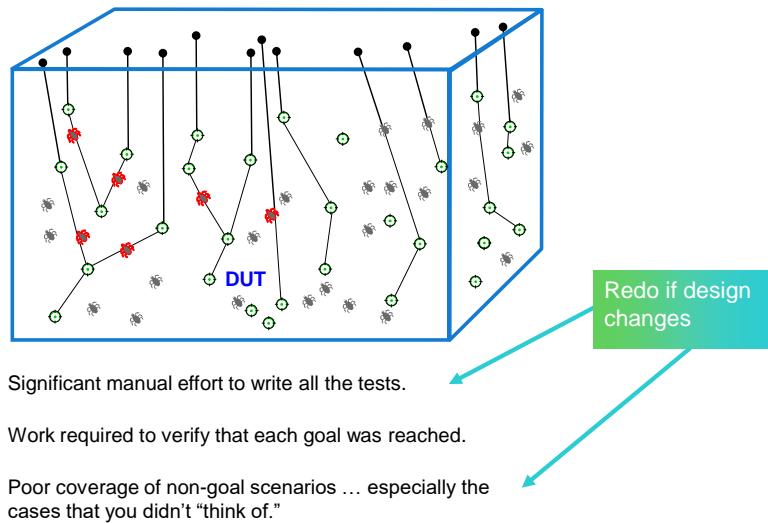


The scale of the problem is really at the root cause of why it's so hard to be sure you've covered everything. For the most complex ICs of today, you might have something like, for example, 500,000 flip-flops, which is a lot. And if you have 500,000 flip-flops, it means you have a possibility of two to the power 500,000 different states at any particular time. Now, of course, that theoretical maximum will never be achieved because, by design, most states will not be possible, or many of those states rather. And there are also many states we don't actually care about because they don't represent real-life operations. Even so, we've still got a huge state space to explore. The first thing to accept is that you can't possibly test everything, and if you're a simulation user, whether you like it or not, that's the decision you already made. If you're using simulation you've recognized, you can't possibly test everything. The problem then becomes one of minimizing the risk of bugs and escaping verification.

Most Basic Approach to Verification: Directed Tests

Verification Engineer:

- Defines DUT states to be tested based on specified behavior and corner cases.
- Writes directed test to visit and verify each item in the test plan.

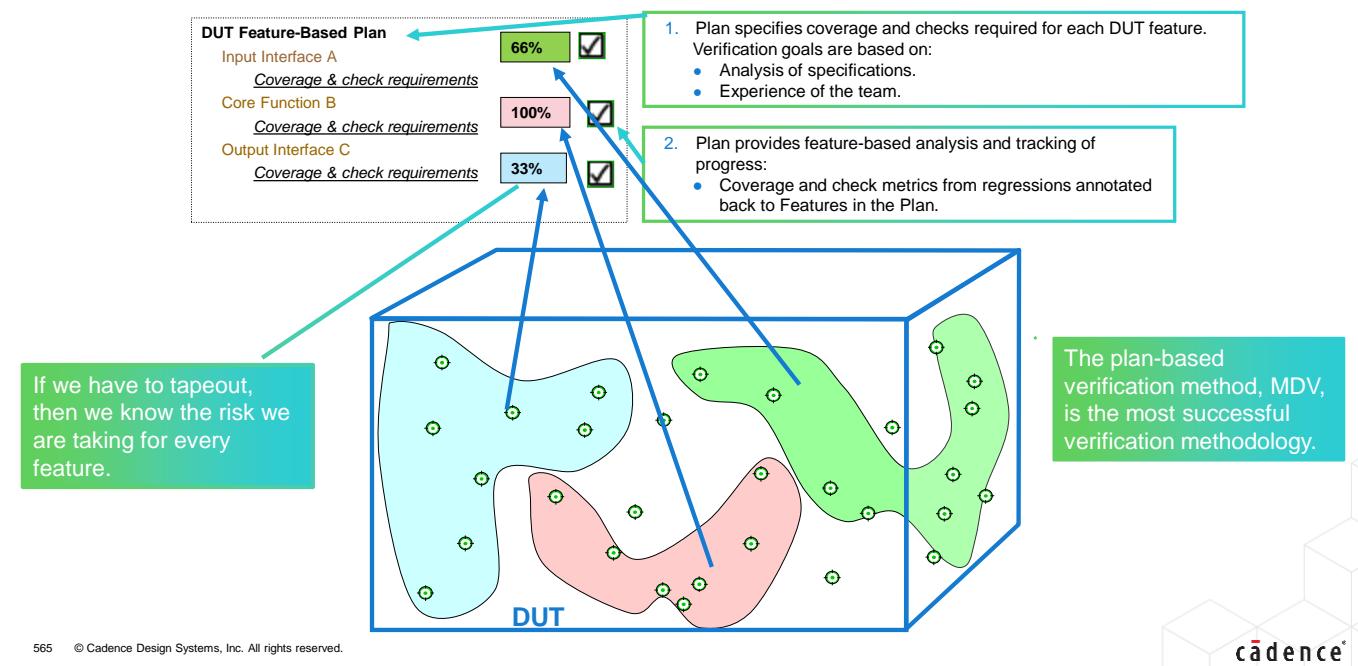


564 © Cadence Design Systems, Inc. All rights reserved.



A history lesson on verification. It started off with specific director tests, so you had to sit there and think of what stimulus to apply and apply that to the design and observe the outputs and check it's behaving correctly. This was a very labor-intensive task because you have to think of and write all tests, and if anything changes, like the design changes, you have to do it all again. You had very poor coverage of non-goal scenarios. It's really the things you don't think of that catch you out, not the things you did think of.

Metric Driven Verification (MDV)



565 © Cadence Design Systems, Inc. All rights reserved.

cadence®

This kind of evolved into a verification methodology driven overall by coverage, and that coverage is defined in some hierarchical verification plan.

So, this is what this is showing here, a DUT feature-based plan. For all the features that are required of the design, we come up with a list of all the different kinds of checks on the different kinds of coverage we need in order to be sure that we verify the design functionality is as expected. These plans are derived based upon the experience of the team, and analysis of the functional spec and what this ends up being is that if you can imagine your whole design now,

this is the entire state space represented here for any given feature; this will cover a certain part of the design. Another feature will cover another part of the design, and another feature will cover another part of the design in terms of both checks and coverage. And we're hoping that adding all these up means we've tested the entire design. Essentially, that's what we're doing. And the advantage of doing it in a hierarchy-based plan like this is that you've got an up-to-date measure, i.e., as you're doing each test, you can update this to see how much of each feature has been tested and so forth. The whole idea of any kind of verification, whether formal or simulation, is that you want to achieve the maximum amount of coverage you can 100%, ideally in the context of no checks failing.

We can view these metrics here as being an indication of how much risk we take. If we tapeout today, verification is about risk management because we're already accepted. We can't test everything.

Concise and Simplistic Summary of What MDV Is

Define a hierarchical verification plan, which:

- States the features the design is supposed to have.
- Specify the checks to be done in order to test those features.
- Specify the coverage required in order to be confident that we tested every important scenario the design is expected to work under.

We know we have finished verification when we have reached 100% coverage in the context of no checks failing.



In order to state what MDV is for the simulation world, this is the best methodology anybody has come up with so far. We devise a hierarchical verification plan which states the features the design is supposed to have, which we get from the functional spec. We specify the checks which need to be done in order to test those features and the coverage required in order to be confident. We tested every important scenario that the design is expected to work under, and our hope is that we reach 100% coverage in the context of no checks failing. Now we might have to settle for a number lower than 100 due to project deadlines like tapeout.

“The Best Laid Plans of Mice and Men”

As the saying goes, “If we fail to plan, then we plan to fail.”

However, having a plan does not guarantee success.

The plan could be wrong.

The plan might not be implemented correctly.

To show how this can happen for even the simplest of designs, we will follow a case study.

The schematic we will show defines the functional specification.

The intent is to verify every feature of the design by defining SVA properties.

We will verify these properties using JasperGold Formal Verification.



However, just because we've got a plan doesn't mean everything is going to be okay because the plan might be wrong. The plan is susceptible to the same human errors that everything else is. The plan might be correct but not implemented correctly, so everything could still go wrong. What we're doing, in this case study is showing you an example of such a case. What we want to do is show a schematic that defines the functionality required. So effectively, the schematic represents what the functional spec is, and the intent is that we verify the features that the design has, and in this case, we will use JasperGold because this is a formal course.

Completeness of Property Sets

We obviously define what we think are sufficient properties:

- As identified in the vPlan.

This is not just limited to properties or formal verification.

- The same argument for a constrained random coverage-driven simulation testbench:
 - Did we specify enough coverage?

How can we be certain they cover all required functionality?

A formal tool might tell us that all I/Os and FFs are covered by assertions.

This does not mean that we have written properties to cover all functionality.

- In the same sense, that code coverage does not tell us we have complete functional coverage.



If we're using JasperGold, we're going to write some properties in order to check our behaviors. Correct. And those checks are being derived from the verification plan.

The reasoning we will apply here doesn't necessarily only apply to formal. You could make similar arguments for simulation, like if I've got a constrained random stimulus testbench, which is coverage driven, our goal is to fill coverage. How do we know that the coverage is correct? You can point to anything and say that might be wrong.

So how can we be certain they cover all the required functionality? Now you do have some help from the tool. It's not like the tool tells you anything. A formal tool, we could get reported, for example, that every IO and every flip-flop is covered in the funding cone of some assertion and was used effectively using something called proof core in JasperGold.

This does not mean with enough properties, however, to cover all functionality, even if the metrics we get from this tool tell us, yes, everything has been covered just in the same sense that if we got code coverage, it doesn't tell us we've got complete functional verification.

Verification [in]Completeness Example

Shown is a schematic of the functional requirements that are implemented in RTL:

- We write blackbox properties, which we think completely check all desired functionality.
- Human error or oversight might mean that we don't specify all desired behavior.

Tool might tell us:

- All I/Os and all FFs are included in some assertion:
 - There is no deadcode
 - All checks pass
 - All cover items pass

At first glance these properties seem sensible, however

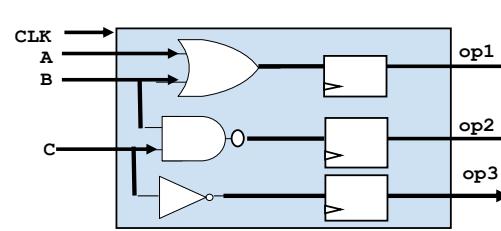
```
property P1 is always ( (A || B) |=> op1 ); 
property P2 is always ( (!A && !B) |=> op2 );
property P3 is always ( !C |=> op3 );
```

Should we conclude that verification is complete?

- No way of knowing that we specified enough properties to check all functionality.
- Human error or oversight might escape detection in the verification process.

Human error. A has no effect upon op2 but property still holds.

a	b	c	op1`	op2`	op3`
0	0	0	0	1	1
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	1	0	0



569 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Here's an example of the schematic we've got here. Those are the requirements, and this is a property set we hope will check all behaviors of the design at the first inspection. These look like a perfectly reasonable set of properties, right?

For example, in this property, I say if I've got A or B on the next cycle op1 should be true. That would appear to check this all-gate and this flip-flop here. And likewise for the other outputs.

However, there might be human error or oversight, which means we didn't capture that behavior correctly. The intended check is not specified correctly so in this example. For P-2, it says Not A or not B implies OP2 is true in the next cycle. Now notice op2, A does not appear in the funding code for that. A has no influence over what OP2 is doing; however, It just so happens, and we can check this by looking at the truth table if we wish; that property will always hold, i.e., always be true.

So even though the tool might tell us that all I owe is a flip flop included in some assertion and that there are no branches of code that can't be reached, and all checks pass. All cover items pass, i.e., 100% coverage in the context of no checks failing, which I said was the end of verification. We still haven't verified our design correctly due to human error, which the tool brought to us. The question is that what do we do about that then?

Design Mutation

Shown is a schematic whose behavior satisfies our properties.

This is a mutant of the desired design.

We can see that our verification environment fails to detect this mutation of our desired design.

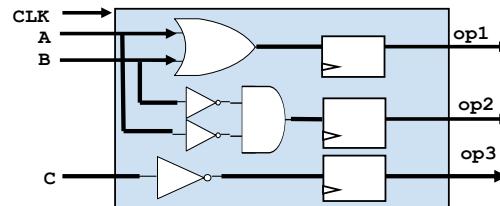
Hence, we did not capture a property set that completely and uniquely defines desired behavior.

- However, maybe we never intended to allow flexibility for the design implementation.

**Change from
Desired Truth
Table**

a	b	c	op1'	op2'	op3'
0	0	0	0	1	1
0	0	1	0	1	0
0	1	0	1	0	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	0	0
1	1	0	1	0	1
1	1	1	1	0	0

```
property P1 is always ( (A || B) |=> op1 ); ✓
property P2 is always ( (!A && !B) |=> op2 );
property P3 is always ( !C |=> op3 );
```



570 © Cadence Design Systems, Inc. All rights reserved.

cadence®

In this example, we can (if the properties are sufficient) detect the mutant design.

In practice, the mutation might not propagate an observable difference from the expected behavior to somewhere it could be detected.

This indicates a problem with complete stimulus; that is, we did not provide the correct stimulus to observe the difference in behavior caused by the mutation.

Now, if we imagine that we make a change to the design, we might expect that these properties, if they completely define the required behavior, would fail. So that's called design mutation.

What we've got here now, we've changed that schematic. Now, if I just go back to the previous slide for a second. Pay attention to the schematic diagram. It's changed from that to this.

Now it just so happens that the change we've made will not be detected by any of these assertions, i.e., if all the previous assertions passed there are going to be no failures here. I've managed to change the design without my properties detecting the design has been changed.

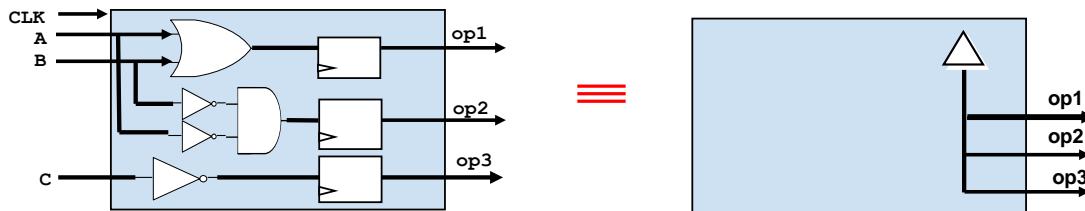
This comes down to what is really a philosophical question do we really want a property set that you completely and uniquely define the desired behavior? It might be the case that we don't care if the design can be changed and none of the properties fail because we've captured all of the required behavior that's important to us inside of these properties. But again, that requires the tool understanding what our intent is, which obviously it can never do, a tool is never going to be able to read our minds.

So again, we could follow that through in a truth table. You notice these bits which are kind of grayed out, these zeros here, there, there, and there. Those are changes in the truth table from the previous one. And it just so happens that all of these properties still hold even with that design mutation.

We Don't Even Have the Fundamentals Correct

All these properties pass with the outputs tied to the power rail.

```
property P1 is always ( (A || B) |=> op1 ); 
property P2 is always ( (!A && !B) |=> op2 ); 
property P3 is always ( !C |=> op3 ); 
```



The correct assertion to check OP1 is not particularly intuitive.

```
property P1 is always ( op1 == $past(A || B)); 
```

571 © Cadence Design Systems, Inc. All rights reserved.



And when you think about it, it's even more; it's even deeper than that. It's more fundamental than that because all of these properties will pass if all outputs are tied to the power rail instead of this required circuit. Somebody could present us with this so all the properties would still pass. They can never fail, can they? Because these outputs, one, two, and three, will always be true.

In order to correct this property P1, for example, what we would have to do is this if we wanted to do it in single assertion and this solution is not exactly intuitive, most people would not be able to come up with this by themselves without having seen a similar example. What we're saying here is OP1 must always equal the past value of this expression or be past one cycle ago, and being as that's a flip-flop, this satisfies that requirement now. But again, it probably wouldn't be our first guess. It's very unlikely we'd come up with this unless we'd seen a similar property before.

Locality and the Dangers of Over Optimism

If one made JG report elements in the COI of the assertion, then it would be both the grey and green parts.

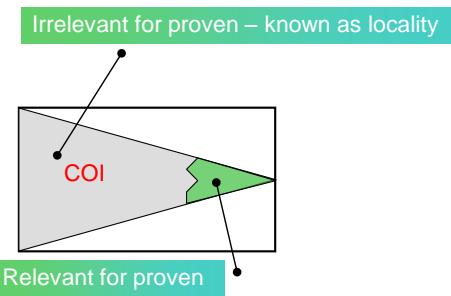
- For example, the static fanin cone of all signals in the assertion.

Question: Does this mean we have covered/verified/exercised all elements in the COI of the property?

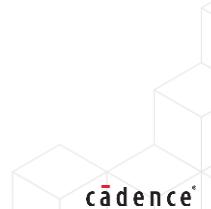
There is the danger of over-optimism on coverage.

The grey part is irrelevant for the proof.

We learned nothing about the grey part.



572 © Cadence Design Systems, Inc. All rights reserved.



Knowing how much of the design was meaningfully exercised by JasperGold is a very important metric for just about every customer.

The coverage app can collect design metrics on:

- Stimuli coverage
- COI coverage
- Proof coverage
- Bound coverage
- Code Coverage
 - Branch
 - Statement
- Functional Coverage
 - SVA/PSL covers

Contact your local AE to discuss the JasperGold Coverage app.

Now, with the report on, you know, for formal, we've got a code of influence here. And it might be that the entire code of influence, which is in effect, the structural funding code for the signals and the assertion, it's not required in order to prove the assertion. It's irrelevant as regards the proof. Obviously, these signals can influence what these signals in the assert do, but they can't change the result of the assertion, which is a completely different thing. It could be the case that this gray portion of this code is not actually required for the proof. It doesn't matter what it is in effect. We've not tested it.

Okay. So how do we know that? We're not being overoptimistic?

If we just looked at the structural function code and says, yes, we tested it, the answer is, well, no, it might not be needed. What we do in Jasper is use the just for gold coverage up in order to tell us this, which is what we said on the previous slide just for gold coverage up.

Can We Achieve Completeness?

Is it impossible for a tool to read our minds?

Probably, but we can get closer using tool features.

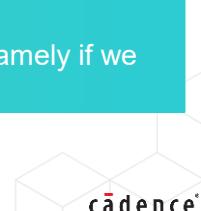
No current tools can tell you when you have a complete property set.

There is no substitute for:

- Understanding the requirements
- Intelligent planning
- Attention to detail
- Hard work

No EDA tool can tell you that your verification is complete.

All the EDA tools can tell us is if we didn't test all the things that we said we would, namely if we are incomplete.



The question being asked is how can we achieve completeness, i.e., can the tool tell us that we've got complete verification? Well, the answer is no, because the tool can't read our minds. And effectively, what you're describing is the same problem as just the tool creating the design for us. If there was a completely unambiguous way of specifying the requirements, and that's not going to happen in the near future, of course, but we can get closer to that minimize risk using the different tool features we have.

So regardless of what anybody claims, no tools can tell you. You've got a complete property set.

There's no substitute for doing all the things only humans can do, which are listed right there. However, what anybody can tell us is we didn't test all of the things that we said we would, which at least gets us part of the way toward understanding whether we're complete or not.

Assessing Completeness

Defining your verification objectives upfront is the best way to assess when you are done.

Beyond this, there are other ways to assess incompleteness:

- Do all the outputs of the block have assertions describing their behavior?
- The JasperGold Coverage app can help assess what portion of the design your assertions are exercising:
 - Cannot tell you when you are done, but when you are *not* done.

There are no tools that can tell you when you have done enough verification at present.



So, in assessing completeness, you really have to measure this against what you've defined as being a complete set of tests. Now, what the tool can tell you is whether you're incomplete in your verification, like do all the outputs of a block have assertions describing their behavior? JasperGold coverage can tell us what parts of the design were and were not exercised during all proofs and all covers. So essentially, what they are telling you is that you've not finished. Not that you are. Incompleteness is an easy thing, knowing you are incomplete; completeness seems to be an intractable problem with the present technologies.

Verification Execution Completeness

This is something that we can know.

The goal is the same in both dynamic (simulation) and formal.

All check items pass/proved in context of coverage items *equals* verification complete.

Execution Method	Execution Item	Completeness	Execution Method	Execution Item	Completeness
Dynamic	Check	All checks are executed with no violation (checks pass)	Formal	Check	All checks are proven with no violation (checks proved)
	Coverage	All coverage items are reached (pass); driver + DUV dependent		Coverage	All coverage items are reachable; constraints + DUV dependent

575 © Cadence Design Systems, Inc. All rights reserved.



We can know for sure that all checks we have defined have passed and we have reached all cover items.

The holy grail of verification is knowing whether those checks and cover items defined all behaviors we want to verify.

This might be possible in the future but not at the present time.

But what you do know is how much you've executed.

So regardless of whether you're doing dynamic verification, which is simulation and emulation, your requirements in that are typically you define some checks and you just find some coverage, which is all the expected scenarios, the designs expected to operate under. And you wish to get 100% coverage in the context of all checks passing in formal. That's exactly the same.

Although in formal coverage means a slightly different thing, it uses exactly the same syntax as simulation. A coverage in formal tells you whether that scenario is ever possible. It's the same situation in formal. Your checks without your coverage are useless. You need 100% coverage in the context of no checks failing.

What Can Be Automated?

It's probably more important to ask the question in reverse, what can't be automated?

- Writing the functional specification
- Writing the verification plan
- Interpreting the verification plan to implement the checks and coverage
- Interpreting the results from verification runs
- Understanding the limitations of the verification technique we use

EDA is about automating all that can be, for example:

- Managing huge amounts of data
- Compilation/recompilation
- Running and rerunning verification tools after bug fixes
- Analyzing which tests are the most valuable in terms of coverage achieved
- Tracking bug rates/coverage accumulation rates



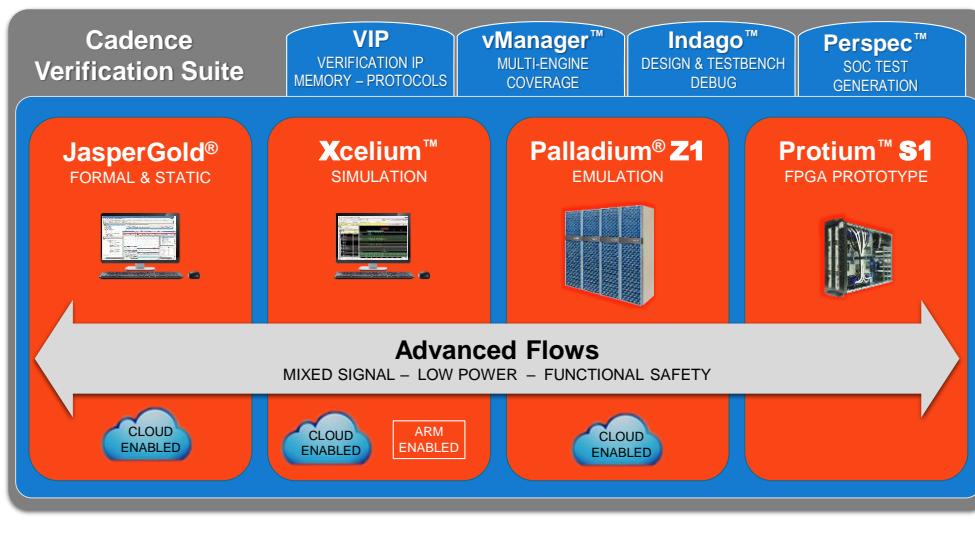
What can be automated, then? Despite all the claims you see in the media for machine learning and AI, the world isn't as advanced as it tells you on the news when you ask what can be automated. And a more important question, and a more realistic one, is what can't be automated? What can't be automated is writing the functional spec in the first place and the verification plan. Interpretation of that verification plan into the individual checks and coverage. It's often the context under which the results were received during the verification runs. And understanding the limitations of any technology we're using.

So clearly, that's why different technologies exist, like formal and simulation and hardware acceleration, emulation, FPGA prototyping, and so on. There are all these different ways of doing things because not one methodology or technique does everything. So basically, there is no single solution.

Eda is about automating all that can be. We're talking about masses of data because you're talking about huge designs which create huge amounts of data. We might have to edit things and compile and recompile, which might take hours running verification runs, and if they fail, rerunning them after bugfixes, analyzing which tests give us the most information in the shortest time, tracking our bug rates, and our coverage accumulation. Those are the kinds of things that EDA does.

There's no magic tool that you can just press one button and say, Go and make my chip for me.

Cadence Verification Solution

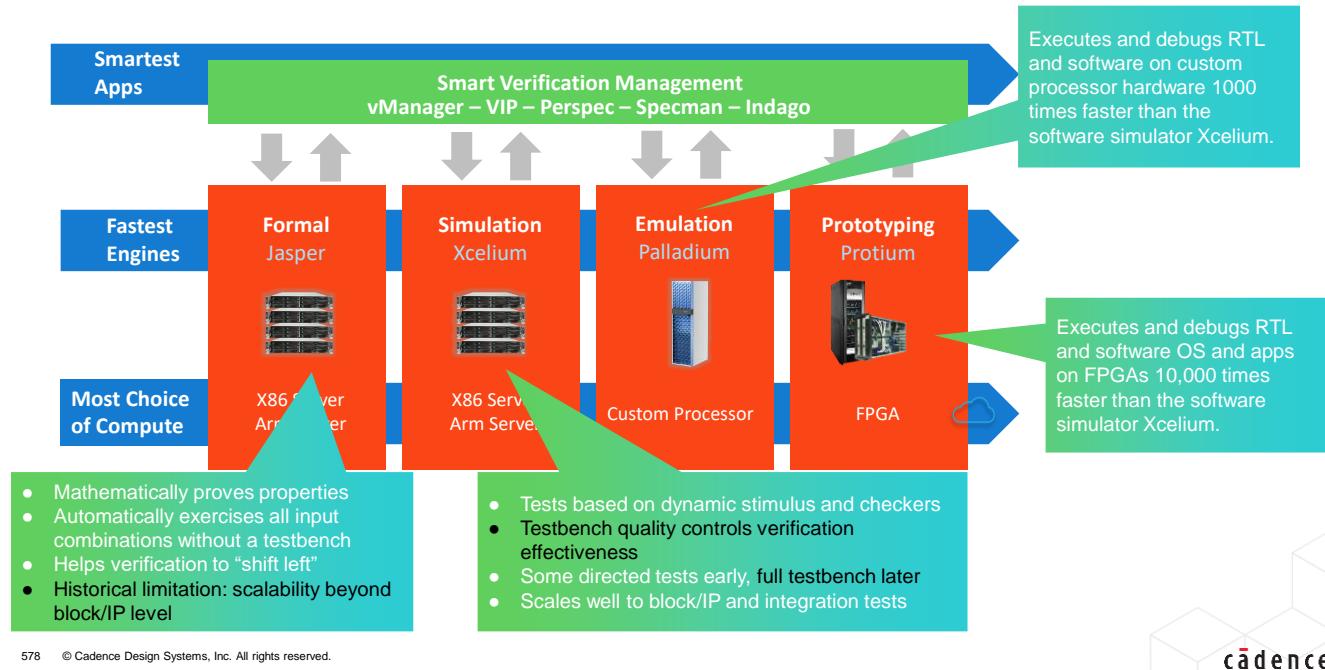


577 © Cadence Design Systems, Inc. All rights reserved.



This is the reason Cadence has a verification suite. Some techniques are appropriate at different levels of integration as well, so formal and static verification and mixed signal, low power, functional safety, all of these things all have to be done.

Where Each Technique Apply



578 © Cadence Design Systems, Inc. All rights reserved.



And this is how they fit into where each technique applies.

So formal are mathematical and algebraic proofs. And it exercises all import combinations without the need for a testbench, and it helps us actually start verification while we're still designing. And a historical limitation is scalability beyond the block and IP level. If you're looking for exhaustive proof, simulation is the oldest form of verification. They started with the director tests, but really most people are using the metric-driven verification environment, which uses random stimulus on the inputs, and it scales well into block and IP, and integration tests. The problem becomes speed.

Emulation is a way of running a lot quicker, so thousands of times faster than a software simulator like Xcelium would.

Prototyping is when you're trying to debug the OS and software, which is tens of thousands of times faster than software and simulations. If you want to see your operating system boot, for example, you will be waiting an awfully long time if you do a simulation of this. In contrast, if your RTL is relatively stable, you can put this into an FPGA array effectively, which is like prototyping your entire chip, and you can run software in real-time on top of it.

Customer Needs: Emulation and Prototyping

Emulation “Debug your design”

Key Care Abouts

- **Predictable fast build:** Rebuild new RTL bug fixes every night.
- **SOC level capacity:** IP level simulations typically run fast enough already.
- **Fast and complete debug:** Full trace, runtime debug controls.

Prototyping “Debug your software”

- **Build time and debug less important:** Design stable, rebuild only every 1-2 weeks.
- **Highest performance:** Software debug requires even longer simulation runs.
- **Lowest cost:** Replicate one build for many different software developers.
- **Some SOC level capacity:** More and more software complexity at the SOC level.

579 © Cadence Design Systems, Inc. All rights reserved.



This page does not contain notes.

There Is No “One Size Fits All”

Verification and Software Platforms Need to Interoperate



Virtual Platform	Formal Analysis	HDL Simulation	Acceleration Emulation	FPGA Prototype	Prototyping Board
<ul style="list-style-type: none"> Almost at speed Less accurate (or slower) Before RTL Great to debug (but less detail) Easy replication 	<ul style="list-style-type: none"> Non-scalable Exhaustive Early RTL Great for IP No SW execution 	<ul style="list-style-type: none"> KHz range Accurate Excellent HW debug Broadly available Mixed-abstractions Limited SW execution 	<ul style="list-style-type: none"> MHz Range RTL accurate After RTL is available Good to debug with full detail Expensive to replicate 	<ul style="list-style-type: none"> 10s of MHz RTL accurate After stable RTL is available OK to debug More expensive than software to replicate 	<ul style="list-style-type: none"> Real time speed Fully accurate Post Silicon Difficult to debug Sometimes hard to replicate

580 © Cadence Design Systems, Inc. All rights reserved.



There is no magic silver bullet, no one size fits all. You need a combination of all these things in order to verify today's complex designs.

Here's showing you a kind of going from left to right probably in terms of time in the project. So here we're doing kind of simulation software simulations using less accurate models.

These models here, virtual platforms, and so on, we can exercise that. Our software works correctly in our OS, but we don't have the information on the implementation of the chip is lower, low enough level of detail that the tools can actually produce a kind of blueprint for the chip.

This is before writing RTL code. The RTL code is what we have to write to give us enough detail to actually manufacture a chip in real life. These two things on the left are all at a very high-level software level.

We then start designing the blocks from the bottom up. Using formal simulation and then HDL simulation, you're talking about as if you've got a clock frequency of kilohertz. It takes an awfully long time to simulate even a second's worth of real-time.

At that point here, the acceleration emulation helps because we can get up to say one megahertz kind of range as if we got a one-megahertz clock. So typically, you know, the final chip is probably going to have a clock of around two gigahertz or something similar to that. So, we're still orders of magnitude off the real-life speed here. This is cycle accurate. We can debug this at the level of detail we need.

And FPGA prototyping gets to the tens or hundreds of megahertz when we can actually run our design, which we think is pretty stable at this point in order to verify our software. Websites and apps that go on top of that, of course.

Submodule Summary

In this submodule, you learned:

- Functional verification is about risk management, not verifying everything.
- We need to minimize the chances of a critical bug escaping the verification process.
- There are many tools and methodologies to help us do this.
- We cannot abdicate the responsibility for thorough verification to automation.
- Automation only helps with managing data, predictable, repetitive tasks, and computing.
- Functional verification is still a human process vulnerable to human error.
- There is no single verification methodology which works best for every circumstance all of the time.



In summary, it's important to understand this from the very beginning. Functional verification is not about completeness. It's about risk management. You can't possibly test everything.

So, we just concentrate on what features are important to us and minimize the chances of bugs escaping.

And we can do that by using lots of different tools and methodologies.

Automation doesn't abdicate our responsibility to make sure everything is correct.

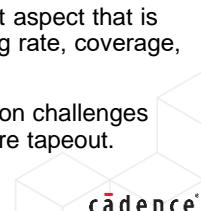
There is no solution to replacing humans, so automation only helps to manage the huge amounts of data and analyze the huge amounts of data that these verification tools will create.

There is no single verification tool you buy that is best for every circumstance. Each has its own kind of advantages and disadvantages. That completes this section. There is no lab exercise associated with this, so please move on to the next section.

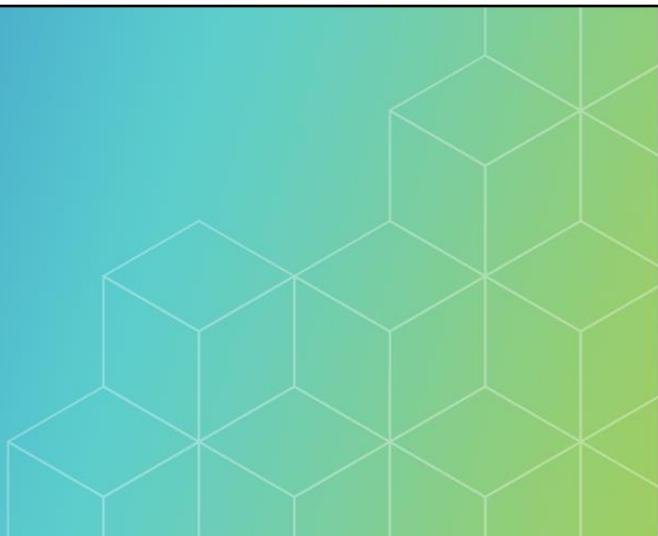
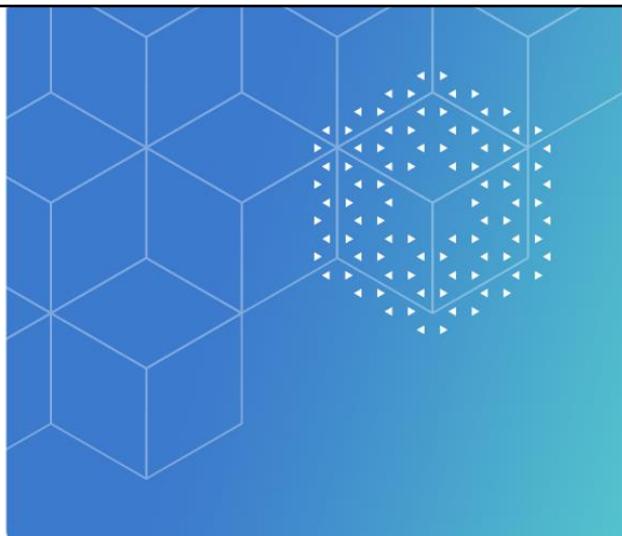
Module Summary

In this module, you:

- Defined functional verification.
- Wrote a simple SV testbench using the FSM example of a drink machine.
- Debugged using the waveform viewer, breakpoints, and Xcelium simulator tool.
- Recognized the 3 corner stones of modern verification flow, randomization, coverage, and assertions and why we need them.
- Defined constraint-based randomization.
- Discussed coverage and other metrics (code coverage, functional coverage, SV cover groups).
- Defined assertion-based verification (assertions and covers).
- Identified class-based testbenches, UVM and methodology.
- Discussed the verification planning, the concept of an executable and dynamic verification plan, and management and mapping with metrics and coverage to the vPlan.
- Recognized the ever-popular MDV Methodology and its different phases.
- Reviewed formal verification and its use models.
 - Exhaustive proof, Automated Formal Checks, Design Exploration, Equivalence Checking and many others.
- Identified Accelerated Verification (Emulation, Prototyping and multicore simulation).
- Identified the Verification Completeness problem. How do we know we are done? completeness problem.
- Identified the verification management aspect that is “Do we know why it isn’t tested?” (bug rate, coverage, risk management)
- Identified and appreciate the verification challenges and the process decisions made before tapeout.



This page does not contain notes.



Module 6

Digital IC Design Methodology

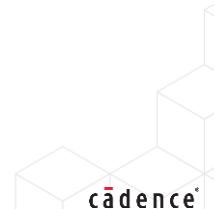
cadence®

In this module, we will go over digital IC methodology.

Module Objectives

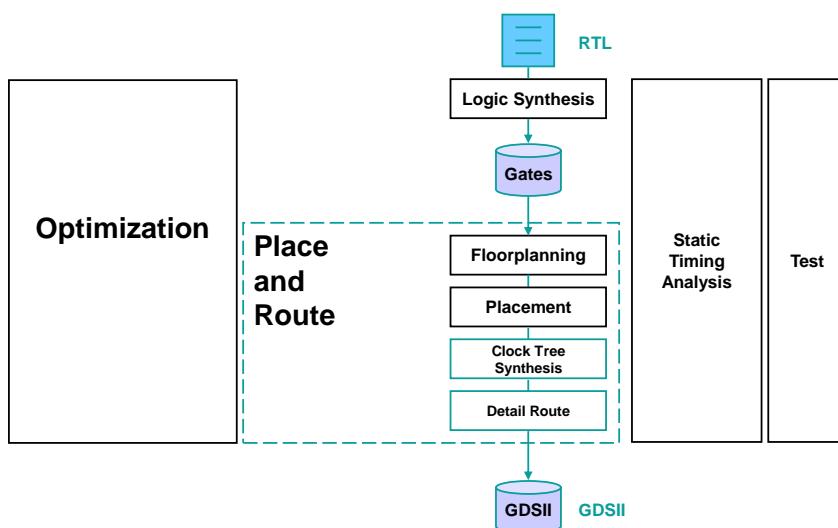
In this module, you will:

- Demonstrate the implementation flow at a high level.
- Describe the purpose of logic synthesis and the function of Static Timing Analysis in the logic synthesis flow.
- Identify Design for Test (DFT), built-in self-test, and Joint Test Action Group (JTAG) design techniques.
- Describe the following steps in the physical implementation flow:
 - Floorplanning
 - Placement
 - Physical synthesis
 - Static timing analysis
 - Clock tree synthesis
 - Pre- and post-CTS optimization
 - Routing
 - Extraction
 - Delay calculation
 - Timing analysis
 - IR drop analysis
 - Design verification
 - Mask prep



In this module, we will describe the high-level implementation flow. We will go over the purpose of logic synthesis, bist, and J-TAG technologies. Next, we will describe the physical implementation flow, starting with floorplanning and continuing with placement, physical synthesis, clock tree synthesis, pre- and post-CTS optimization, routing, extraction, delay calculation, timing analysis, IR drop analysis, design verification, and, finally,, mask prep.

Implementation Flow Overview



585 © Cadence Design Systems, Inc. All rights reserved.

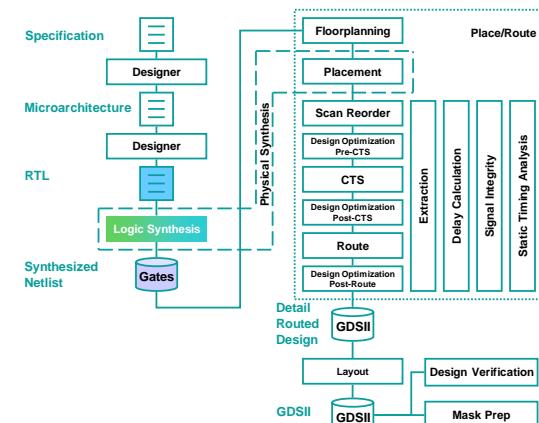


This diagram shows an overview of implementation starting with logic synthesis, which is the process of generating a gate-level netlist from an RTL and a timing constraints file. Synthesis is followed by place-and-route, which includes floorplanning, placement, clock tree synthesis, and detail routing. If timing is not met during placement, or clock tree synthesis, or routing stages, optimizations will be run for setup and hold to meet timing. The static timing analysis tool will analyze the timing at the various stages. Test structures will be added during synthesis to detect manufacturing defects, if any, after fabrication.

What Is Logic Synthesis?

The process of parsing, translating, optimizing, and mapping RTL code into a specified standard cell library.

Example: To determine the feasibility of the design, we need to synthesize the RTL code into gates and measure timing, power, and area.



586 © Cadence Design Systems, Inc. All rights reserved.



Let's understand the logic synthesis stage with the help of this flowchart. Design architects create the spec for the design in terms of functionality, frequency, power, area, and timing. Hardware designers use the spec and the microarchitecture to create an RTL. After creating the RTL, logic synthesis is used to parse, translate, optimize, and map the RTL code into a specified standard cell library. During logic synthesis, the RTL is optimized with the help of technology libraries, and the output is the synthesized netlist that the place-and-route tool uses to generate an implemented physical design.

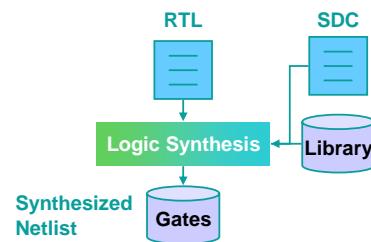
Logic design is the representation of a functional design of a circuit in the form of logic operations, arithmetic operations, control flows, etc. The logic operations include booleans, and, or, X-or and nand.

After place and route the GDS2 representation of the physical layout is sent to design verification and mask prep steps.

So, during the flow, when you do the logic synthesis, you determine the feasibility of the design.

Logic Synthesis: Input and Output, Format

- Input
 - RTL in the Verilog language
 - Constraints in Standard Design Constraints (SDC) format
 - Timing Libraries in Liberty (.lib) format
- Output
 - Gate-level netlist



587 © Cadence Design Systems, Inc. All rights reserved.



Let's check out different inputs and outputs for logic synthesis.

When you're running a logic synthesis, the required inputs are RTL in the form of Verilog, or VHDL, directives, pragmas, System Verilog format, constraints in the form of SDC, and library in the .lib format.

The main output generated from the logic synthesis process is the optimized netlist which is a gate-level netlist.

Logic Synthesis Goals

- Minimize area
 - In terms of cell count and cell size.
- Minimize power
 - In terms of switching activity in individual gates, deactivated circuit blocks.
 - In terms of leakage power.
- Maximize performance
 - In terms of maximum clock frequency of synchronous systems, throughput for asynchronous systems.
- Quickly produce accurate functional models
 - Gate-level model is functionally equivalent to the RTL model.
 - Gate-level model is produced in less time than is required by an experienced logic designer to create the same model.
- Produce predictable and accurate results
 - Timing, area, and power consumption calculations should correspond with actual values measured on physical devices once manufactured.

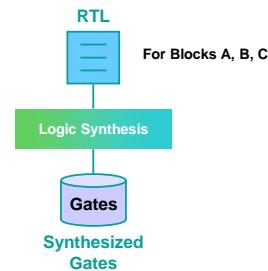


The function of logic synthesis is to minimize the area in terms of cell count and cell size. Logic synthesis minimizes power in switching activity in individual gates, de-activated circuit blocks, and leakage power reduction. It maximizes performance in terms of the maximum clock frequency of synchronous systems and throughput for asynchronous systems. It quickly produces accurate functional models where the gate-level model is functionally equivalent to the RTL model. Using logic synthesis tools produces a gate-level netlist in less time than the time it would take an experienced logic designer to create the same model. Logic synthesis produces predictable and accurate results. Timing, area, and power consumption calculations should correspond with the actual values measured on the physical device once manufactured.

Example: Logic Synthesis

We use the RTL for blocks A, B, and C to produce the following netlists:

```
block_a.vg
block_b.vg
block_c.vg
```



At the top-level EX, the module is instantiated:

```
// top.vg
module ex (...);
    block_a u0 (...);
    block_b u1 (...);
    block_c u2 (...);
endmodule
```

```
block_a.vg
block_b.vg
block_c.vg
top.vg
```

589 © Cadence Design Systems, Inc. All rights reserved.



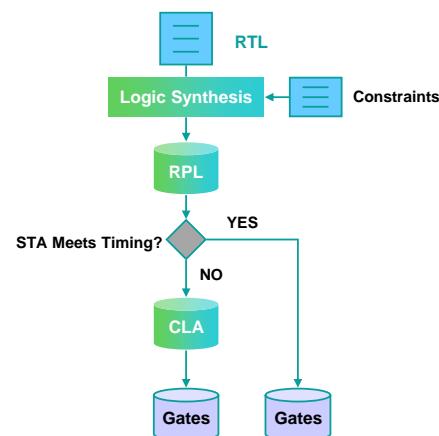
Let's understand the logic synthesis process with this example.

As an input for the synthesis, you provide RTL for blocks A, B, and C. The synthesis process optimized this RTL to the gate-level netlist as block-a, block-b, and block-c.

And these block-level modules are instantiated at the top level to form the final netlist.

STA Usage During Logic Synthesis

- Let's assume the logic synthesis tool has two adder architectures to choose from: RPL and CLA.
- The logic synthesis tool first implements an RPL adder, then performs static timing analysis.
 - If the design meets timing, it creates a netlist based on the RPL adder.
 - If the design does not meet timing, it modifies the architecture and creates a netlist based on the CLA adder.



590 © Cadence Design Systems, Inc. All rights reserved.



So, how is timing optimized during logic synthesis?

Let's assume there are two adder architectures for a given design, Ripple carry adder and Carry look ahead adder, available for a synthesis tool to choose from. Then, in this case, the tool first implements the ripple carry adder and performs static timing analysis to optimize the timing. If the timing results are good and the design can meet timing, a netlist is created based on the Ripple carry adder.

But, if the design fails to meet timing, then the tool modifies the architecture and implements the carry look-ahead adder, and consequently, the netlist is based on CLA adder architecture.

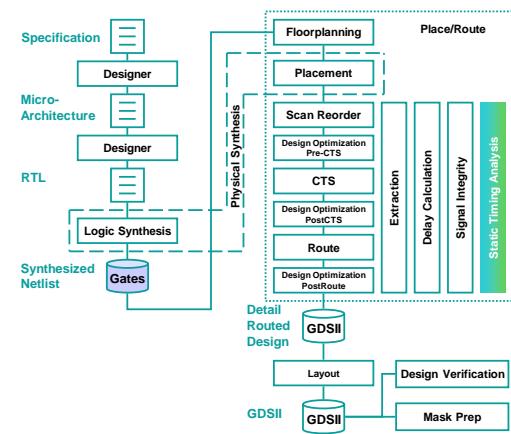
So, during the Static Timing Analysis process in synthesis, the tool looks for that architecture that helps optimize the timing in the best possible way.

What Is Static Timing Analysis (STA)?

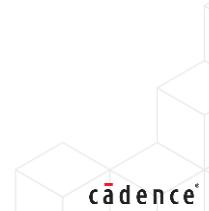
The process of computing the timing of logically-related paths for a digital design without regard to large-scale functional behavior.

The preferred method for timing signoff.

Example: To determine the timing of the design, we ran static timing analysis after detail route and saw several paths violating their setup time requirements.



591 © Cadence Design Systems, Inc. All rights reserved.



Static timing analysis is the process of adding the delays in the paths and verifying the path delay against the required timing for the digital design to work. Static timing analysis is an integral part of design closure. First, STA calculates the path delays for optimization tools. Then, based on the path delays, the optimization tools choose cells from the timing library to create a circuit that meets your timing requirements. Second, STA analyzes the timing of a circuit to verify that the circuit works at the specified frequency. During timing closure, the timing analysis can be run after the detail route, and several paths can be checked for their setup and hold time requirements.

Static Timing Analysis: Input, Output, and File Format

Static Timing Analysis

- Input

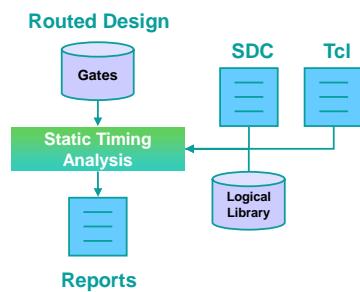
- Gate-level design

Note: STA can be run on a design at any stage of the back-end flow.

- Constraints in Standard Design Constraints (SDC) format
 - Logical timing libraries in Liberty (.lib) format
 - Constraints and commands in Tcl

- Output

- Timing reports



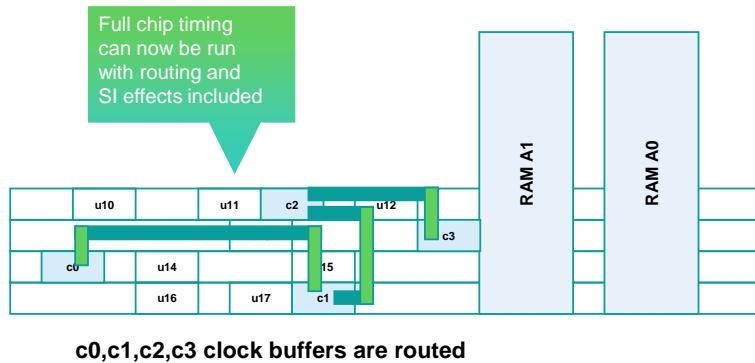
To complete the timing analysis, the inputs required are a gate-level netlist, a Verilog netlist, design constraints in the form of SDC, timing libraries which are the .libs with timing data, and the SPEF file for reading the RC extraction data. At the end of the timing analysis, the output is in the form of timing reports.

Example: Static Timing Analysis

At the end of the physical implementation phase, we will need to run signoff STA to make sure that all of the paths in our design meet timing.

STA can be used:

- During the implementation phase to check on timing, etc.
- For signoff just before tapeout to ensure all paths meet timing.



593 © Cadence Design Systems, Inc. All rights reserved.

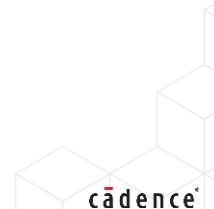


In a design, you can run timing analysis multiple times to make sure that all the paths in your design meet timing requirements. It can be done during the physical implementation phases, at pre-CTS or post-CTS, or after routing. At the end of physical implementation, we will need to run signoff STA to verify the timing. STA can be done just before tapeout to ensure the design is closed with respect to timing.

What Are Timing Constraints?

Timing constraints represent the performance goals for your designs.

- Software tools use the timing constraints to guide the timing-driven optimization tools in order to meet these goals.
- Some of the timing constraints that the STA tool follows are:
 - Clocks definition
 - Input delay/arrival time
 - Output delay/required time
 - Operating conditions



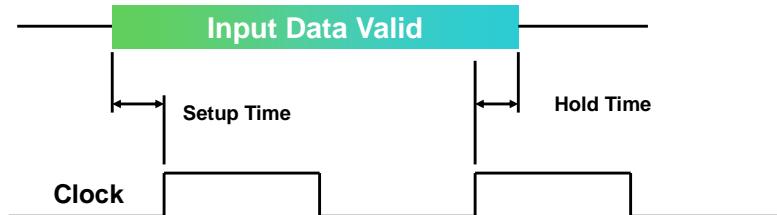
Timing constraints provide the specifications that our design must meet. These can be the constraints for clocks, external constraints, power constraints, yield constraints, net delay constraints, environmental constraints, and even constraints for design rules for manufacturing. STA tools have a specific syntax and conform to Tcl syntax. All these constraints are specified to achieve the common goal of meeting the timing for the digital design.

What Are Setup Time and Hold Time?

Setup Time: The time a synchronous input must be stable before active clock edge.

Hold Time: The time a synchronous input must be stable after active clock edge.

Synchronous inputs have setup/hold specifications relative to the clock.



595 © Cadence Design Systems, Inc. All rights reserved.



Setup Time is the amount of time for a synchronous input to be stable before the capture edge of the clock. This is so that the data can be stored successfully in the storage device.

Setup violations can be remedied by either slowing down the clock or by decreasing the path delay.

Hold Time is the amount of time that the synchronous input must maintain its value after the capturing edge of the clock so that the data can be stored successfully in the storage device.

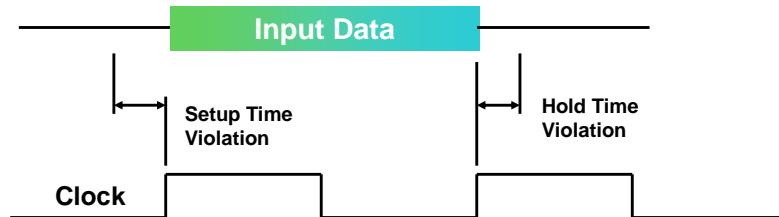
Holds are independent of clock frequency.

Hold violations may be remedied by increasing the delay of the data path or by decreasing the clock uncertainty.

Setup Time and Hold Time Violations

A setup time violation is when a signal arrives too late and misses the time when it should advance.

A hold time violation is when a signal arrives too early and advances one clock cycle before it should.



The time when a signal arrives can vary due to many reasons. The input data may vary, the circuit may perform different operations, the temperature and voltage may change, and there are manufacturing differences in the construction of each part. The main goal of static timing analysis is to verify that, despite these possible variations, all signals will arrive neither too early nor too late; hence, proper circuit operation can be assured.

Timing Report for Setup Violations

Path 1: **VIOLATED Setup Check** with Pin reg_2/CK

Endpoint: reg_2/D (v) checked with leading edge of 'CLK1'

Beginpoint: reg_1/Q (v) triggered by leading edge of 'CLK1'

Other End Arrival Time 0.104

- **Setup 0.167**

+ Phase Shift 2.000

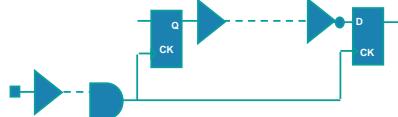
= Required Time 1.937

- Arrival Time 1.946

= **Slack Time -0.009**

 Clock Rise Edge 0.000

= Beginpoint Arrival Time 0.000



Instance	Arc	Cell	Delay	Arrival Time	Required Time
clk_0	A ^ -> Y ^	BUFX2	0.091	0.091	0.178
clk_1	A ^ -> Y ^	BUFX2	0.097	0.188	0.275
clk_2	A ^ -> Y ^	BUFX2	0.094	0.282	0.369
clk_3	A ^ -> Y ^	BUFX2	0.092	0.374	0.462
clk_4	A ^ -> Y ^	CLKAND2X2	0.150	0.524	0.612
reg_1	CK^ -> Q v	DFFRHQX1	0.288	0.812	0.900
t_1	A ^ -> Y ^	BUFX8	0.111	0.923	1.011
t_2	A ^ -> Y ^	BUFX8	0.092	1.015	1.103
t_3	A ^ -> Y ^	BUFX8	0.092	1.107	1.195
t_4	A ^ -> Y ^	BUFX8	0.092	1.199	1.287
t_5	A ^ -> Y ^	BUFX4	0.132	1.331	1.379
t_6	A ^ -> Y ^	BUFX8	0.092	1.423	1.471
t_7	A ^ -> Y ^	BUFX6	0.112	1.535	1.563
t_8	A ^ -> Y ^	BUFX8	0.092	1.627	1.655
t_9	A ^ -> Y ^	BUFX4	0.128	1.755	1.747
t_10	A ^ -> Y ^	BUFX8	0.088	1.843	1.835
t_11	B ^ -> Y ^	NAND2X1	0.066	1.909	1.901
t_12	A ^ -> Y ^	INVX1	0.037	1.946	1.937
reg_2	D v	DFFRHQX1	0.000	1.946	1.937

597 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Another end arrival time is the capture clock path from the clock source to capture the flop register.

This is a setup violation report because:

The setup constraint for the flip-flop is subtracted from the End arrival time to get the required time.

The Arrival time is subtracted from the required time.

Timing Report for Hold Violations

Path 1: VIOLATED Hold Check with Pin reg_3/CK

Endpoint: reg_3/D (v) checked with leading edge of 'CLK1'

Beginpoint: reg_1/Q (v) triggered by leading edge of 'CLK1'

Other End Arrival Time 0.973

+ Hold 0.179

+ Phase Shift 0.000

= Required Time 1.152

Arrival Time 1.099

= Slack Time -0.053

Clock Rise Edge 0.000

= Beginpoint Arrival Time 0.000

Instance	Arc	Cell	Delay	Arrival Time	Required Time
clk	^			0.000	0.088
ck_0	A ^ -> Y ^	BUFX2	0.091	0.091	0.178
ck_1	A ^ -> Y ^	BUFX2	0.097	0.188	0.275
ck_2	A ^ -> Y ^	BUFX2	0.094	0.282	0.369
ck_3	A ^ -> Y ^	BUFX2	0.092	0.374	0.462
ck_4	A ^ -> Y ^	CLKAND2X2	0.150	0.524	0.612
reg_1	CK^ -> Q v	DFFRHQX1	0.288	0.812	0.900
t_1	A ^ -> Y ^	BUFX8	0.092	0.904	0.992
t_2	A ^ -> Y ^	BUFX8	0.092	0.996	1.084
t_15	B ^ -> Y ^	NAND2X1	0.066	1.062	1.115
t_16	A ^ -> Y ^	INVX1	0.037	1.099	1.152
reg_3	D v	DFFRHQX1	0.000	1.099	1.152

598 © Cadence Design Systems, Inc. All rights reserved.



Another end arrival time is the capture clock path from the clock source to capture the flop register.

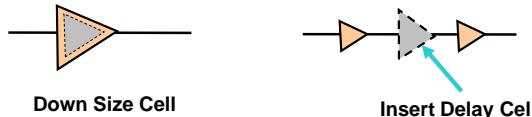
This is a hold violation report. The hold constraint for the flip-flop is added to the Other End arrival time to get the required time. The Required time is subtracted from the Arrival time ending in a negative slack, which is a violation.

Techniques to Reduce Timing Violations

- To fix a setup violation, we need to speed up the delay path causing the violation by:
 - Increasing cell drivability by upsizing the cell.
 - Adding buffers to optimize the critical path and reducing the load on complex gates with large fanout.



- To fix hold violation, we need to make the signal path slow by:
 - Adding delay cells to slow the signal.
 - Reducing drivability of cells.



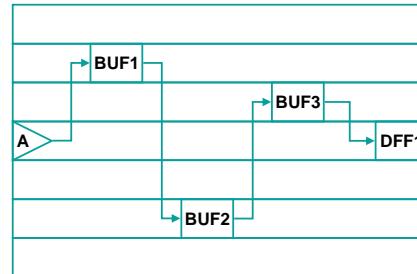
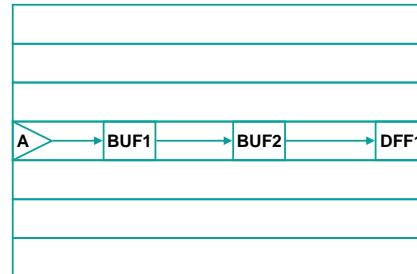
599 © Cadence Design Systems, Inc. All rights reserved.



Timing violations broadly fall into two categories. They are set up and hold. To fix a setup violation, we need to speed up the delay path causing the violation. This can be done by either increasing the cell driving strength by upsizing it or, by adding buffers to optimize the critical path and reducing the load on complex gates with a large fanout. On the other hand, hold violations can be reduced by making the signal path slower by adding delay cells or by reducing the drivability of cells by downsizing them.

STA Usage During Physical Design

- In physical design, STA is also used for optimization decision, just as it is done for logic synthesis.
- Consider the signal A driven by two or three buffers. STA is used during this optimization to determine if the extra buffer is needed in physical design, depending on the:
 - Placement of the buffers.
 - Connections between the buffers.
 - Timing calculated through the path from A to DFF1.



600 © Cadence Design Systems, Inc. All rights reserved.



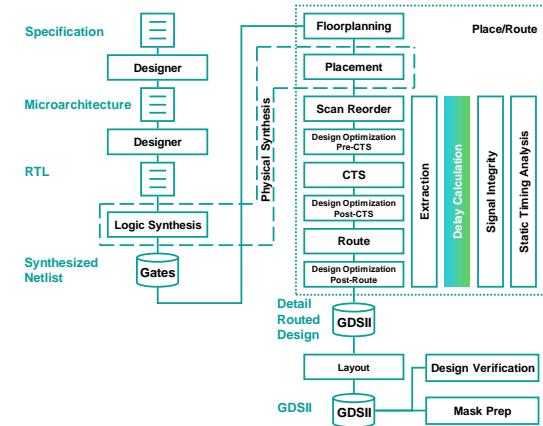
Static timing analysis, or STA, is used for both analysis and timing optimization, for example, physical and logical synthesis. In this example, consider signal A being driven by two or more buffers. STA can be used during this timing optimization to determine if an extra buffer would be needed in the physical design depending on its logical placement, its fanin and fanout logic cone, and the timing calculation through the path from A to DFF1.

What Is Delay Calculation?

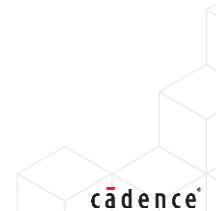
- The process of computing the delay of interconnect and standard cells in a digital design.
- Delay calculation is a part of static timing analysis.

Example:

In the example design, delay calculation was performed after CTS and also after final route. Using the delay information, we were able to find several timing violations in the design.



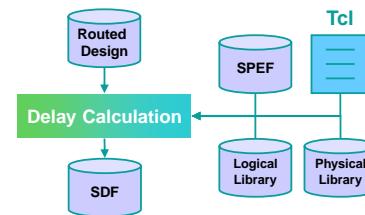
601 © Cadence Design Systems, Inc. All rights reserved.



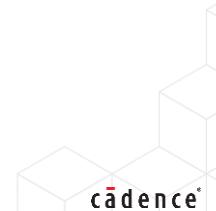
Delay calculation is the process of computing the delay of interconnect and standard cells in a digital design. It is a part of STA and can be performed during physical implementation stages like CTS and postroute. Using the delay information, several timing violations can be caught at the early stages of the design flow and reduce the time to design closure.

Delay Calculation: Input, Output, and File Format

- Input
 - Routed design
 - Parasitic extraction file (SPEF)
 - Logical Timing Libraries in Liberty (.lib) format
 - Physical Libraries in LEF format
 - Constraints and commands in Tcl
- Output
 - Standard Delay Format (SDF) file containing all of the delay information in the design

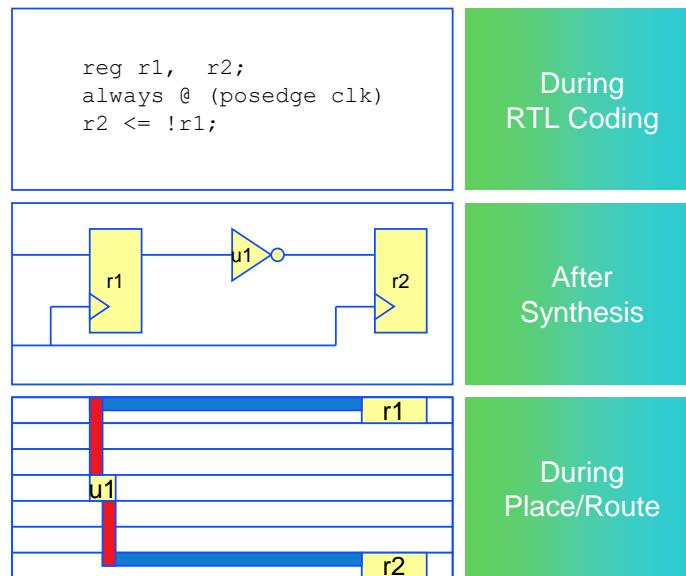


602 © Cadence Design Systems, Inc. All rights reserved.



The delay calculator requires a routed design, SPEF files for reading the RC extraction data, logical timing libraries, which are dot libs with timing data, physical libraries in LEF, design constraints in the form of SDC, and Tcl. The output would be incremental delay format information in the form of SDF.

How Is Delay in a Circuit Estimated or Calculated?



603 © Cadence Design Systems, Inc. All rights reserved.



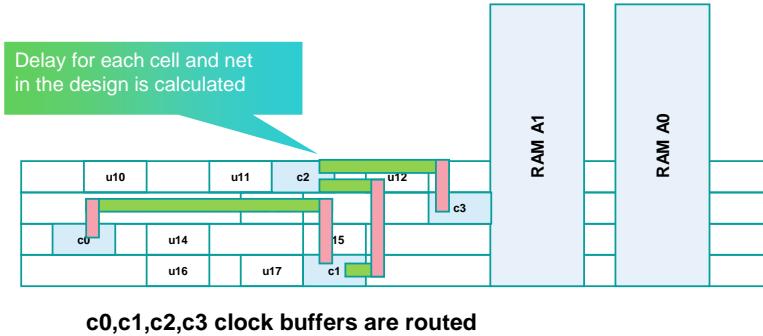
This slide shows how the delay is estimated or calculated in a circuit during various design stages. During behavioral modeling, it is defined using an RTL code. It is depicted using logic gates after synthesis and finally in the form of physical cells during place-and-route.

Example: Delay Calculation

We can perform delay calculations for all the cells and nets in the design and generate an SDF file. This can be done in a:

- Separate delay calculator
- STA tool

The reason for generating an SDF file is to have consistency for all timing calculations throughout the flow. Once it is generated, all tools can access the same SDF file.

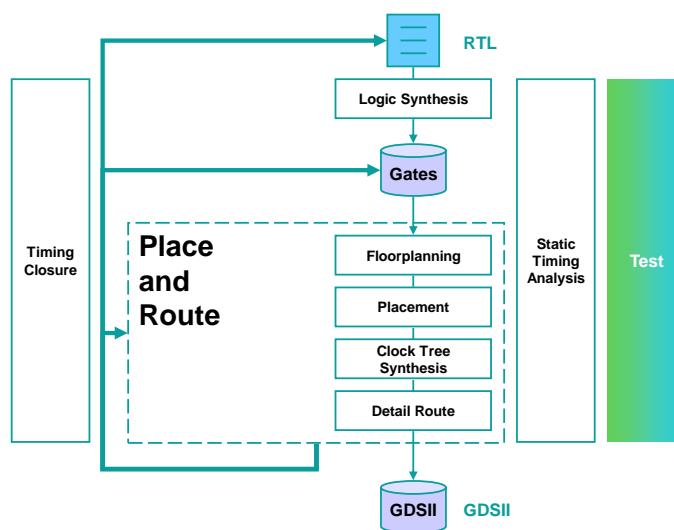


604 © Cadence Design Systems, Inc. All rights reserved.



The delay calculation can be performed for all the cells and nets in the design, and an SDF file can be generated. This can be done with a separate delay calculator or through the STA tool. The reason for generating an SDF file is to have consistency for all timing calculations throughout the flow. Once it is generated, all tools can use the same SDF file.

Implementation Flow Overview



605 © Cadence Design Systems, Inc. All rights reserved.



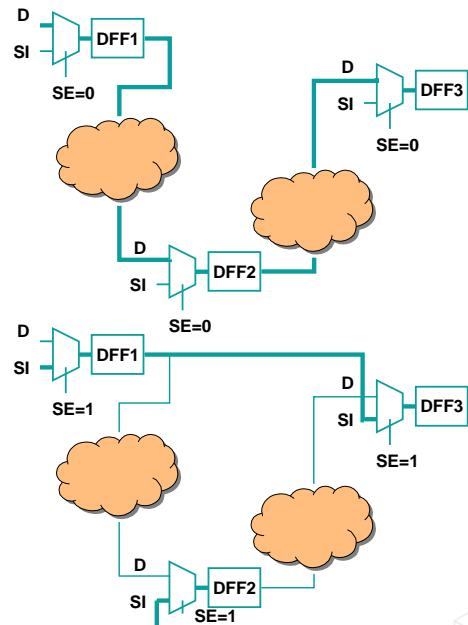
This slide illustrates the implementation flow. The starting point is the RTL which is synthesized into a gate-level netlist by the synthesis tool. After a gate-level netlist is generated, the next steps are a part of place-and-route. Place-and-route includes standard cell and block placement, clock tree synthesis, and routing. At each stage of the flow, timing analysis ensures that the design is converging towards the timing constraints, which is the definition of timing closure.

What Is Design for Test (DFT)?

Design techniques to add testability features in integrated circuits to make it easier to apply manufacturing tests.

In short, the registers in a design are swapped with “scannable” registers or registers with MUXed inputs.

- In normal mode, the registers perform their normal operation.
- In test mode, the registers for scan chains are used by the automated testers for manufacturing tests.



606 © Cadence Design Systems, Inc. All rights reserved.

cadence®

Let's understand DFT, also known as Design for Test.

The test process refers to testing the ASIC for manufacturing defects on Automatic Test Equipment or ATE. It is the process of analyzing the logic on the chip to detect logic faults.

Before inserting test points in the design, you should investigate the DFT requirements of your design.

Let's see what a design with a test circuit looks like. The circuit here shows normal flops and combinational logic, with data in, data out, and clock inputs for the registers.

When scan insertion is enabled, these normal flip-flops are replaced by equivalent scan flip-flops from the libraries. These flip-flops have additional ports, which are, scan in, scan out, and shift enable. The test process sets the circuit to one of the three test modes: capture mode, scan shift mode, and system mode. Out of the three modes, the system mode is the circuit's normal or functional operation mode. Any logic dedicated to DFT is not active in system mode. The scan shift mode is the part of the test process in which registers act as a shift register in the scan chain. This is the test mode where the automated testers use the scan chain registers to detect the faults.

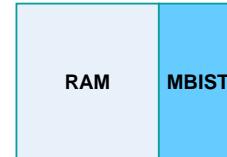
In the third mode, which is capture mode, it analyzes the combinational logic on the chip.

What Is Built-In Self-Test (BIST)?

Extra logic in a design to verify all or a portion of the internal functionality.

In short, BIST structures are added to memory (MBIST) or logic (LBIST) to help improve the overall testing efficiency of the integrated circuit.

Design



The Built-In Self-Test or BIST is inserted into a design to generate self-test patterns. This intent is to verify the internal functionality.

These structures can be added to memory or to logic to improve overall testing efficiency.

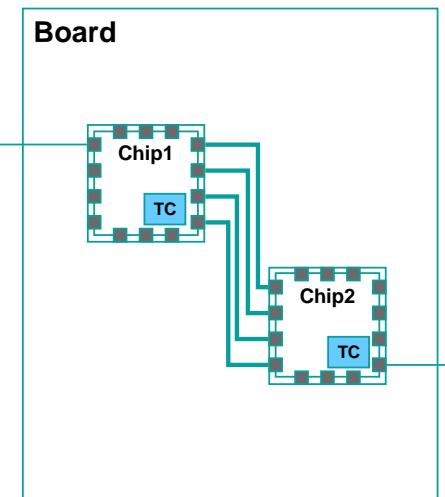
The synthesis tool provides an automated way to insert BIST logic, which is further supported by the ATPG (Automatic Test Pattern Generator) to generate the patterns and observe the responses.

What Is Joint Test Action Group (JTAG)?

Standard used to test board connectivity using boundary scan.

In short, JTAG allows for a standard method to check the traces of a board using boundary scan.

- Each chip has standard ports and a *tap controller* (TC).
- Each chip has special scannable I/Os that form a chain.
- Using boundary scan, patterns are shifted into the I/Os of Chip1 and received on the I/Os of Chip2
- The tester can determine if there are shorts or open in the board traces by comparing the data shifted into Chip1 with the data shifted out of Chip2.



608 © Cadence Design Systems, Inc. All rights reserved.

A JTAG macro contains test logic to access and control DFT features built into the design.

JTAG, or Joint Test Action Group, is an IEEE standard for the control of test access ports and is used for testing designs with boundary scans.

Originally, the JTAG macro was developed to support boundary scan insertion, but its use has since been extended.

The JTAG macro is used to observe the input value and controls the logic of each output by shifting values in and out of the boundary register.

There are several ports included in the JTAG macro, but the minimum required ports are Test Data Input, Test Clock, Test Mode Select, Test Reset, which is optional, and Test Data Output.

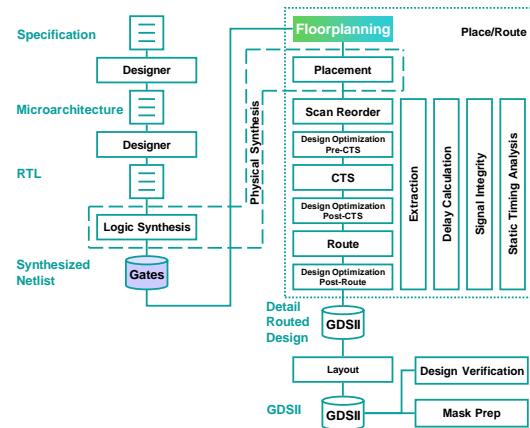
The JTAG macro has standard ports and a tap controller with specific scannable inputs and outputs to create a chain.

In this design, which contains a boundary scan, patterns are shifted into the input ports of Chip1 and shifted out of the output ports of Chip2.

The tester can determine if there are shorts or opens in the board traces by comparing the data shifted into Chip1 with the data shifted out of Chip2.

What Is Floorplanning?

The process of deriving the die size, allocating space for soft blocks, planning power, and macro placement.



609 © Cadence Design Systems, Inc. All rights reserved.

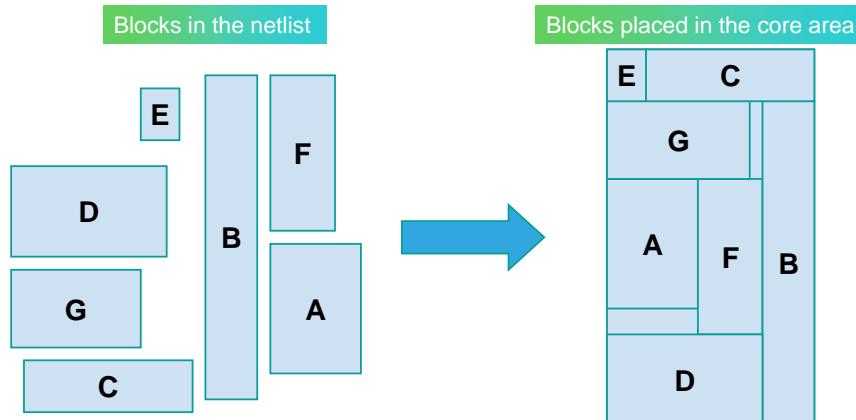


Once you generate a gate-level netlist with the logic synthesis tool, the next step is floorplanning. Floorplanning is the implementation step where you derive the die size and allocate spaces for soft blocks if you have a hierarchical design. Plan the power and place the critical macros. At this stage, you might want to experiment with different floorplans and prototype the design before selecting the floorplan that is going to meet your power, performance, and area or PPA target.

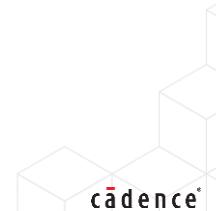
What Is Floorplanning?

(continued)

Example: The blocks in the netlist are placed in the core area, making the best use of the available space. The objective is to reduce the routing between the blocks and, thus, improve the timing and routability of the design.



610 © Cadence Design Systems, Inc. All rights reserved.



If there are blocks instantiated in the netlist, the floorplan would include the blocks placed in locations that would lead to the best timing and congestion for the design. In this example, the blocks are shown as being placed to best utilize the available area to arrive at a compact floorplan.

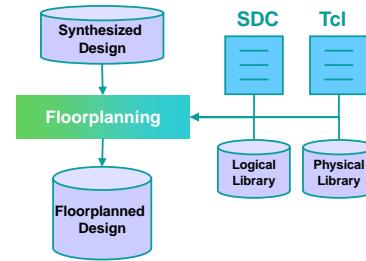
Floorplanning: Input, Output, and File Format

- Inputs

- Gate-level netlist from the output of logic synthesis.
- Constraints in SDC (Standard Delay Constraints) format are needed so that timing with Static Timing Analysis can be accurate and measured against the specifications of the design.
- Timing library (.lib) contains the timing information for each discrete logic gate or macro.
- Physical library (LEF) contains information about the shape and connectivity of the technology library cells.

- Outputs

- Floorplan of the design.



611 © Cadence Design Systems, Inc. All rights reserved.



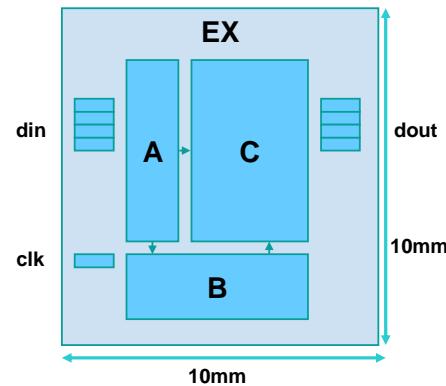
Floorplanning is a mostly interactive step, although a script can generate an initial floorplan that can be the starting point of the step. The inputs are a gate-level netlist, timing constraints in SDC format, timing libraries, and a LEF file. The gate-level netlist is the output of the previous synthesis step.

Timing constraints specify the timing characteristics of the paths in the design. They are used by the static timing analysis tool to determine if the design meets timing or if, instead, there is negative slack in the design. Timing libraries contain lookup tables that include timing information, for example, cell delays, transition times, and setup and hold times for all cells in the library. The LEF file contains the physical characteristics of the cells and blocks, specifically their size and shape, as well as the internal connectivity within the cells. The result of floorplanning is the floorplan of the design.

Example: Floorplanning

With a top-level netlist, we can begin to floorplan the chip.

- Set die size to 10x10 mm².
- Assign the *din*, *clk*, and *dout* I/Os to the perimeter.
- Create hard blocks for A, B, and C.
- Size the blocks A, B, and C.
- Perform power planning.
- Perform macro placement.
- Check for early routing congestion.
- Check for early block utilization.



612 © Cadence Design Systems, Inc. All rights reserved.



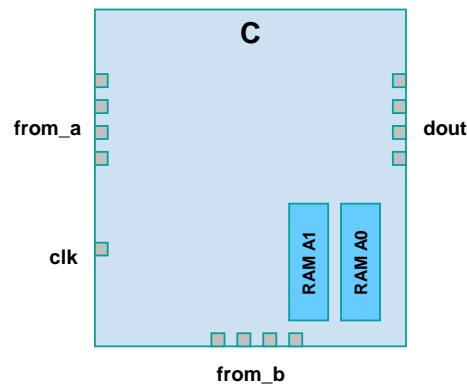
As mentioned earlier, floorplanning is a process of deriving the die size, allocating spaces for soft blocks, planning power, and macro placement. In this example, we have a 10x10 micron die size. The I/Os have been placed around the periphery. If this is a block-level design, instead of pads, these should be pins. For hierarchical implementation, soft blocks A, B, and C have been created. Optionally, you can place the critical macros, such as the RAMs, as shown in this example. Perform power planning, which defines the global VDD and VSS. The common tasks for floorplanning include performing macro placement, checking for early route congestion, checking for early block utilization, power domain definition, and flip-chip bump placement.

Example: Floorplanning (continued)

For each block, we can also perform some early checks.

- Assign pins.
- Place RAMs and macros.
- Check the power plan.
- Check for early routing congestion.
- Check for early block utilization.

It is important to make sure the floorplan is routable and meets the utilization requirements with a given RAM and macro placement, pin assignment, etc.

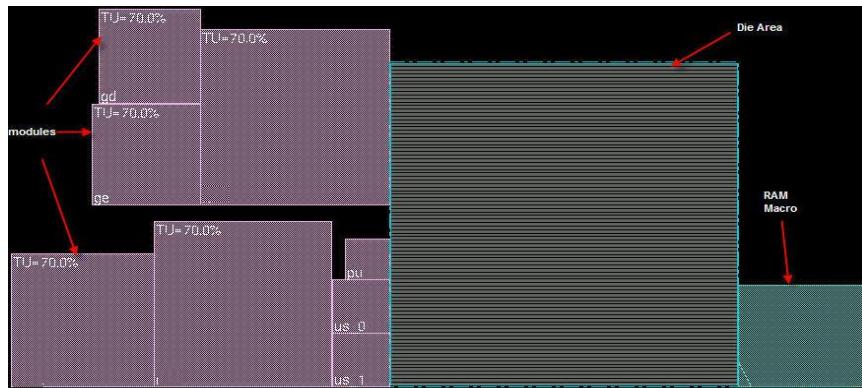


613 © Cadence Design Systems, Inc. All rights reserved.

If the floorplan is for a block, an additional task will include assigning pins on the periphery of the block. Other floorplanning tasks include placing RAMs and other critical blocks, checking the created power plan for IR drops, running a fast route to check for route congestion or routability, and also checking the placement utilization. Running checks during floorplanning will prevent downstream issues later in the implementation flow.

How to Floorplan

- When the design is imported into the tool, a default die size is calculated and displayed, and each module is assigned a physical representation using a default placement density of 70% and an aspect ratio of 1.
 - Each unit represents a particular module in the design.
 - Floorplanning allocates position and area to each unit.



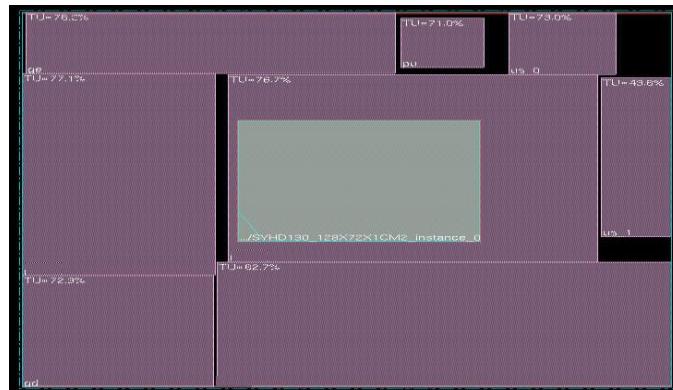
614 © Cadence Design Systems, Inc. All rights reserved.



What you see here is the design after the input files have been imported. The pink boxes represent the modules in the Verilog netlist with a default average utilization of 70%. This utilization number is the starting point density of the modules consisting of standard cells which belong to that particular module. The macros are shown on the right-hand side in blue. In this case, it's a RAM macro. The die area, including the I/Os, is shown in the center.

How to Floorplan (continued)

2. Position the modules and blocks in the die area. In general, position the modules and blocks such that the area of the bounding rectangle is minimum or meets the die size requirement. Try different orientations, aspect ratios, and placement densities of the modules to puzzle-fit them into the die area.
 - The bounding rectangle represents the die area.



615 © Cadence Design Systems, Inc. All rights reserved.



The pink modules that you see are the module guides that contain the standard cells in the design. The process of floorplanning includes trying various orientations, aspect ratios, and placement densities to fit all the modules in the core area. Various iterations may be required before you arrive at the best floorplan to meet your timing and congestion goals.

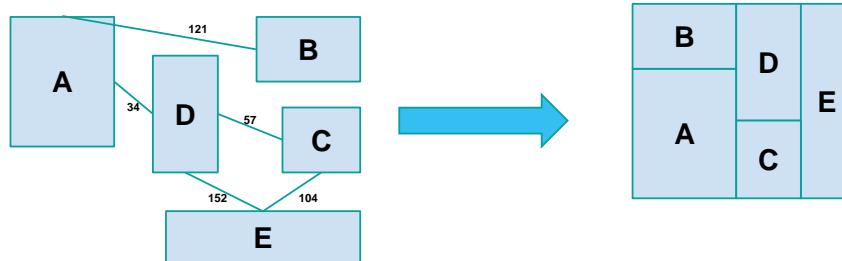
How to Floorplan

(continued)

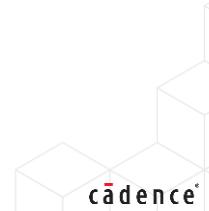
3. Identify modules that should be placed close together.

- The tool shows flightlines (lines showing the number of connections) between the modules. The higher the flightlines between two modules, the closer these modules will have to be within the design.
- Flightlines indicate how much communication occurs between two modules.

The diagram below shows how to floorplan optimally. The numbers over the flightlines indicate the number of nets between corresponding modules.



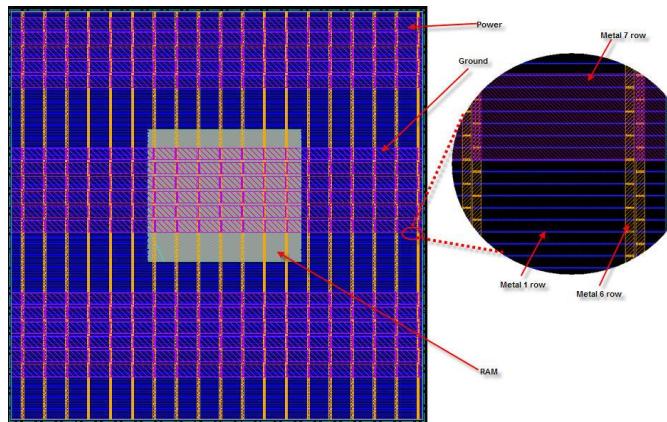
616 © Cadence Design Systems, Inc. All rights reserved.



Start by identifying modules that should be placed close together. The place-and-route tool displays flightlines. Flightlines show the number of connections between modules. The higher the number of flightlines between modules, the closer these modules should be placed within the design. Flightlines indicate how much routing is needed between modules. Modules that need a greater number of routes between them need to be placed close together to optimize their eventual routes. The diagram shows how to arrive at an optimal floorplan based on how many connections, or nets, they have between them. The numbers you see above the flightlines indicate the number of nets that connect between corresponding modules.

What Is Power Planning?

The task of creating the global power plan for a design. These are typically created as VDD/VSS rings and stripes.



617 © Cadence Design Systems, Inc. All rights reserved.



Power Planning is the process of designing a global power distribution network that supplies sufficient power and ground nets to all the instances of the design. Power planning includes sizing the power wires and choosing the optimum metal layers necessary to deliver the required power to different parts of the chip. If you have inadequate power distribution, it can affect chip timing due to excessive rail voltage drop and ground bounce. It can lead to device failure due to electromigration effects. The negative effects of IR drops and other power-related issues can be mitigated by a good power-grid design and sufficient VDD and VSS pads.

What Are Voltage (IR) Drop and Electromigration (EM)?

Voltage (IR) drop is the voltage drop across a chip's power network caused by current and resistance associated with the power network.

Electromigration (EM) is the mechanical failure of metal wires because of metal atoms migrating over a long period of time due to high current densities, causing open circuits, short circuits, or unacceptable increases in resistance.



Voltage drop or IR drop is the drop across a power network caused by current and resistance associated with the network. Excessive IR drop can affect a chip's performance, and therefore it is important to design a power grid that does not have a drop that exceeds acceptable levels. Electromigration is when routed wires fail because of the phenomena where electrons migrate, resulting in opens, shorts, or unacceptable levels of resistance. If electromigration is not dealt with, it can lead to a multitude of failures in the chip.

Power Planning Goals

- To design a global power distribution network that supplies the appropriate power and ground nets to all the instances of the design.
- To size the power wires and choose the metal layers necessary to deliver the required power to different parts of the chip without causing failure.

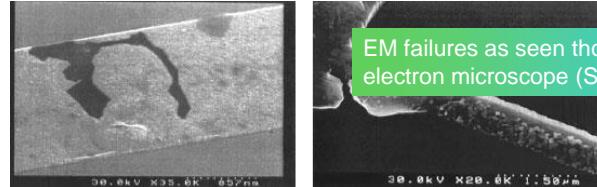


The goal of power planning is to design a robust power network. It consists of creating a global power distribution network that supplies the appropriate power and ground nets to all the instances of the design. Power planning sizes the power wires and selects the metal layers necessary to deliver the required power to different parts of the chip without causing excessive IR drops or electromigration violations.

Need for Power Planning

Power-related issues can:

- Affect chip timing due to excessive rail voltage drop (“IR drop”) and ground bounce.
- Lead to complete device failure due to electromigration effects.



The effects of IR drop, and other power-related issues can be limited by:

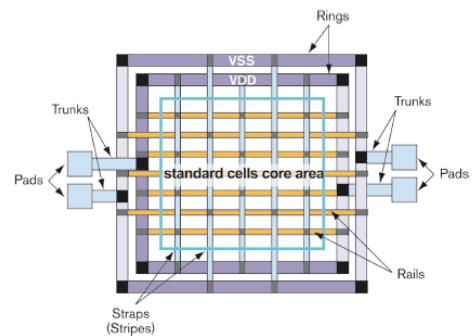
- Good power-grid design.
- Sufficient VDD and VSS pads.

Power planning is a critical aspect of chip design as it affects the timing as well as its reliability. Where excessive IR drop and ground bounce can affect the timing, electromigration effects can lead to device failure from shorts and opens. These issues can only be addressed by a good power grid design as well as an efficient power delivery network in the chip.

Basics of Power Planning

Ensure adequate power and ground connections by including the following basic elements in the power network.

- Power pads that supply power to the chip.
- Power rings around the periphery of the die that carry power to the standard cells and macros.
 - Rings are put on higher-level routing layers leaving the lower layers for signal routing.
- Power rails and trunks that cross the entire die or sections of the die.



621 © Cadence Design Systems, Inc. All rights reserved.



Inadequate power distribution will lead to IR drop issues which affect chip performance. Therefore, ensure adequate power and ground connections by including the following elements.

First, an adequate number of power pads supply power to the chip.

Second, power rings around the periphery of the core, which provides power to the standard cells and macros.

It's recommended that rings are created at higher-level routing layers, thus leaving the lower layers free for signal routing.

Third, ensure that an adequate number of power rails and stripes that cross the entire die, or sections of the die, are created.

Early Planning for Power

- Simulation of major power dissipation components.
- Quantification of chip power:
 - Total chip power
 - Maximum power density
 - Total chip power fluctuations
 - Power grid analysis.
- Allocation and coordination of chip resources:
 - Wiring tracks for the power grid
 - Low Vt devices
 - Dynamic circuits
 - Clock gating
 - Placement and quantity of decoupling capacitors

622 © Cadence Design Systems, Inc. All rights reserved.



Early power planning includes early rail analysis, which provides statistics on the major power dissipation components in the design.

The quantification of chip power involves estimating the following properties.

Total chip power, maximum power density, total chip power fluctuations, and running power grid analysis.

The allocation and coordination of chip resources include:

Assigning routing tracks for a power grid, inserting low-Vt devices, evaluating dynamic circuits that consume more power, inserting clock gating circuits and components, and determining the placement and quantity of decoupling capacitance.

Types of Power Planning

- Trunks and rings:
 - Used for upper-level routing.
 - Rings are placed around blocks to ensure even power distribution within the block.
- Uniform grid:
 - Usually used inside lower-level partitions.

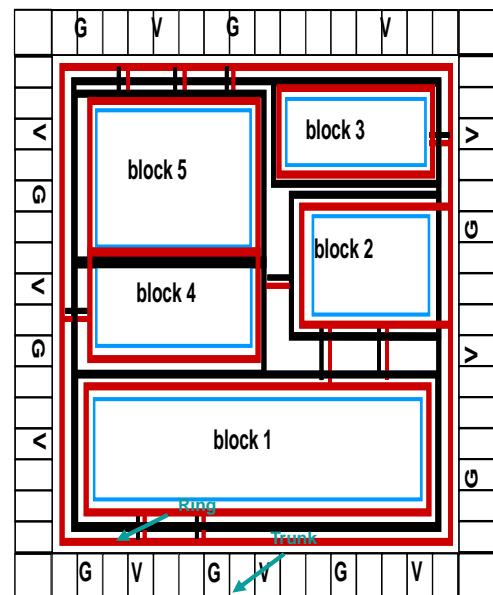


The main types of power planning are trunks and rings. It is recommended to reserve upper-level routes for the global power plan, which includes rings, trunks, and stripes. The lower-level layers would then be available for signal routing. Blocks in the design can have local block rings for optimal power distribution.

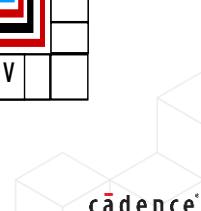
A uniform power grid inside blocks, which are also referred to as partitions, ensures an even power distribution.

Trunks and Rings Methodology

- Each block has its own ring structure.
- Each block has a trunk that connects the top level to the block.
- Rings can be shared between abutted blocks.
- Requires fewer routing resources.
- Changes in design may require changes to the power structure.



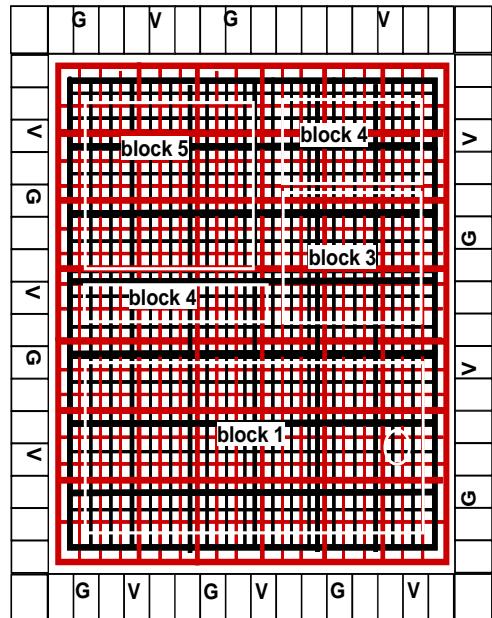
624 © Cadence Design Systems, Inc. All rights reserved.



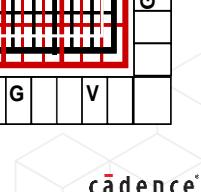
This example shows a combination of trunks and rings. Notice that each block, 1 through 5, has its own ring structure. You can see that the rings connect to the global or top-level ring with additional VDD and VSS power routes. Rings can be shared between abutted blocks, as sharing rings uses fewer routing resources. Changes in the design or the floorplan may require changes to the power structure.

Uniform Chip Grid Methodology

- Robust and redundant power network.
- Seen in microprocessors and high-end large ASICs.
- Primary distribution through upper metal layers.
- Grids of different blocks need to align with each other.



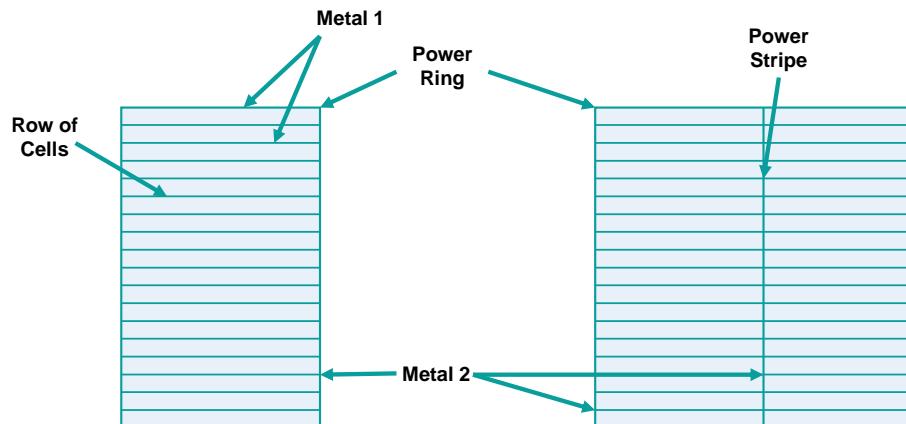
625 © Cadence Design Systems, Inc. All rights reserved.



This example illustrates a robust power network created as a power grid. This type of structure is utilized by microprocessors and high-end large ASICs. Power distribution is through upper metal layers. Power grids that cross multiple blocks need to align with each other.

Power Planning: Power Stripes

- Power stripes
 - Specified and created by the chip designer, typically using a place-and-route tool.
 - Distribute power vertically within a ring.
 - Typical power routing routes horizontally in Metal 1 (including standard cell row power rails) and vertically in Metal 2.



626 © Cadence Design Systems, Inc. All rights reserved.

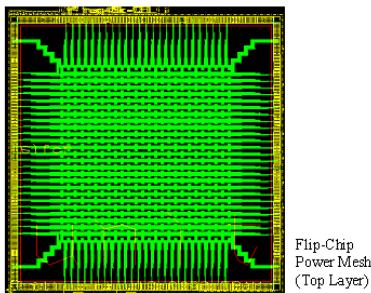


Here you see that power stripes are created over placement rows using a place-and-route tool. In this example, they are created in the vertical direction, but they can be created in both a vertical and a horizontal direction for a grid structure. Usually, the vertical power stripe is created in the same direction as the preferred routing direction of Metal1, and the horizontal power stripe is created in the same direction as the preferred routing direction for Metal2. These global stripes will connect to the global rings to complete power planning.

Power Planning: Power Mesh

- Power mesh
 - Meshes are created to cover large areas of a chip.
 - Created by layers of power straps going in alternate vertical and horizontal directions.
 - Distributes power across a chip so that IR drop and electromigration targets are met.

Example



627 © Cadence Design Systems, Inc. All rights reserved.



Power meshes are created to cover large areas of a chip. To create power meshes, draw stripes in both vertical and horizontal directions. Creating a power mesh distributes power across a chip so that IR drop and electromigration targets are met. The diagram shows a flip-chip with the power mesh cerated on the top layer.

Uses of Followpins and Connect Ring

Followpins are used to:

- Route power/ground along the standard cell rows.
 - Follows the pins of each cell and stitches them together.
- Connects these routes to power rings (and vertical stripes).

Connect ring is used to:

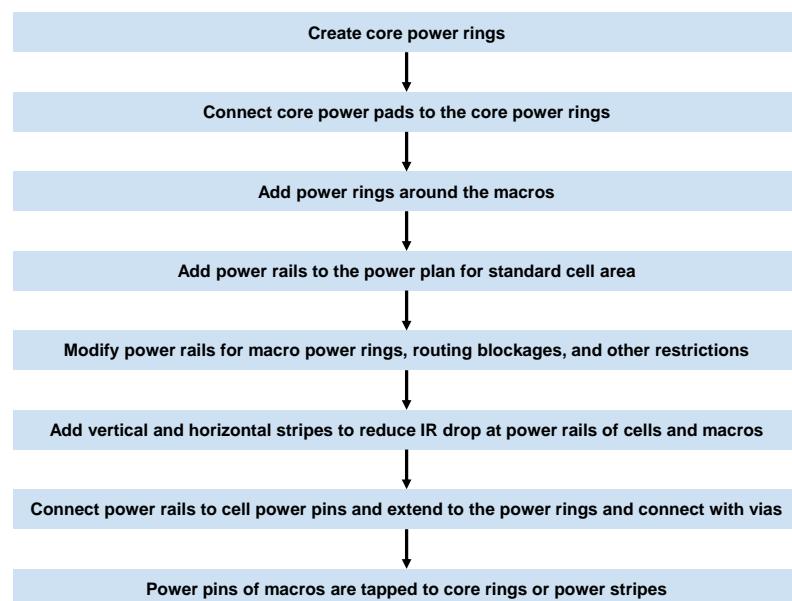
- Connect dangling power routes to stripes/rings.
- Connect power rings to I/O power pads.



Follow pins are a type of power route that is used to route power and ground along the standard cell rows so that the VDD and VSS pins of the standard cells can connect to the global power. Follow pins follow the power pins of each cell, and stitch them together. Follow pins connect these routes to power rings and to vertical stripes.

Routes to connect to rings are used to connect dangling power routes to stripes or rings. They also connect power rings to I/O power pads.

Steps Involved in Power Routing



629 © Cadence Design Systems, Inc. All rights reserved.



These are the steps to power planning. Create the core power rings around the die. This is the start of creating a global power plan. Then, connect the power pads to the power pins. Determine during early rail analysis if power rings also need to be created around the macros. Add power rings around macros as identified by the early rail analysis. Add power rails or follow pins for the standard cell power connections. Add connections from the macro power pin to the global power. Add routing blockages to prevent signal routing in areas that were reserved for adding power rings around macros. Add stripes in the vertical and horizontal directions. Connect all power rails to the standard cell follow pins and extend the power stripe connections to the rings. Vias will be used to jump between layers. The power pins of macros will also be connected to the global power plan with power routing.

An important point to keep in mind is the distinction between power planning and power routing.

Power planning is the creation of global VDD and VSS as a part of the global power plan.

Power routing is the creation of follow pins and the final hookup for all the power to the global nets.

Power Consumption and Its Components

- Power on a chip is consumed when it is active (dynamic power) as well as inactive (leakage power).
- Leakage power:
 - Power consumed when cells are not switching.
 - Main sources of leakage power are sub-threshold leakage currents, which reduce linearly with supply voltage (V_{dd}) and exponentially with a threshold voltage (V_{th}).
- Dynamic power:
 - It is the power associated with the switching of nets and cells.
 - It is calculated as **Power = $f \times C \times V^2$** .
- How can the power consumption on a chip be reduced?

630 © Cadence Design Systems, Inc. All rights reserved.



There are two components of power dissipation, active or dynamic power, and inactive or leakage power.

Leakage power is when power is consumed when cells are not switching. The main sources of leakage power are sub-threshold leakage currents, which reduce linearly with supply voltage (V_{dd}) and exponentially with a threshold voltage (V_{th}).

Dynamic power is the power associated with the switching of nets and cells.

It is calculated as $\text{Power} = fCV(\text{square})$.

Since power dissipation is something we want to control, the question is, how can we reduce the power consumption on a chip?

Multiple Supply Voltages

- Using multiple supply voltages is one method of reducing a chip's power consumption.

- It aims at minimizing the supply voltage level wherever possible.

Instead of the chip operating from a single uniform supply voltage, a range of supply voltages are assigned to different areas of the chip => Voltage islands.

- It also assigns separate power-nets to different blocks and steps the power-net voltages down wherever the chip and block performance allow.



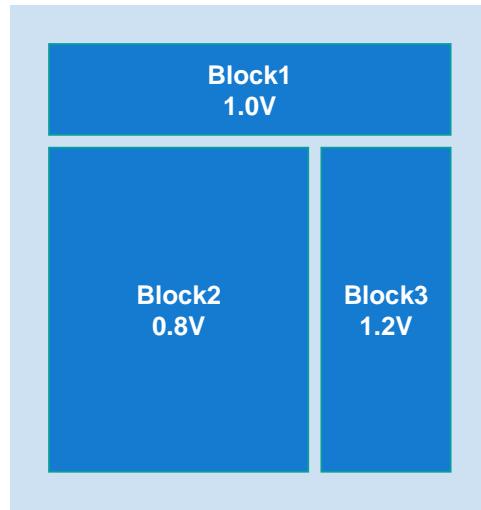
Using multiple supply voltages or voltage islands is one method of reducing a chip's power consumption. By creating voltage islands with their localized power sources, the supply voltage level can be lowered wherever possible.

Instead of the chip operating from a single uniform supply voltage, a range of supply voltages are assigned to different areas of the chip in these voltage islands.

Since the power network has to be localized to the various islands, power planning steps would assign separate power nets to different blocks and steps up or down the power net wherever the chip and block performance allow.

Discussion Question

Assuming the following chip diagram, what considerations should be taken into account when designing a power plan?



632 © Cadence Design Systems, Inc. All rights reserved.



As mentioned in the previous slide, this is a multi-voltage chip with three separate voltage domains.

The reason for doing so is to reduce the overall power of the chip. Maybe Block2 doesn't require the performance of the other blocks, and a move to 0.8V will not affect the block from meeting its performance targets. Lowering its voltage is a good way to reduce the overall power. Block3, on the other hand, has a higher voltage, maybe because it requires more performance. But, by having only Block3 at the higher voltage, the overall chip power is reduced.

The power plan becomes more complicated here. If everything were at 1.2V, then we could simply create a power plan based on 1.2V. Each of these blocks will have its own rings and stripes connected to dedicated power pins. Therefore, each block will have a separate grid.

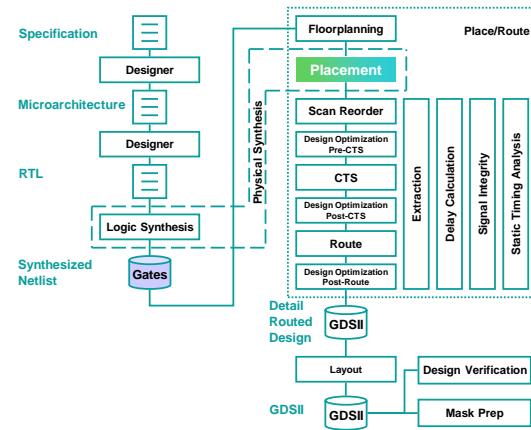
We will discuss level shifters, SRPGs, etc., later in the power section.

What Is Placement?

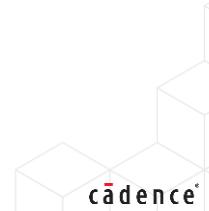
The process of placing the standard cells in a floorplanned design.

Example: After the chip was floorplanned, we performed placement and discovered the floorplan was too small to fit all the cells and macros in the design.

Question: How can we avoid this problem?



633 © Cadence Design Systems, Inc. All rights reserved.



Placement is the next step after floorplanning. Placement is the process of placing the standard cells and blocks in a floorplanned design. When a design is read in, the tool creates rows for the standard cells to be placed into the core area by the placer. There are several different types of rows, for example, standard cell rows and I/O rows. If the chip was floorplanned and then it becomes apparent that the floorplan was too small to fit the design, how can this problem have been avoided? One way to avoid this problem is to run a fast placement after floorplanning to validate that all the components can fit into the floorplan.

Placement Goals

- Goals of the placement step are to:
 - Guarantee that the router can complete the routing step.
 - Minimize all the critical net delays by placing cells close to each other, thus reducing interconnect lengths.
 - Minimize the die size as much as possible.
 - Reduce routing congestions, if any.
- Good placement is essential for meeting timing goals.
- Bad placement can lead to sub-optimal routes and cause paths to fail timing.



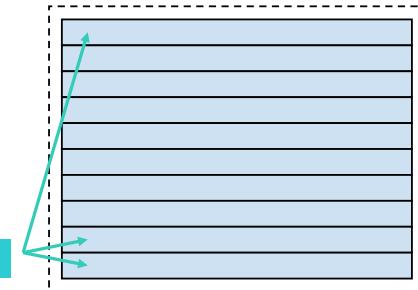
The primary goal of the placement step is to place the standard cells in the placement rows such that the router can complete routing. The placement should reduce all the critical net delays by placing cells close to each other to reduce interconnect lengths. Reducing net lengths alleviates routing congestion and avoids signal integrity issues downstream. The placer should place cells that minimize the die size as much as possible. Placement affects routing, which in turn affects timing. Good placement is essential for meeting timing goals, whereas bad placement can lead to sub-optimal routes and cause paths to fail timing.

Standard Cell Placement

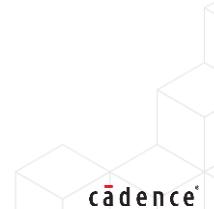
- The core area of the die is defined by specifying the distance between the edge of the layout and the core.
- Standard cells are placed in rows that are drawn within the core area.
- Placement should be legalized, meaning standard cells are placed correctly on the placement grid, not overlapping, and power pins of standard cells are aligned correctly.
- Placement should be routable and meet timing requirements.

VDD	VDD	
	CELL	
GND	GND	

Standard cell rows in core area



635 © Cadence Design Systems, Inc. All rights reserved.

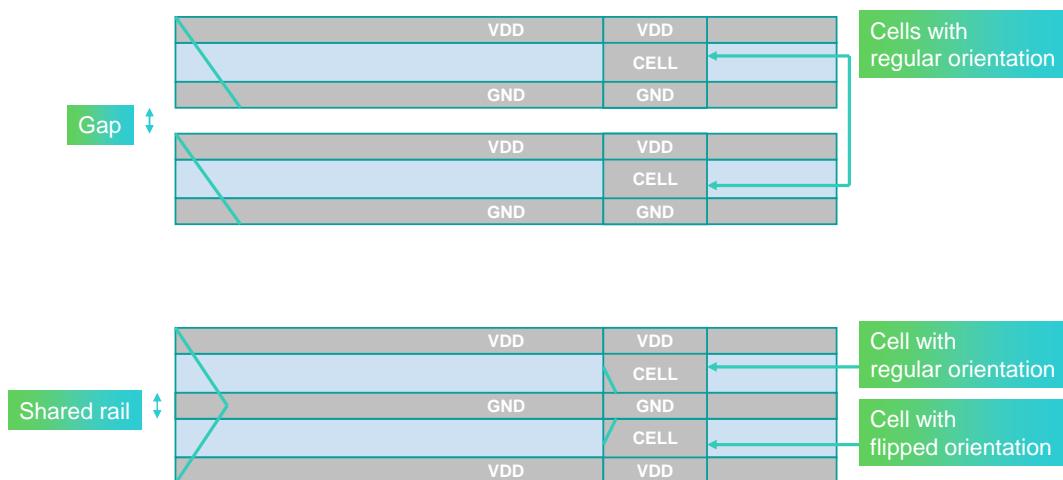


The core area of the die is the distance between the edge of the die to the center, not including the IOs.

Standard cells are placed in rows that are created within the core area. The height of the core rows is derived from information in a technology file called a LEF (Library Exchange Format) file.

The placer will place the cells in legal locations, which are in core rows, without overlaps. The standard cells have preferred orientations defined in the LEF file, and the placement tool will adhere to those defined rules. The placer will ensure that power pins of standard cells are aligned correctly. For the implementation to be successful, the design's placement should be able to be routed and meet timing requirements.

Standard Cell Rows



636 © Cadence Design Systems, Inc. All rights reserved.

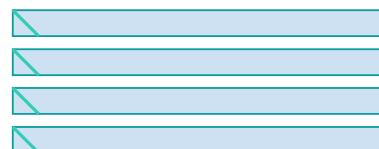
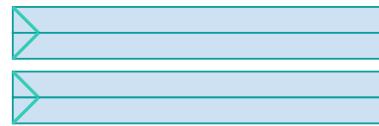
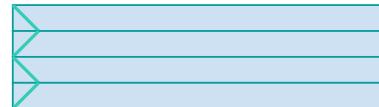


This illustration shows two types of standard cell rows. In the first diagram, there is a gap between rows for standard cell placement. In the second diagram, the rows are flipped and abutted, which means that the power rail can be shared between rows. The second method leads to a more compact placement. Therefore, the second method is the most common style of row creation.

Cell Row Placement

There are three ways to arrange cell rows:

- Sometimes, technology allows rows to be flipped and abutted so the pairs can share power and ground rails. This is the most common approach.
- Second configuration is to flip every other cell row but leave a gap between every two cell rows, mainly for routing purposes. Creates larger power rails and densely packed cell structures.
- Last configuration is to leave a gap between every cell row and not flip the rows. Useful when only two or three metal layers are available for routing.



637 © Cadence Design Systems, Inc. All rights reserved.



There are three ways to arrange rows, as represented in the three diagrams shown.

The first diagram illustrates flipped and abutted rows so that pairs of rows can share power and ground rails. This is the most common approach. When standard cells are placed in the rows, they, too, will be flipped so that they can share the power and ground rails.

The second configuration shows how every other cell row is flipped. However, there is a gap between cell row pairs. The space is to leave room for routing if there is a possibility of over-congestion.

The third configuration illustrates the case where there is a gap between every cell row. The rows will not be flipped. This method is seldom used and is considered useful when only two or three metal layers are available for routing.

How to Place?

- Placement is one of the most classic EDA problems.
- The problem is very hard (NP-complete).
 - There are still many research papers and continuous improvements.
- Scalability/CPU time is important – need to deal with millions of cells.
- Global placement determines the rough location and spread of the cells.
 - Partition-based approach.
 - Analytical approach (formulate into some mathematical programming).
 - Simulated annealing (probabilistic) approach.
- Legalization (remove cell overlaps and snap them into circuit rows).
- Detailed placement (for further improvement).



Placement is one of the more complex EDA challenges.

As a result, there are still many research papers written on the topic, and there are continuous improvements being made.

Challenges include scalability and CPU time. The placer often needs to place millions of cells in an acceptable amount of time.

The first stage of global placement determines the rough location and spreading of the cells.

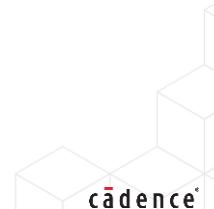
If a partition-based approach is used, the design can be broken into smaller blocks (or partitions), and each block can be placed and routed at the same time, in parallel.

Next, after global placement, placement legalization is run to remove cell overlaps and snap the cells into the placement grid and rows.

Detailed placement may be run for further improvements.

Timing-Driven Placement

- Placement of standard cells takes into account the timing constraints.
 - Placer balances the importance of meeting setup-type timing constraints with routability.
 - Placer identifies critical nets and performs placement to meet the constraints. It pays less attention to meeting timing constraints on non-critical nets but more attention to enhancing routability.
- Why do we need this?
 - Growing interconnect versus gate delay ratios.
 - Higher levels of on-die functional integration make global interconnects even longer.
 - Increased chip operating frequencies that make timing closure tougher.
 - Increased number of macros and standard cells for modern designs.



The placement of standard cells takes into account the timing constraints that are defined in the SDC file, which is read in when the netlist and technology are read in during design import.

The placer balances the priority of meeting setup time constraints along with routability.

The placement tool identifies critical nets and runs the placement algorithm to meet the constraints. It pays less attention to meeting timing constraints on non-critical nets, but for those non-critical nets, attention will be paid to increasing routability and reducing congestion.

Increasing routability and reducing route congestion are important because interconnect delays are more dominant than gate delays. Reducing the interconnect delays will have a positive impact on timing.

Higher levels of on-die functional integration make global interconnects even longer, so anything that can reduce interconnects must be explored.

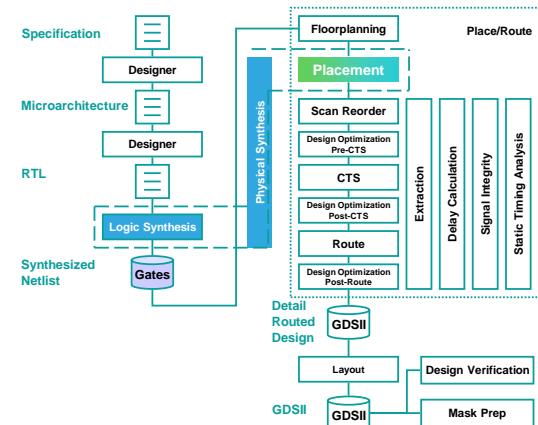
Increased chip operating frequencies make timing closure tougher.

Increasing number of standard cells and macros compressed into ever-decreasing core areas also make timing closure a challenge.

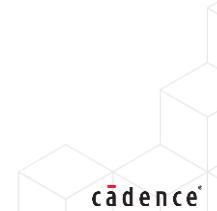
What Is Physical Synthesis?

The combination of logical synthesis and placement.

Example: To meet timing, we ran physical synthesis, which, in addition to upsizing and downsizing components, also ran logic restructuring.



640 © Cadence Design Systems, Inc. All rights reserved.



Traditional synthesis uses vendor-supplied net delay models, which do not provide accurate wire delay information, especially for technology nodes where a significant portion of the delays are contributed by the wires. Consequently, you can see relatively big differences in performance, area, and power between the logic and physical designs.

The difference between synthesis timing and backend timing creates a difficult challenge for timing closure.

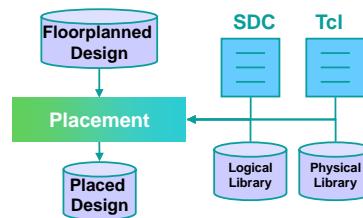
To bridge the gap for better timing closure, synthesis now uses floorplan and placement information during logic synthesis to provide better results that correlate with the backend result.

Hence, this floorplan acts as a bridge to close the implementation gap between logic synthesis and physical synthesis in the place-and-route stage.

So, to meet timing, you run physical synthesis along with other optimizations like upsizing, downsizing, and logic restructuring.

Placement and Physical Synthesis: Input, Output, and File Format

- Input
 - Floorplanned design
 - Constraints in Standard Design Constraints (SDC) format
 - Logical Timing Libraries in Liberty (.lib) format
 - Physical Libraries in LEF format
 - Placement constraints and script in Tcl
- Output
 - Placed design



641 © Cadence Design Systems, Inc. All rights reserved.



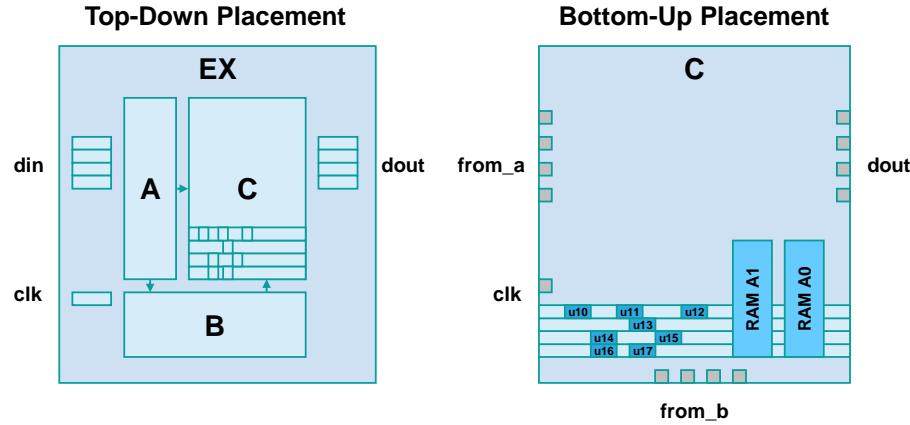
When you run physical synthesis, you need additional inputs as compared to logical synthesis.

In addition to timing libraries in Liberty (.lib) format and constraints in Standard Design Constraints (SDC) format, for physical synthesis, you also need to provide physical libraries in LEF format, floorplan information, placement constraints, and script in Tcl format.

The output of physical synthesis is a placed design.

Example: Placement

- We can run standard cell placement “top-down” or flat at the EX level and place everything at once.
- Or we can place the standard cells for each block separately as a hierarchical design.



642 © Cadence Design Systems, Inc. All rights reserved.



This slide shows two examples of placement. One is implemented using a top-down approach, and the other uses a bottom-up approach.

The first circuit shows a top-down placement, where you can run standard cell placement top-down or flat at the EX level and place everything at once.

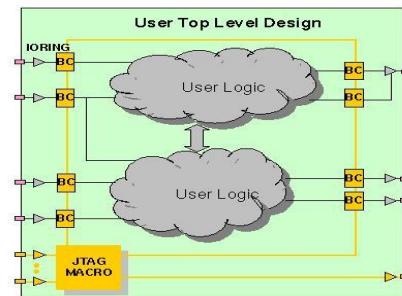
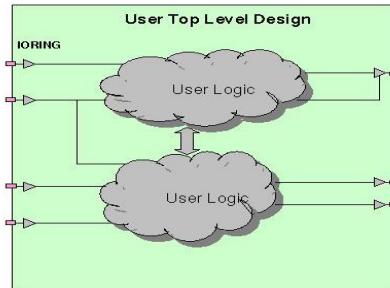
The second placement shows a bottom-up approach of the placement, where you can place the standard cells for each of the blocks separately as a hierarchical design.

What Is Boundary Scan Architecture?

It is a method that enables the chip tester to test connectivity of the I/O pins on the fabricated chip.

- Provides a means to test interconnects between integrated circuits on a board without using physical test probes.
- Is synonymous with Joint Test Action Group (JTAG).

JTAG is the name used for the IEEE 1149.1 standard, *Standard Test Access Port, and Boundary-Scan Architecture to Test Access Ports*.



643 © Cadence Design Systems, Inc. All rights reserved.

cadence®

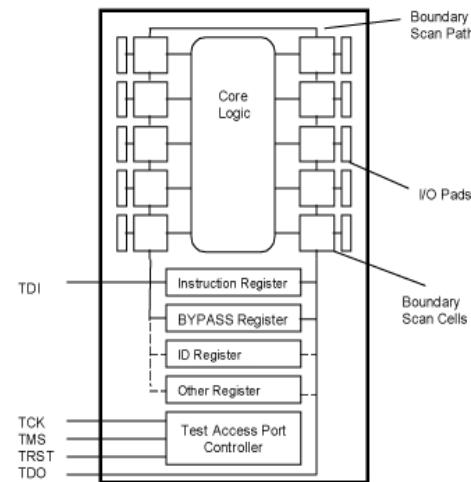
A boundary scan logic is inserted to observe and control the functional ports of a chip, independent of the system logic. Boundary scan cells are inserted between each chip port and the system logic. They are then connected in a boundary scan chain or boundary register.

In boundary scan architecture, the tool inserts boundary scan structures that conform to the JTAG IEEE standard used to observe and control the functional inputs or outputs of a chip using a boundary register.

The JTAG macro is used to observe each chip's input value and control the logic value of each chip output by shifting values in and out of the boundary register.

Boundary Scan Architecture Overview

- Boundary scan adds one or more memory elements, called *boundary-scan cells*, to each I/O pin of the device, which can selectively override the functionality of that pin.
- The collection of boundary scan cells is configured into a parallel-in, parallel-out shift register.
- The test sequence is passed into the shift register, and the data coming out is compared.
- Boundary scan cells do not contribute to the functionality of the internal core logic.
- Test access port (TAP) controller is a state machine whose transitions are controlled by a TMS signal.



644 © Cadence Design Systems, Inc. All rights reserved.



Here we are showing the boundary scan architecture overview.

Every port in the design must have an IO pad instantiated to insert boundary scan cells. The pad pins on the I/O pad cells must be connected to the top-level ports, whereas the input, output, and enable on the I/O pad cells must be connected to the core logic. Note that the boundary scan insertion engine will not connect any of the other disconnected pins on the I/O pad cells.

The boundary scan adds one or more memory elements, called boundary scan cells, to each input and output pin of the device, which can selectively override the functionality of that pin.

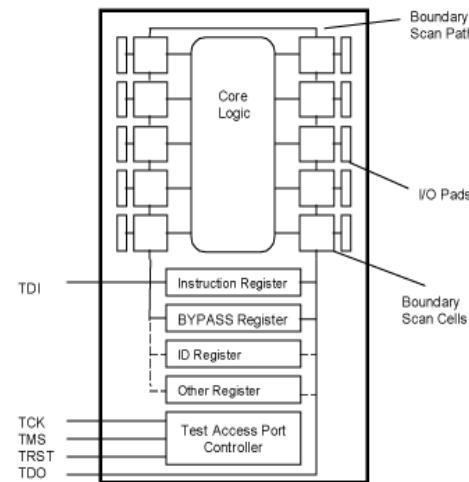
These cells are configured into a parallel-in, parallel-out shift register. So, the test sequence is passed into the shift register, and the data coming out is compared.

During the process, the architecture ensures that boundary scan cells do not contribute to the functionality of the internal core logic.

There is also a Test Access Port (TAP) controller, a state machine whose transitions are controlled by a Test Mode State signal.

Boundary Scan Architecture Overview (continued)

- JTAG interface, collectively known as the *TAP controller*, uses the following signals to support boundary scan operation.
- Test data is shifted around the shift register in serial mode from input pin Test Data In (TDI).
- Test data is terminated at output pin Test Data Out (TDO).
- Test Clock (TCK) synchronizes the internal state machine operation.
- Test Reset (TRST) is an optional input pin to reset the TAP controller's state machine.
- Test Mode State (TMS) determines the next state.



645 © Cadence Design Systems, Inc. All rights reserved.



Let's check out the boundary scan architecture.

You have the core logic, boundary scan cells, which form the boundary scan registers. You have an instruction register, bypass register, ID register, and optionally test data registers, and then a tap controller, which is a test access port controller.

The J-TAG interface, collectively known as the TAP controller, uses various signals to support boundary scan operation.

The component includes certain ports like test clock, test data-out, test data-in, test mode selects, and optionally you have test resets. So, these TCK, TDO, TDI, TMS are the mandatory ports, and you can have a test reset port optionally to reset the TAP controller's state machine.

So, when testing is performed, an instruction is serially loaded into the J-TAG macro through its TDI pin.

Test data is shifted around the shift register in serial mode from input pin Test Data In (TDI).

Then the test data is terminated at the output pin Test Data Out (TDO). The signal Test Mode State (TMS) determines the next state.

Depending on the instruction, the instruction decode logic selects which data is registered to connect between the J-TAG macro, TDI, and TDO pins. The synthesis process will build a default J-TAG macro for the four mandatory instructions that are: the bypass, x-test, preload, and simple. The J-TAG macro can also be created based on the user-defined instructions to control any number of the custom test data registers for other special test structures. If a custom test data register exists in the design and you completely define the user define instructions before inserting the boundary scan logic, the boundary scan insertion engine will connect the J-TAG macro pins and the custom test register pins.

Characteristics of Boundary Scan Methodology

- It is chain integrity testing.
- The basic form of testing is by JTAG (tests that the JTAG devices meant to be in the chain exist).
- Each JTAG-compliant device contains an ID code.
- When a correct sequence of JTAG commands is issued, the ID codes of all the devices can be read out.
- The ID codes read out from the JTAG chain are compared with the actual ID codes of the device. If they match, the JTAG chain is correctly connected, and the devices are in place.
- Benefits of JTAG:
 - Shorter test times
 - Higher test coverage
 - Increased diagnostic capability
 - Lower capital equipment cost

646 © Cadence Design Systems, Inc. All rights reserved.



This slide lists the characteristics of boundary-scan methodology.

It is chain integrity testing, and the basic form of testing is by JTAG. The architecture also tests that the JTAG devices meant to be in the chain exist.

Each JTAG-compliant device contains an ID code. When the correct sequence of JTAG commands is issued, the ID codes of all the devices can be read out.

After this, the ID codes read out from the JTAG chain are compared with the actual ID codes of the device. If they match, the JTAG chain is connected correctly.

There are several benefits to JTAGs. JTAG technology has shorter test times with higher test coverage. It increases diagnostic capability and has lower capital equipment costs.

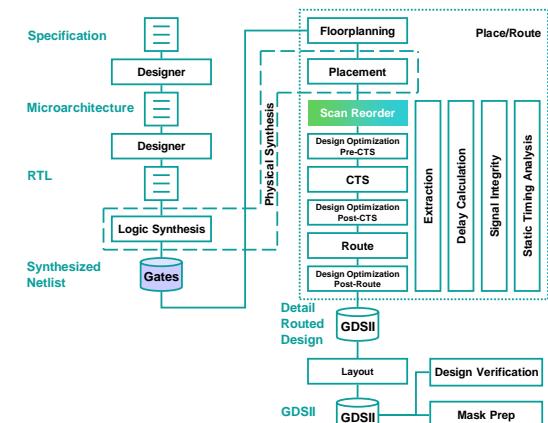
What Is Scan Reorder?

The process of re-connecting the scan chains in a design to optimize for routing, timing, etc.

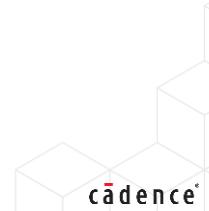
Example: Since logic synthesis arbitrarily connects the scan chain, we need to perform scan reorder after placement so that the scan chain routing will be optimal.

What is a scan chain?

A scan chain is the connection of the flip-flops in a design, such that test patterns can be scanned in, and results scanned out during automated testing.



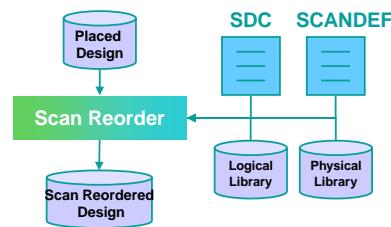
647 © Cadence Design Systems, Inc. All rights reserved.



Scan reorder is a process of reordering or reconnecting the scan chains to optimize routing length and timing. Scan chain is a connection of flip-flops in a design so test patterns can be scanned in and results scanned out during the automated testing. The purpose of adding a scan chain is to make flip-flops in the design controllable and observable. If we are reordering the scan chains, the design will use fewer routing resources compared to the design without scan reordering.

Scan Reorder: Input, Output, and File Format

- Input
 - Placed design
 - Constraints in Standard Design Constraints (SDC) format
 - Logical Timing Libraries in Liberty (.lib) format
 - Physical Libraries in LEF format
 - Scan chain information in SCANDEF format
- Output
 - Scan chain reordered design

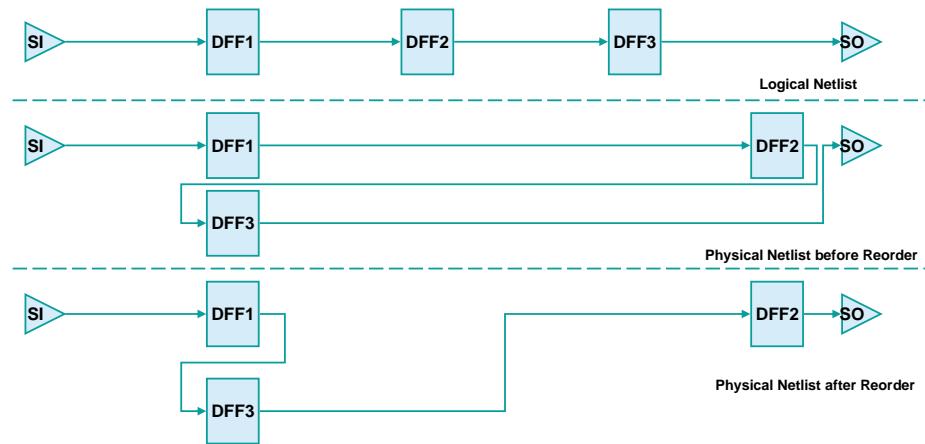


Scan reorder is an intermediate step of placement and clock tree synthesis. The inputs of scan reorder are a placed design, design constraints in SDC format, timing library in Liberty format, physical library in a LEF format, and a Scan def file with the scan chain information. The output would be the final scan chain reordered design.

Example: Scan Reorder

Scan chains that were stitched in the logical netlist need to be reordered now that placement is done.

- Logical netlist was stitched numerically.
- Physical netlist is reordered based on placement.



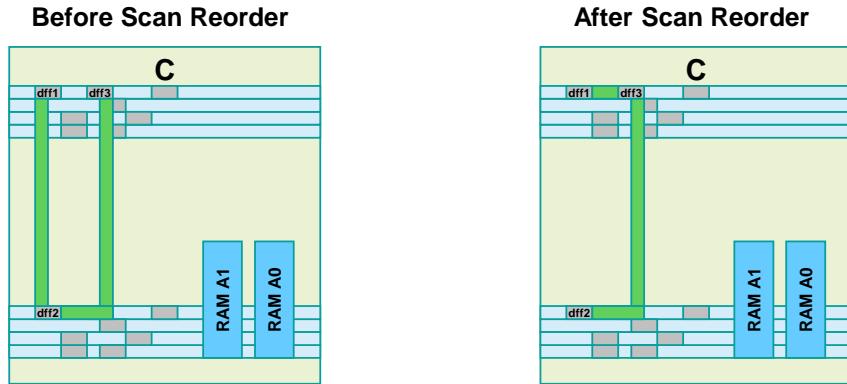
649 © Cadence Design Systems, Inc. All rights reserved.



This slide shows the importance of scan chain reordering; the flops in the original logical netlist are not placed. After placing the design, the wiring between the flops takes into consideration the actual placement of the cells. To improve the routing resources, reordering the scan chain connectivity results in fewer routing resources used, as shown in the picture.

Example: Scan Reorder (continued)

Reordered scan chain requires much fewer routing resources in the example design.



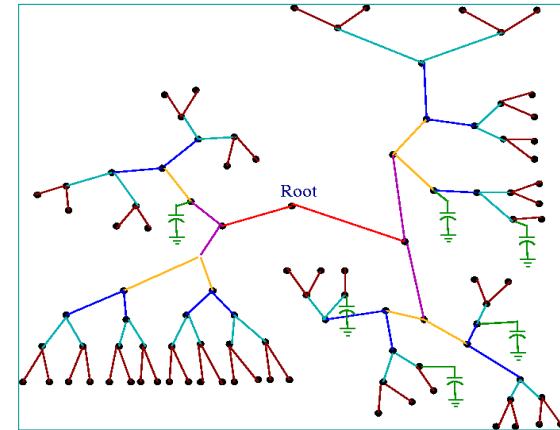
650 © Cadence Design Systems, Inc. All rights reserved.



Here is the same example shown for scan reorder in the layout view. We can observe here that before the scan reorder, more routing resources are used when compared to the layout after the scan reorder. After the scan reorder, we see the flip-flops have been reordered, which results in minimized net length, and it also results in fewer routing resources being used.

What Is a Clock Tree?

- A network of buffers inserted into the clock signal path in such a way that the overall delay from the generator to all destinations is minimized.
- In a synchronous digital system, a clock signal is used to define a time reference for the movement of data within that system.



Example:

Instead of one electrical signal path being optimized, the path in the design was broken up and strategically buffered to minimize the delay.

The resulting network resembled a tree in that the central clock signal branches throughout the chip using these buffers and ends up with the clock signal reaching all of the leaf cells.

651 © Cadence Design Systems, Inc. All rights reserved.



In a synchronous digital system, a clock signal is used to define a time reference for the movement of data within that system. A clock tree can be defined as a network of buffers inserted into the clock signal path in such a way that the overall delay from the generator to all destinations is minimized.

Need for a Clock Tree

- When complexity (the number of gates in a design) increases, the need to distribute clock signals in a controlled manner becomes more important.
- Reasons why we need to build a clock tree:
 - Large chip area
 - Different flop densities
 - Non-uniform distribution of flops
 - All flops need to get a clock signal at the same time
 - Power budget
 - Clock routing: hard problem
- The *clock distribution network* distributes the clock signal(s) from a common point to all the elements that need it.

652 © Cadence Design Systems, Inc. All rights reserved.



A large, pipelined chip may easily contain hundreds of clocked elements (latches, flip-flops, etc.), so a lot of buffering is required to meet clock constraints as they relate to skew and insertion delays.

Since it is usually desirable to have the clock pins switch at the same time, care has to be taken when designing the clock distribution.

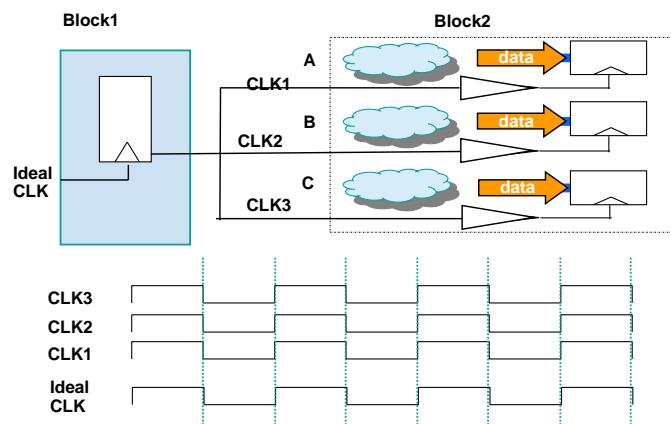
A clock signal is vital to the operation of a synchronous system.

Clock signals are often regarded as simple control signals. However, these signals have some very special characteristics and attributes.

What Is an Ideal Clock?

All flip-flops are clocked together.

- Simplifies clock analysis over hierarchical boundaries.
- Used before clock tree insertion and place-and-route for timing analysis.



Note: This diagram assumes zero clock skew and insertion delay.

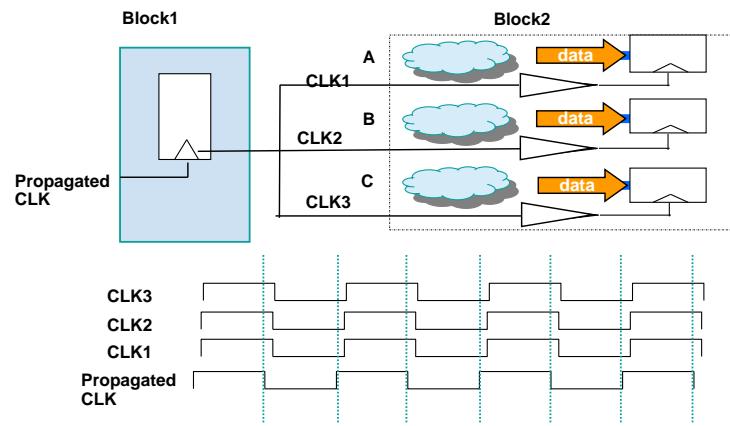
An ideal clock models a clock in its ideal state, where the clock signal of a certain period and duty cycle arrives at the clock pins of flops, macros, and soft blocks at the same time. An ideal clock simplifies the timing analysis, especially for clock pins of hierarchical blocks.

A clock is in an ideal state before clock tree synthesis. Clock tree synthesis is the stage when an actual clock tree structure is inserted into the physical design. In the waveforms that are shown, clocks 1, 2, and 3 mirror the ideal clock since the clocks are ideal.

What Is a Propagated Clock?

Clock delays extracted from clock tree routing.

- Clock skew is correctly modeled using propagated delay.
- More accurate and used in the final timing closure.



654 © Cadence Design Systems, Inc. All rights reserved.

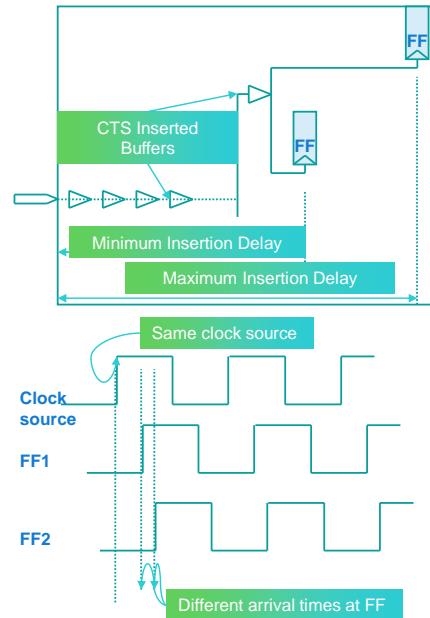


A clock is described as propagated after clock tree synthesis, where the actual clock's timing characteristics can be extracted. When clock buffers are inserted into the design, the static timing analysis tools replace the modeled clock information, previously the ideal clock, with the actual clock information. A propagated clock includes the actual skews and insertion delays of the inserted clock tree.

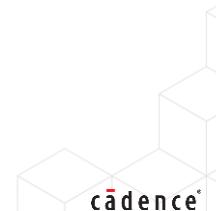
What Is Clock Skew?

The measure of the difference of delay between the minimum and maximum time it takes the clock to reach different leaf cells.

- Typically hurts design performance, although in some cases, helps achieve timing targets (useful skew).
- Caused by a clock tree with unbalanced branches occurring due to:
 - Different types of buffers.
 - Varying capacitance and resistance values of nets.
 - Gating components.
 - Off-chip or on-chip variations.



655 © Cadence Design Systems, Inc. All rights reserved.



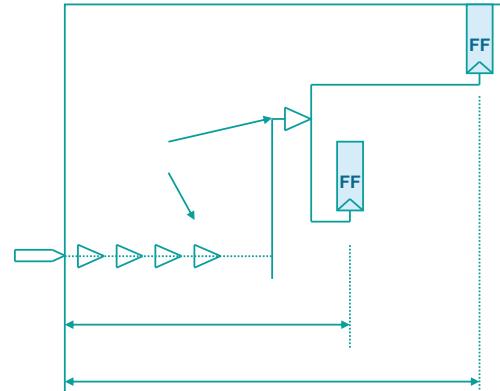
Clock skew is the difference in delay between the minimum and maximum time it takes the clock signal to reach the clock pins of leaf cells. Clock skew can hurt design performance, although in some cases, clock skew can help achieve timing targets. This helpful type of skew is called useful skew.

Clock skews are caused by timing differences within clock buffers, varying capacitance and resistance values of nets, clock gating components, and off-chip or on-chip variations.

What Is Insertion Delay?

Insertion delay is the time the clock signal (rise or fall) takes to propagate from the clock definition point (root) to a register clock pin (leaf cells).

Insertion delay is also known as a **clock network latency**.



656 © Cadence Design Systems, Inc. All rights reserved.

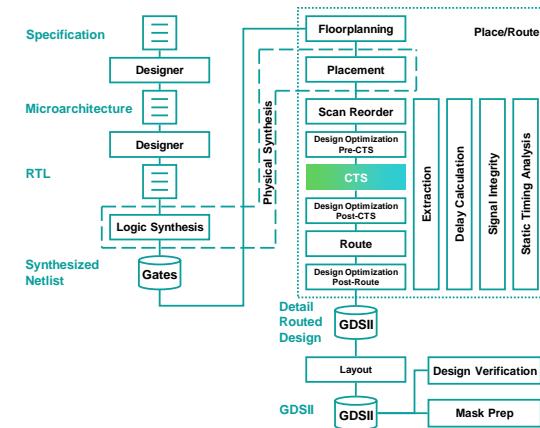


Insertion delay is the time taken by a clock signal to propagate from the clock definition point to a register clock pin. Insertion delay is also known as clock network latency.

What Is Clock Tree Synthesis (CTS)?

The process of inserting buffers in the clock path, with the goal of minimizing clock skew and latency to optimize timing.

Example: We ran clock tree synthesis on the example block and saw a large clock skew due to bad clock constraints. We re-ran clock tree synthesis with better constraints to get an optimal result.



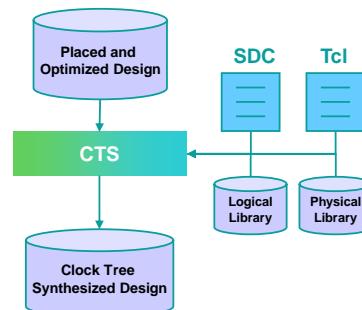
657 © Cadence Design Systems, Inc. All rights reserved.



Clock tree synthesis or CTS is the step after placement. CTS is a process of inserting buffers in the clock path, to minimize clock skew and latency resulting in optimized timing. To ensure that the chip will work correctly at the required clock frequency, a clock tree needs to be inserted to synchronize memory elements such as RAMs and flops.

Clock Tree Synthesis: Input, Output, and File Format

- Input
 - Placed and Optimized design
 - Constraints in Standard Design Constraints (SDC) format
 - Logical Timing Libraries in Liberty (.lib) format
 - Physical Libraries in LEF format
 - Clock constraints and commands in Tcl
- Output
 - Post-CTS design with clock trees inserted



658 © Cadence Design Systems, Inc. All rights reserved.



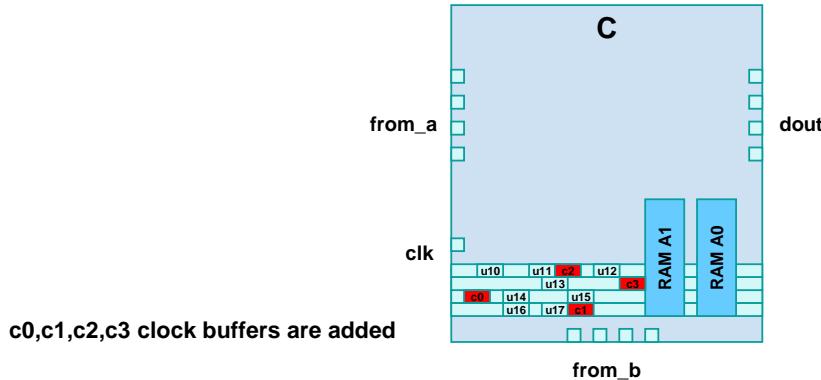
The input to CTS is a placed and optimized design. Other inputs include constraints in SDC format, timing libraries in Liberty format, also known as .lib files, physical libraries in LEF format, clock constraints, and Tcl commands.

The result of CTS is a design with clock trees inserted.

Example: Clock Tree Synthesis

Up to now, the clocks in the design have been treated as ideal (no clock skew, no clock latency, ideal transition time, etc.). In CTS, we add buffers for the real clock tree in order to minimize:

- Clock skew in the design
- Clock latency in the design



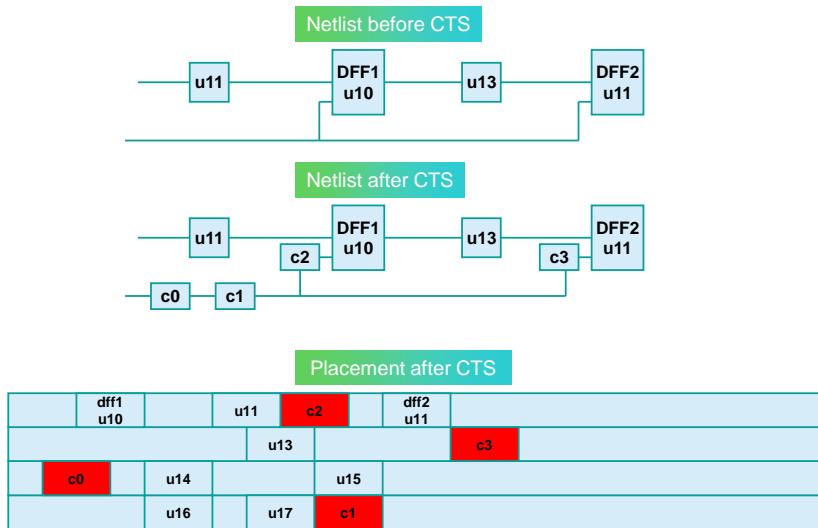
659 © Cadence Design Systems, Inc. All rights reserved.



Before CTS, the clocks in the design have been treated as ideal clocks without taking into account clock skews and clock latencies and with ideal transition times. During CTS, we add buffers for the real clock tree to minimize clock skew and clock latency in the design. In the example that is shown, clock buffers have been added to the physical design to help meet clock constraints.

Example: Clock Tree Synthesis (continued)

Buffers are added to the clock tree in our example design.



660 © Cadence Design Systems, Inc. All rights reserved.



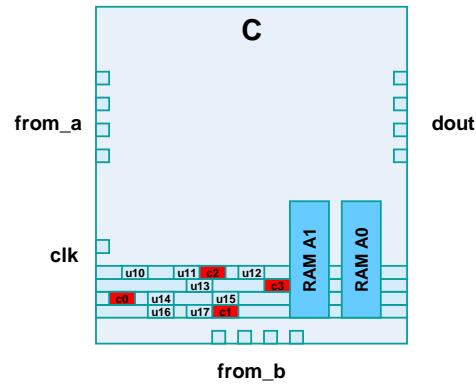
This example illustrates the changes in the netlist as a result of clock tree synthesis. After a clock tree has been inserted, buffers c0, c1, c2, and c3 have been added to the netlist. Also, notice that after clock tree synthesis, clock buffers have been placed in their optimum locations in the layout. Both the logical netlist and the layout will have been modified as a result of CTS.

Example: Design Optimization, Post-CTS

Another round of design optimization takes place because it is possible that CTS could have disturbed the timing of some of the paths in the design.

Post-CTS optimization can include:

- Buffering
- Upsizing or downsizing cells
- Modifications to the clock tree itself



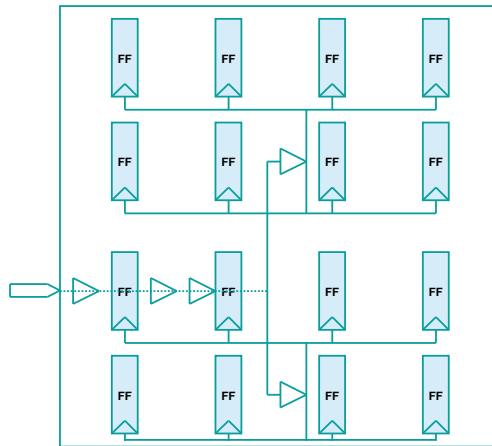
661 © Cadence Design Systems, Inc. All rights reserved.



Another round of design optimization takes place after CTS, called post-CTS optimization. This step optimizes the timing of the paths after clock tree insertion. Post-CTS optimization includes buffering, upsizing, or downsizing cells or modifications to the clock tree.

What Happens After CTS?

- Clock buffer tree is built to balance output loads and minimize clock skew.
- Buffers can be added to the network to meet the minimum insertion delay.



662 © Cadence Design Systems, Inc. All rights reserved.



After the clock has been built, the resulting tree will balance output loads and minimize the clock skew. Buffers are also added to the clock network to meet the minimum insertion delay.

Goals of CTS

- Deliver clock to all memory elements with:
 - Acceptable skew.
 - Least amount of insertion delay.
- Deliver clock edges with acceptable sharpness.
- Minimize the power consumption.
 - Clock network switches at every clock cycle, thus consumes a lot of power.



The main goal of clock tree synthesis is to deliver the clock signal to all the memory elements with acceptable skew, the least amount of insertion delay, and acceptable clock transition times while minimizing power consumption.

When the clock network switches at every clock cycle, it can consume a significant amount of power.

What Is Signal Integrity?

Unintended effects on digital signals caused by interconnect parasitic resistance or capacitance that causes noise and/or changes delays.

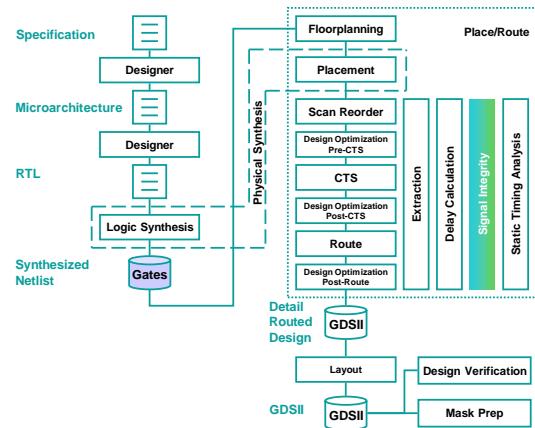
Example: In our example design, we saw signal integrity (SI) effects such as noise-on-delay and glitches due to long nets that were running in parallel.

What is noise-on-delay?

Crosstalk-induced delay or incremental delay due to coupling capacitance.

What is a glitch?

A glitch is a bump or change in value caused by a changing signal affecting a neighboring wire.



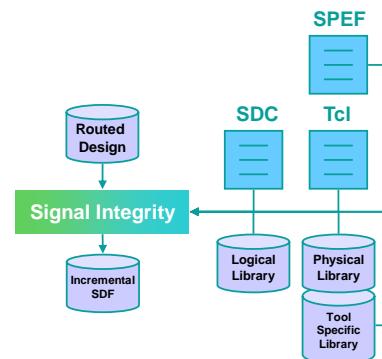
664 © Cadence Design Systems, Inc. All rights reserved.



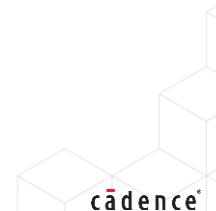
Signal Integrity (SI) is the unintended effects on digital signals caused by interconnect parasitic resistance or capacitance that impacts the delay or introduces a glitch on a given net due to the switching of nets in its proximity.

Signal Integrity: Input, Output, and File Format

- Input
 - Routed design
 - Constraints in Standard Design Constraints (SDC) format
 - Constraints and commands in Tcl
 - Parasitic extraction file (SPEF)
 - Logical Timing Libraries in Liberty (.lib) format
 - Physical Libraries in LEF format
 - Power rail IR drop data
 - Tool-specific SI libraries
- Output
 - Incremental Standard Delay Format (SDF) file containing all of the delay information in the design related to noise-on-delay
 - Reports for glitch nets
 - List of problem nets that need to be re-routed



665 © Cadence Design Systems, Inc. All rights reserved.



The inputs to signal integrity analysis are a routed design, SDC constraints, SPEF files, timing libraries or the .libs, physical LEF files, power and IR drop data, SI libraries, and cross-coupling capacitance files. The outputs are an SDF file, glitch reports, and ECO files containing a list of problem nets that need to be rerouted.

Example: Signal Integrity

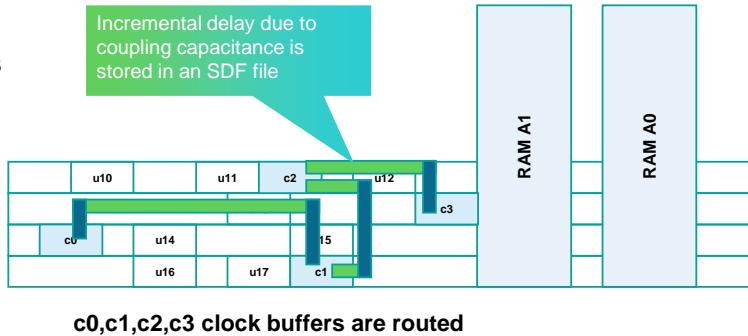
We can run checks and produce data or reports to help us identify timing and reliability issues due to SI. For submicron designs, closely coupled nets can produce:

- Crosstalk-induced delay
- Noise

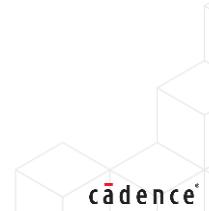
Power rail IR drop can cause:

- Weakened drivers
- Increased delays
- Lower noise margins

Incremental delay due to coupling capacitance is stored in an SDF file



666 © Cadence Design Systems, Inc. All rights reserved.

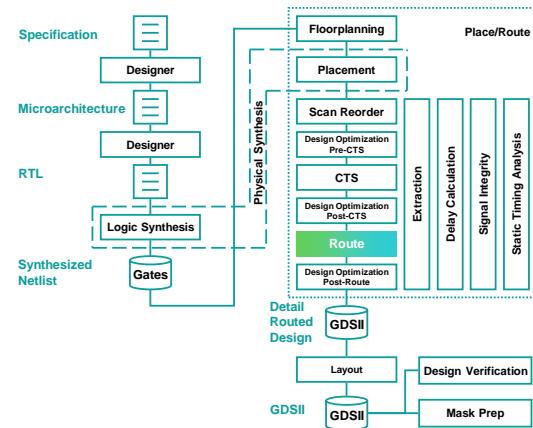


SI issues can lead to timing and reliability issues. SI problems can increase or decrease signal delay, which can, in turn, cause setup or hold failures. IR drop issues can lead to lowered drive, increased delays, and lower noise margins. Therefore, robust signal integrity checks are required to help us identify these issues for faster design convergence.

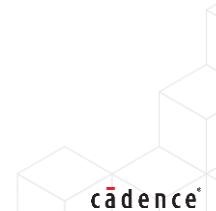
What Is Route?

The process of connecting the pins of the standard cells, macros, and I/Os of a digital design to specific metal layers in the process technology to match the schematic.

Example: We ran a preliminary route on the example block and saw that routing congestion was an issue. To fix it, we re-ran placement with a placement density screen to force a lower utilization in that area and allow for more routing resources.



667 © Cadence Design Systems, Inc. All rights reserved.

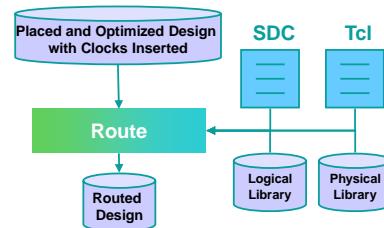


Routing is a process of connecting the pins of the standard cells, macros, and Input-Output ports of a digital design. Different metal layers are used from the specific process technology to make the connections in the design and match the schematic.

If there is any congestion for the preliminary route of the design, we must re-run the placement with a placement density screen to force lower utilization in that area and allow more routing resources.

Route: Input, Output, and File Format

- Input
 - Placed and optimized design with clock tree inserted
 - Constraints in Standard Design Constraints (SDC) format
 - Logical Timing Libraries in Liberty (.lib) format
 - Physical Libraries in LEF format
 - Route constraints and commands in Tcl
- Output
 - Routed design



668 © Cadence Design Systems, Inc. All rights reserved.



Let's check out different inputs and outputs for routing.

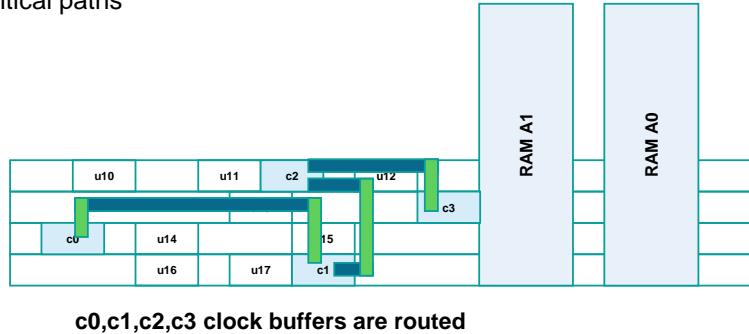
The input to the routing step is a placed and optimized design after the clock tree synthesis, constraints in SDC format, timing libraries in the .lib format, and physical libraries in .LEF file format, routing constraint, and commands in Tcl file. The output of routing is a detail routed design.

Example: Route

When the design has been fully placed with all the clock tree buffers, it is time to perform routing. Routing connects all the I/Os, standard cells, RAMs, and macros to their specific routing layers according to the synthesized netlist.

The router will try to minimize:

- Route congestion
- Timing impact on critical paths



669 © Cadence Design Systems, Inc. All rights reserved.



Let's discuss an example of routing when the design has been fully placed, and including the clock tree buffers or inverters. The next step is to make the connections or routes using specific metal layers for all clock tree buffers. Routing creates connections for all the standard cells, RAMs, Input-Output ports, and macros according to the connectivity defined in the synthesized netlist.

This image shows the routing of all the clock tree buffers.

Goals of Routing

- Responsible for functionally connecting all signal nets, power nets, and buses in a design.
- Route the design quickly and be free from design rule check (DRC), layout versus schematic (LVS), and signal integrity (SI) errors.
- Effectively meet design for manufacturability and overall timing specifications.



There are several sets of goals for routing. Goal one, make connections for all the signal nets, power nets, and buses according to the functionality. Goal two, route the design quickly while avoiding Design Rule Checks (DRC), Layout Versus Schematic (LVS), and Signal Integrity errors. Goal three is effectively meeting the design requirements for manufacturability and overall timing specifications.

Steps Involved in Routing

1. Global routing

- Assigns nets to specific metal layers and global routing cells.
- Tries to avoid congested global cells while minimizing detours.
- Tries to avoid pre-routed power and ground signal, placement, and routing blockages.

2. Track assignment

- Assigns each net to a specific track.
- Tries to avoid a large number of vias.
- Operates on the entire design at once.

3. Detail routing

- Tries to fix DRC violations using a fixed-size, small area known as Sbox.
- Traverses the whole design box by box until the entire routing pass is complete.

4. Search and repair

- Fixes any shorts or violations that are present.

671 © Cadence Design Systems, Inc. All rights reserved.



This slide explains the four major steps involved in the routing stage. They are global routing, track assignment, detail routing, and search and repair. Let's take a look at each step.

Global routing is the process of assigning nets to specific metal layers and global cells called gcells. It avoids congested cells while minimizing the detours, and it avoids pre-routed power and ground signals, as well as placement and routing blockages.

Track assignment is completed once for the entire design. It is the process of assigning each net to a specific routing track in the design.

Global routing also avoids placing a larger number of vias in the design.

Detail routing is the process of routing the entire design while meeting timing and signal integrity requirements and, at the same time, minimizing the design rule violations.

And the final step is search and repair, which is the process of fixing the remaining design rule violations, such as spacing violations, shorted nets, and open nets.

What Is Global Routing?

Global routing guides the detailed router in large designs.

- Creates a coarse routing plan for detailed router to follow.
- Does not create actual routing wires.
- May perform quick, initial detail routing.
- Commonly used in cell-based design, chip assembly, and datapath.
- Also used in floorplanning and placement.



Global routing guides the detail router by creating a route plan for the detail router to follow. The global router may perform quick, initial routing, especially during floorplanning and placement. These quick routes are used for validating the floorplan and for running timing analysis at the initial stages of implementation.

The global router does not create final routes that become a permanent part of the design.

Global Routing Goals

- Minimize the wire length
 - Total wire length calculated by the global router should be within a few percentage points of that estimated by the placer.
- Minimize worst congestion value
 - Congestion value is associated with each boundary crossing (edge) between adjacent *global routing cells* (gcells) on a specific layer.
- Optimize routes for timing and signal integrity
 - Tries to meet hold and set up timing.
 - Minimizes design rule violations.

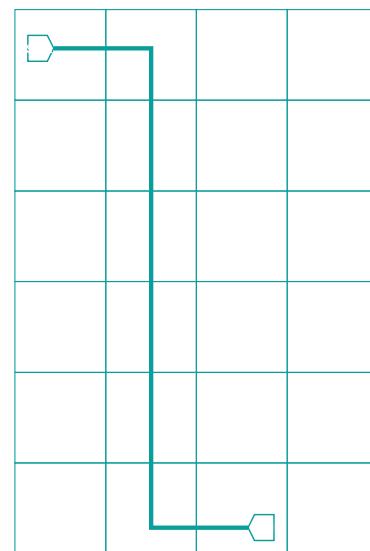


The global router creates a route plan that minimizes the wire length and congestion.

Route congestion is associated with each boundary crossing or edge between adjacent global routing cells, also known as gcells, on a specific layer. The global router optimizes routes for timing and signal integrity while trying to meet setup and hold timing. The global router creates a route plan that strives to minimize design rule violations in the created routes.

Global Routing Steps

1. Router breaks the routing portion of the design into rectangles gcells.
2. Router then assigns the signal nets to the gcells.
3. Router attempts to find the shortest path through the gcells.
 - No actual connections are made.
 - No nets are assigned to specific tracks within the gcells.

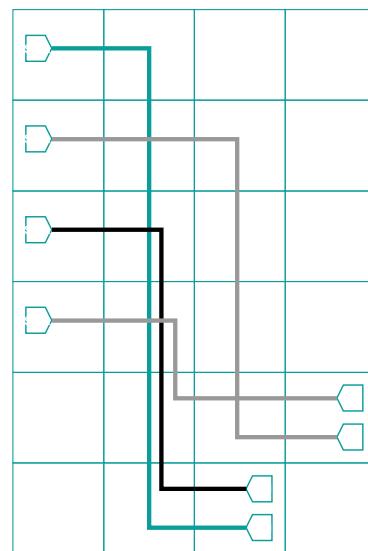


674 © Cadence Design Systems, Inc. All rights reserved.

The global router breaks the routing portion of the design into rectangles called gcells. Next, the router assigns the signal nets to the gcells. The router attempts to find the shortest path through the gcells. At this point, no actual connections are made, and nets are not assigned to specific tracks within the gcells

Global Routing Steps (continued)

4. Tries to avoid assigning more nets to a gcell than the tracks can accommodate.



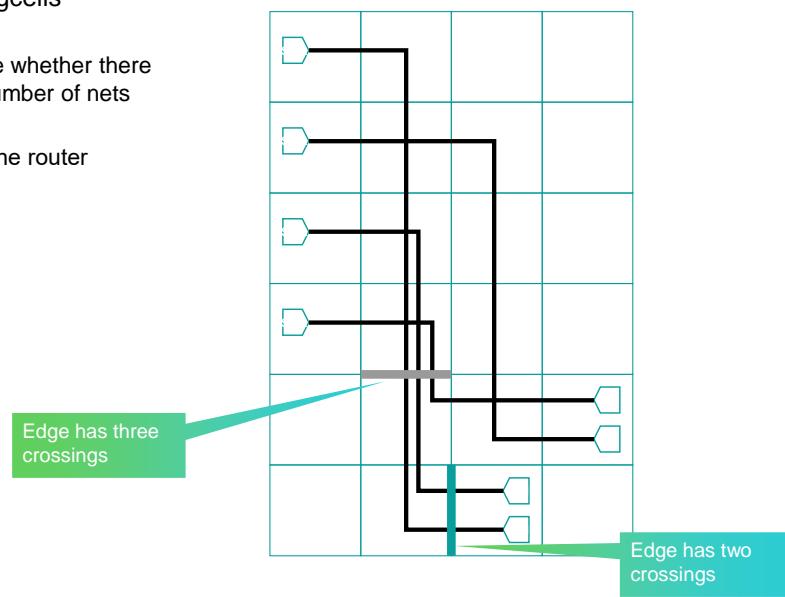
675 © Cadence Design Systems, Inc. All rights reserved.



This diagram illustrates several nets that traverse multiple gcells. To prevent downstream congestion, the global router will avoid assigning too many nets to a given gcell.

Global Routing Steps (continued)

5. Router then generates a map of the gcells (congestion map).
 - Congestion map uses colors to indicate whether there are too few, too many, or the correct number of nets assigned to the gcells.
 - gcells are marked “over-congested” if the router assigns too many nets to a gcell.



676 © Cadence Design Systems, Inc. All rights reserved.

cadence

A part of global routing is to generate a map of the gcells called a congestion map.

The congestion map uses colors to indicate whether there are too few, too many, or a reasonable number of nets assigned to the gcells. gcells are marked “over-congested” if the global router assigns too many nets to a gcell. Evaluating the congestion map and fixing congestion issues will help converge timing and signal integrity while minimizing potential DRV violations during the detail route.

What Is Detail Routing?

Connects all pins in each net.

- Must understand most or all design rules.
- Necessary in all applications.
- The goal is to complete all of the required interconnects without violations.
- All nets will be routed, even if they contain violations (better to have a route with a violation than no route at all).



Using the route plan created by the global route, detail routing will connect the pins of all the signal nets using metal layers. The detail router will create routes based on an understanding of the design rules coded in the technology file. The goal for the router is to complete all of the required interconnects without violations. All nets will be routed, even if the routes contain violations because the understanding is that it is better to have a route with an unavoidable violation than no route at all, which is called an open.

Detail Routing Steps

1. Router divides the chip into areas called switch boxes (Sboxes).
 - Sboxes align with gcell boundaries.
2. Router follows global routing plan.
 - Lays down actual wires that connect the pins to the corresponding nets.
 - Creates shorts or spacing violations rather than leaving unconnected nets.
3. Router runs search and repair.
 - Locates the shorts and spacing violations.
 - Re-routes affected areas to eliminate as many violations as possible.
4. Runs post-route optimization.
 - Runs rigorous search-and-repair steps.
5. Stops once it cannot make further progress on routing the design.

678 © Cadence Design Systems, Inc. All rights reserved.



The detail router divides the chip into areas called switch boxes or S-boxes.

These S-boxes align with gcell boundaries.

The router follows the plan that was created by the global router.

The detail router creates actual wires that connect the pins to the corresponding nets.

The router will create shorts or spacing violations rather than leave unconnected nets. This is because a short or a DRC violation is a localized problem that can be solved during search and repair, whereas an unconnected net is a more difficult problem to solve later in the routing flow. The router runs search and repair as the next step. During search and repair, the router locates the shorts and spacing violations and re-routes affected nets to eliminate as many violations as possible. If there are post-route optimization steps required, there will be additional search and repair loops to fix the remaining DRC violations. The router will stop routing once it cannot make further progress.

Timing-Driven Routing

- Routing along the timing-critical path is given priority.
- Creates shorter and faster connections along the critical path.
- Non-critical paths are routed around critical paths.
 - Reduces routing congestion problems for critical paths.
 - Does not adversely impact the timing of non-critical paths.
- Input files needed for timing-driven routing:
 - Physical libraries in LEF.
 - Timing library in .lib format.
 - Timing constraints in .sdc format or a timing graph.
 - Extended capacitance table.
 - Verilog netlist.
 - Placed design in DEF.



When routing is run in timing-driven mode, the routes along the timing-critical path are given priority. The router creates shorter connections, which will result in faster timing along the critical path. Non-critical paths are routed around critical paths. The objective is to reduce routing congestion problems for critical paths. This strategy does not adversely impact the timing of non-critical paths.

The input files needed for timing-driven routing are LEF files, a timing library in .lib format, timing constraints in SDC format, a timing graph, a capacitance table or similar technology files for extraction, Verilog netlists, and a placed design.

Congestion-Driven Routing

- Router tries to reduce congestion.
 - Routing occurs based on a cost function.
 - Congestion reduction is given the highest priority.
- Nets that are in the congested area are spread apart and routed through other areas.



During congestion-driven routing, the detail router tries to reduce the congestion of the routes by spreading the nets farther apart. For congestion-driven routing, the net length is not as important a consideration as reducing congestion. The router balances the cost of various variables, and in this case, congestion reduction is given the highest priority.

Nets in the congested area are spread apart and routed through other areas.

SI-Aware Routing

- Crosstalk effects, such as glitches and delay, are measured after the physical wires are made available.
- Router tries to reduce crosstalk between wires.
- Creates routes with reduced coupling capacitances by:
 - **Parallel wire minimization:** Limiting the distance that two wires travel adjacent to each other.
 - **Layer switching:** Changing the track assignment for a wire so that potential victim nets can be moved away from a strongly driven signal net.
 - **Net shielding:** Using power and ground lines to shield critical high-speed signals such as clocks.
 - **Track reassignment:** Assign tracks to parallel wires that are further apart, with in-between tracks assigned to shorter, less noise-sensitive wires.
 - **Soft spacing:** Making use of available free space to spread wire segments apart.



Crosstalk effects such as glitch and delay are measured after the physical wires have been routed.

During routing, the router tries to reduce crosstalk between wires which adversely impacts the timing and even the functionality.

The router creates routes with reduced coupling capacitances by using several techniques.

One technique is to minimize the occurrence of long parallel wires.

Another technique is to switch layers or reassign the layer tracks for a wire so that potential victim nets can be moved away from a strongly driven or aggressor signal net.

A technique called net shielding uses power and ground lines to shield critical high-speed signals such as clocks.

The track reassignment technique can be used to assign tracks to parallel wires so that they are further apart, with in-between tracks assigned to shorter, less noise-sensitive wires.

Soft spacing, which is a technique to make use of available free space to spread wire segments apart, is also used for mitigating SI effects.

Wide Wire Routing

- Routing is done with wider wires for post-route yield optimization.
- Router widens wires where resources are available.
 - Does not add DRC.
 - Does not add antenna violations.
 - Does not affect timing.
- Wire widening uses non-default rules.

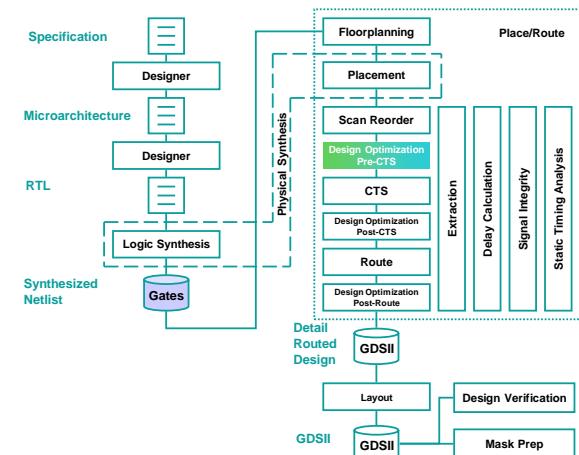


Wide wire routing is a technique that uses wider wires than the minimum width for post-route yield optimization. The router widens wires where resources are available. The wire widening will not add DRC violations. The wire widening will not add antenna violations. The wire widening will not adversely affect timing. Default rules are specified in the technology LEF file. However, wire widening uses non-default rules that need to be defined.

What Is Design Optimization?

Process of using automated algorithms to improve the quality of a digital design.

Example: After initial placement, we run a pass of pre-CTS design optimization to fix timing violations that may show up now that the design is placed, and we have delays based on estimated interconnect.



683 © Cadence Design Systems, Inc. All rights reserved.



Design optimization is the process of using automated algorithms to improve the quality of a digital design in terms of its power, performance, and area. Optimization occurs at various stages of the implementation flow: pre-CTS, post-CTS, and post-route. Something you might see is that after initial placement, pre-CTS optimization was also run on the design to improve the power, performance, and area before clock tree synthesis. During placement, a quick route is also run so that the timing can be derived.

What Is Design Optimization?

(continued)

- It's (almost) guaranteed that the design will not meet the timing requirements on the first run.
- Optimization is the process of iterating through a design such that it meets timing, area, and power specifications.
- In general, optimization can be broken down into the following areas: Power, Performance, and Area (PPA).



It is often the case that the design will not meet the timing requirements on the first run.

If the timing is met, you may still want to optimize the design for power or area. Optimization is the process of iterating through a design such that it meets timing, area, and power specifications.

In general, optimization can be broken down into the following areas: Power, Performance, and Area (PPA).

Optimizing for Timing

There are many ways to reduce delay; we will cover some fundamental techniques here.

- Upsizing gates increases their drive strength and, thus, reduces the time it takes for that gate to transition based on a given load.
 - Upsizing a gate decreases its output resistance (reducing R).
 - Upsizing a gate increases its own input capacitance, giving its driver a higher capacitive load.
 - A technique called *logical effort* was invented to optimize the size of gates along a path for a minimal delay.
 - The tool will usually perform calculations for you.
- Reduce wire capacitance (reducing C)
 - Usually involves shortening the wire lengths of critical paths by moving cells or inserting buffers.
 - Switching to a higher metal layer can also reduce capacitance.



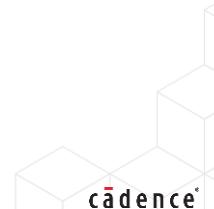
Optimization for setup time is the process of reducing delays. Techniques include upsizing gates to increase their drive strength, which reduces the time it takes for that gate to transition based on its load. Upsizing a gate decreases its output resistance and increases its input capacitance.

A technique called logical effort was invented to optimize the size of gates along a path for a minimal delay. During optimization, the tool will reduce wire capacitance.

This usually involves shortening the wire lengths of critical paths by moving cells or inserting buffers. Switching the route to a higher metal layer can also reduce capacitance because higher metal layers have wider minimum width definitions, which reduces resistance.

Optimizing for Timing (continued)

- Often, your design will contain an adder or multiplier unit in the logic path.
 - For a large number of bits, for example, a carry-lookahead adder performs much better than a ripple carry adder.
 - Physical synthesis tools optimize datapath elements to meet timing while balancing area and power.
- If all fails and the datapath contains too much combinational delay, it is often viable to simply break the path and insert a register in between, creating an extra pipeline stage.
 - An extra pipeline stage means more latency and more area.
 - Such a change usually requires changing the RTL itself.



During timing optimization, if the optimization occurs at an early stage of the implementation process, physical synthesis can help in timing convergence.

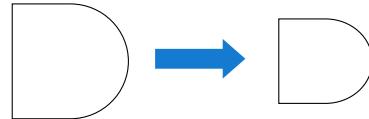
If your design contains an adder or multiplier unit in the logic path for a large number of bits, a carry lookahead adder performs much better than a ripple carry adder.

Physical synthesis tools optimize datapath elements to meet timing while balancing area and power.

If the datapath contains too much combinational delay, it is often viable to simply break the path and insert a register in between, creating an extra pipeline stage. An extra pipeline stage means more latency and more area. Such a change usually requires changing the RTL.

Optimizing for Power

- To reduce power.
 - Reduce capacitance.
 - Decrease the size of standard cells. Power is also a linear function of the driving current, and smaller gates output less current.



One way to reduce power consumption is to reduce the capacitance by downsizing cells without degrading timing to the point where timing is violated. Smaller cells contain less capacitance and therefore dissipate less power by outputting less current when compared to larger cells.

Power and Timing Tradeoff

- As we were discussing power optimization, you may have noticed that some of the techniques are in direct conflict with those in timing optimization.
 - For example, downsizing gates leads to less power but also more delay.
- This is an age-old problem in the development of ICs, and there is no one correct solution.
- Every chip has its own priorities regarding power or delay.
 - For example, for a mobile phone processor, battery life and power consumption may be more important than performance.

688 © Cadence Design Systems, Inc. All rights reserved.



The techniques used for power optimization are sometimes in direct conflict with those used for timing optimization.

For example, downsizing gates leads to less power but also leads to more delay.

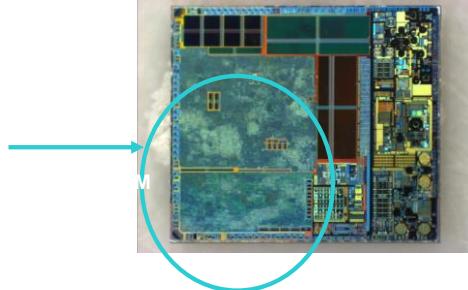
There is often not a perfect solution, but you can set tradeoffs based on the application of the chip.

Every chip has its priorities regarding power or delay.

For example, for a mobile phone processor, battery life and power consumption may be more important than performance. For a GPU, the reverse may be true.

Optimizing for Area

- The purpose of shrinking device dimensions to 5nm is to fit more transistors on a die, giving the chip more functionality for the same area.
- Area is, therefore, a very important specification, especially for chips used for medical purposes such as hearing aids and pacemakers.
- The components that usually take up the most area on a chip are RAMs and register files.
- Shrinking the size of RAMs is an architectural issue and must be settled with the RTL designer.



689 © Cadence Design Systems, Inc. All rights reserved.



The purpose of shrinking device dimensions to 5nm and below is to fit more transistors on a die, thus packing more functionality into the chip for the same area.

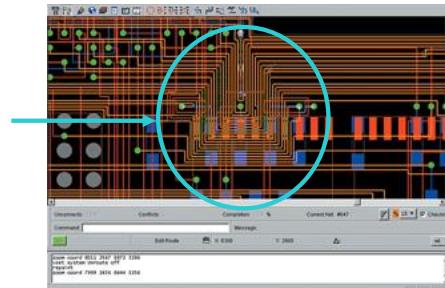
The area constraint is an important specification, especially for chips used for medical purposes, such as hearing aids and pacemakers. Conversely, the area constraint may not be critical for server applications.

The components that usually take up the most area on a chip are RAMs and register files.

Shrinking the size of RAMs is an architectural issue and must be discussed with the RTL designer.

Optimizing for Area (continued)

- Downsizing gates also have a small effect but come at the cost of reduced speed and signal integrity.
- Utilization is defined as how much percentage of the floorplan area has been taken up.
- If the utilization is too high, the design may become congested, making it difficult to route. Longer routes also make it harder to meet timing.



690 © Cadence Design Systems, Inc. All rights reserved.



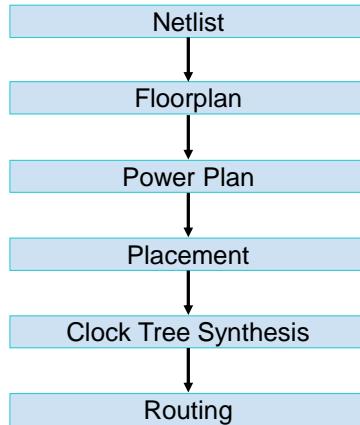
Downsizing gates has a positive impact on power consumption and area but could be at the cost of reduced speed and signal integrity.

Utilization is defined as how much percentage of the floorplan area has been taken up by the standard cells and blocks in the design.

If the utilization is too high, the design may become congested, making it difficult to route. Longer routes also hurt timing.

Optimization During the Design Flow

- Now that you have learned all of these optimization techniques, where do you use them?
- Below is a typical back-end flow you may be familiar with from past lectures.



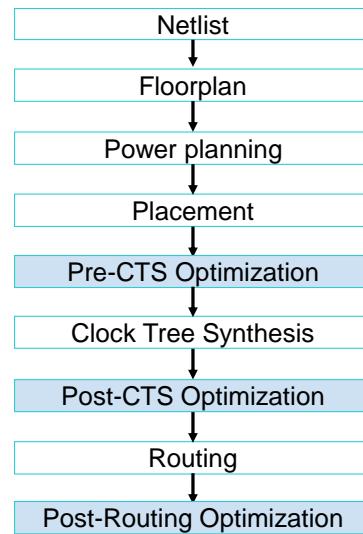
691 © Cadence Design Systems, Inc. All rights reserved.



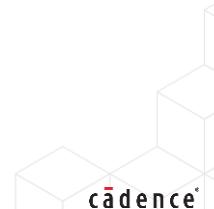
Here is the basic implementation flow. You start by reading in a netlist, floorplan the design, run power planning, placement, clock tree synthesis, and routing. There are stages in this flow that will include optimization to improve PPA.

Optimization During the Design Flow (continued)

Typically, tools have three stages of optimization within the flow:



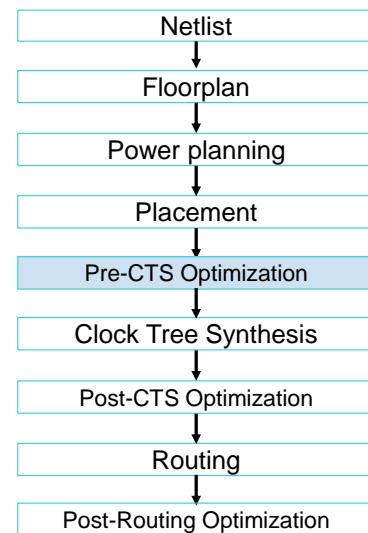
692 © Cadence Design Systems, Inc. All rights reserved.



This flowchart illustrates the stages after which optimizations will need to be run if the PPA targets have not been met. You run optimization after placement, which is called pre-CTS optimization. After clock tree synthesis, optimization is called post-CTS optimization, and after routing, optimization is called post-route optimization.

Pre-CTS Optimization

- The first optimization that takes place is right after the placement stage.
- It is here that we have the most freedom.
- The techniques that are commonly used here include:
 - Inserting buffers for high fanout nets.
 - Upsizing and downsizing gates.
 - Restructuring logic to meet timing.
- Because the metal routes are not in place yet, we cannot perform any optimization by moving metal layers.



693 © Cadence Design Systems, Inc. All rights reserved.



The first optimization that takes place is right after the placement stage. Because this is an initial stage of implementation, it is here that we have the freedom to make bigger changes to the design.

The techniques that are commonly used during pre-CTS optimization are, inserting buffers for high fanout nets, upsizing and downsizing gates, and restructuring logic to meet timing.

Because the detailed routes are not in place yet, we would not be able to optimize the timing with routing changes.

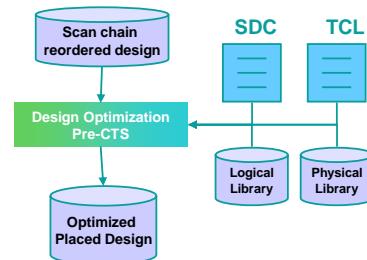
Pre-CTS Design Optimization: Input, Output, and File Format

- Input

- Scan chain reordered design
- Constraints in Standard Design Constraints (SDC) format (ideal clocks)
- Logical Timing Libraries in Liberty (.lib) format
- Physical Libraries in LEF format
- Commands in Tcl

- Output

- Optimized placed design



694 © Cadence Design Systems, Inc. All rights reserved.



The input to optimization is a scan chain reordered and placed design. Other inputs include SDC constraints, timing libraries, physical Libraries in LEF format, and additional commands in Tcl format.

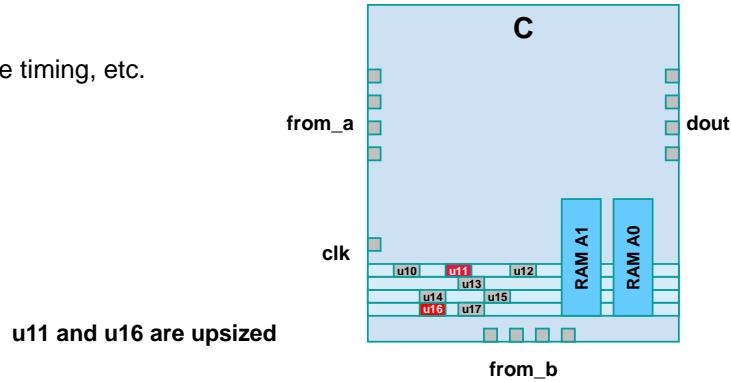
The result or output of pre-CTS optimization is an optimized placed design.

Example: Pre-CTS Design Optimization

Because logical synthesis uses wire load models (estimates of net delay), the design choices it makes can sometimes lead to sub-optimal results in placement.

Pre-CTS design optimization can clean up some of these issues by:

- Upsizing or downsizing cells
- Buffering nets
- Re-synthesizing paths to improve timing, etc.

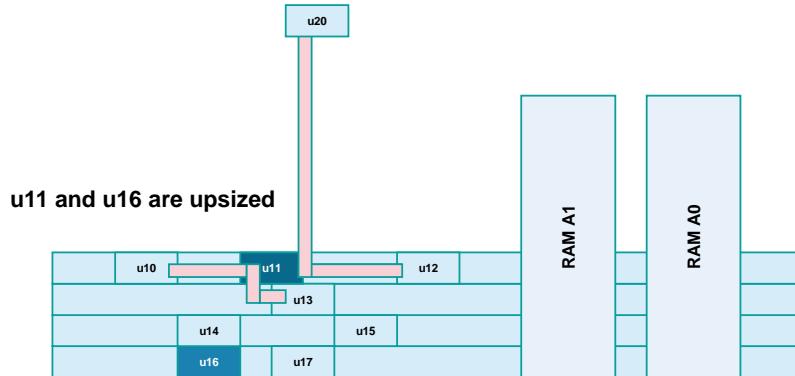


695 © Cadence Design Systems, Inc. All rights reserved.

Because logical synthesis uses estimates of net delay, the design choices it makes can sometimes lead to sub-optimal results in placement. Pre-CTS design optimization can fix some of these issues by sizing cells. This could be upsizing or downsizing the nets, buffering nets, re-synthesizing paths to improve timing, etc. In the example that is illustrated, cells u11 and u16 have been upsized as a result of optimization. This upsizing could have been to either fix timing or mitigate an SI issue.

Example: Pre-CTS Design Optimization (continued)

Cell *u11* was driving several cells, and one of them, *u20*, was far away. In order to drive the long net and meet timing, the cell was upsized. Cell *u16* was upsized for the same reason.



696 © Cadence Design Systems, Inc. All rights reserved.



In this example, cell *u11* is driving several cells, one is *u12*, which is close, and another is *u20*, which is far away. To drive the long net and meet timing, the *u11* cell was upsized.

Post-CTS Design Optimization: Input, Output, and File Format

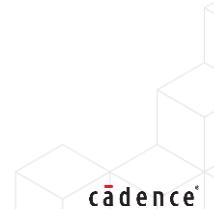
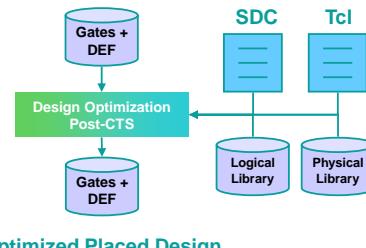
- Input

- Clock tree inserted in the design
- Constraints in Standard Design Constraints (SDC) format (ideal clocks)
- Logical Timing Libraries in Liberty (.lib) format
- Physical Libraries in LEF format
- Commands in Tcl

- Output

- Optimized design

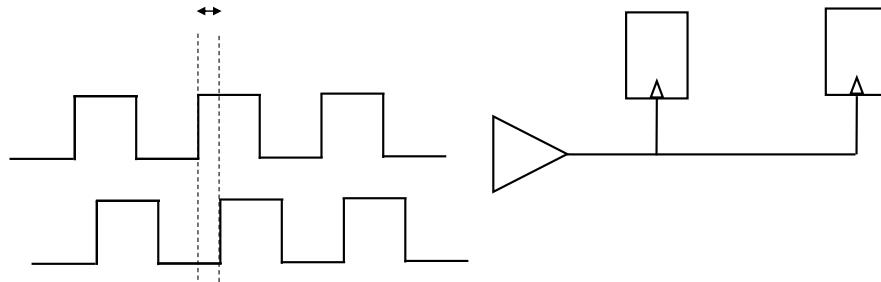
Clock Tree Synthesized Design



The inputs to post-CTS optimization are a clock tree inserted in the design, SDC constraints, timing libraries, physical Libraries in LEF format, and Tcl commands. The output is a design that contains a clock tree and has been optimized after CTS.

Post-CTS Optimization

- When the clock network is put in place, a new element comes into play called **clock skew**.
- This factor is because the clock needs to propagate from the center of the clock tree toward the peripherals.



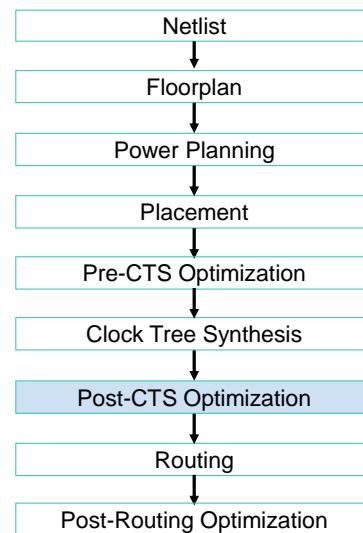
698 © Cadence Design Systems, Inc. All rights reserved.



Before clock tree synthesis, estimations for the skew are used in calculating timing. After clock tree synthesis, actual skew values as a result of the clock buffers that were added will be used for timing analysis to determine if further optimizations are required. In some cases, the addition of a buffer may result in signals arriving at the clock ports of leaf cells at different times, which is defined as clock skew.

Post-CTS Optimization (continued)

- When harmful skew is added to the timing path, the path can violate timing depending on the amount of the skew and the nature of the path.
- To mitigate the effects of skew, you can:
 - Insert buffers in the clock tree to lessen the skew.
 - Re-time and use any of the previously mentioned techniques to fix timing.
- Once again, no metal routes have been placed, although the clock signals are often routed during clock tree synthesis.



699 © Cadence Design Systems, Inc. All rights reserved.

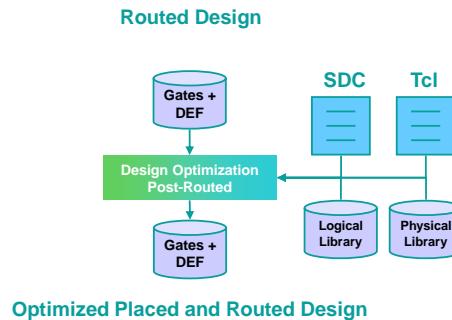


When skew is added to the timing path, the path can violate timing depending on the amount of the skew and the characteristic of the path.

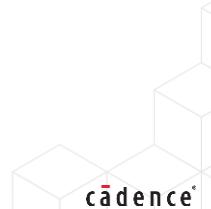
To mitigate the effects of skew, you can insert buffers in the clock tree to lessen the skew so that signals arrive at the clock pins at close to the same time, within a tolerance. Re-time and use any of the previously mentioned techniques to fix timing. A part of optimization would be to route the connections of the buffers.

Post-Routed Design Optimization: Input, Output, and File Format

- Input
 - Routed design
 - Constraints in Standard Design Constraints (SDC) format (ideal clocks)
 - Logical Timing Libraries in Liberty (.lib) format
 - Physical Libraries in LEF format
 - Commands in Tcl
- Output
 - Optimized design



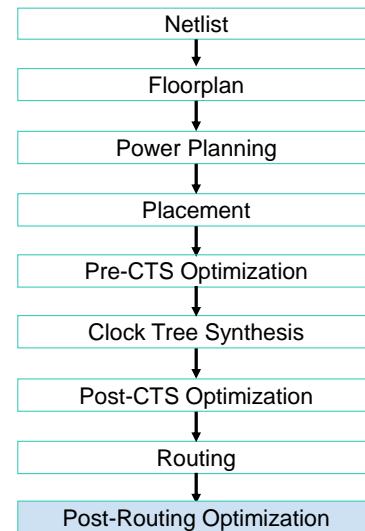
700 © Cadence Design Systems, Inc. All rights reserved.



The input to post-route optimization is a routed design, constraints in SDC format, .lib files, physical libraries in LEF format and Tcl commands. The output is an routed and optimized design.

Post-Routing Optimization

- Now that the design is fully placed, routed, powered, and clocked, it is time to undergo the final phase.
- This is the stage to perform fixes on hold violations.
- Note, however, that at this stage, there is usually not enough room to do much modification.
- Moving standard cells and macros may require intensive re-routing.
- Therefore, the following techniques are usually used:
 - Changing metal layers
 - Moving metal layers
 - Resizing gates



701 © Cadence Design Systems, Inc. All rights reserved.



Now that the design is fully placed, routed, powered, and clocked, it is time for the final phase. Before routing the design, the focus of optimization would have been to fix setup violations. During post-route optimization, the focus will shift to fixing hold violations.

Since the design is already routed, there is usually not a lot of flexibility to do lots of design modifications. Too many design modifications will disrupt the implementation that has already occurred. Moving standard cells and macros may require intensive re-routing, and therefore too many moves at this stage will not be performed.

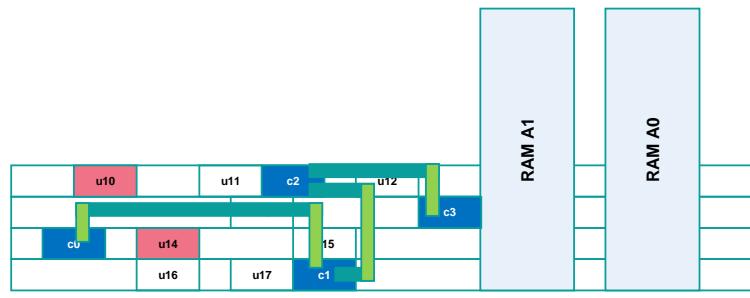
To minimize the disruption to the routed design, the techniques of changing metal layers, moving metal layers, and resizing gates will be used to fix the remaining violations.

Example: Design Optimization, Post-Route

Another round of design optimization takes place because the timing is more realistic now that there are actual wires and not just estimates of wires.

Post-route optimization can include:

- Buffering
- Upsizing or downsizing cells
- More advanced and aggressive modifications



702 © Cadence Design Systems, Inc. All rights reserved.



This example illustrates post-route optimization, which takes into account the resistance and capacitance of the actual routed wires. The timing at this post-route stage is more realistic now that there are actual wires and not just estimates of wires. Post-route optimization can include buffering nets, upsizing, or downsizing cells. More aggressive modifications would be the last resort as it would result in disrupting a routed design.

What Is Timing Closure?

A placed and routed design achieves timing closure when it meets its timing specifications while also satisfying electrical, design rules, and signal integrity constraints.

- Timing closure is often one of the greatest causes of tapeout schedule slips.
- The problem lies in the discrepancy between front-end and back-end designers' concepts of timing.
- There are several methods of addressing this discrepancy from within implementation tools.

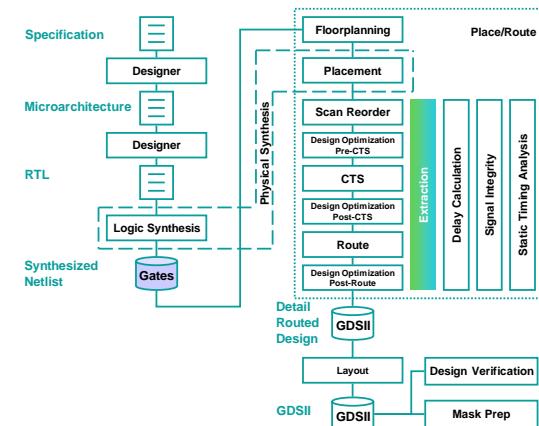


A placed-and-routed design achieves timing closure when it meets its timing specifications while also satisfying electrical, design rule, and signal integrity constraints. Timing closure is often one of the greatest causes of ASIC tapeout schedule slips. The problem often lies in the discrepancy between frontend and backend designers' concepts of timing or the introduction of new violations with or without the packaging effects. There are several methods of addressing this discrepancy from within the implementation tool and in the form of Engineering Change Orders or the ECOs.

What Is Extraction?

The process of calculating the parasitic resistance and capacitance of the interconnect of the physical design.

Example: Extraction can be performed at various parts of the design with varying accuracy. The most accurate results are achieved when the extraction is performed on a fully routed design because all of the nets are of known metal type and length. There are no estimates for nets at this point.

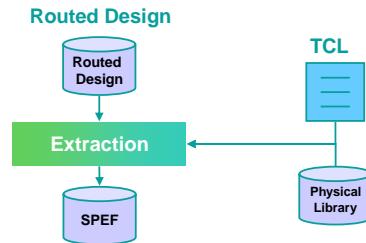


704 © Cadence Design Systems, Inc. All rights reserved.

Each interconnect, wire or net in a design can be modeled as a network of resistance and capacitance. Modeling techniques vary from extraction using 3D field solvers, to 2D extraction tools. Field solvers are much more accurate compared to 2D extraction tools but consume a lot of time. Most design tools use something called 2.5D extraction to improve accuracy and speed up runtime and use correlation mechanisms to compare with the 3D field solvers. Extraction can be performed at various steps of the design with varying accuracy. But, the most accurate results are achieved when the extraction is performed on a fully routed design because the routing length and the metal type are known at this stage. Extracted values of resistance and capacitance are saved in a Standard Parasitic Extraction Format SPEF file.

Extraction: Input, Output, and File Format

- Input
 - Routed design
 - Physical Libraries in LEF format
 - Extraction constraints and commands in Tcl
- Output
 - Standard Parasitic Extraction Format (SPEF) file containing all of the RC information for the routed nets in the design



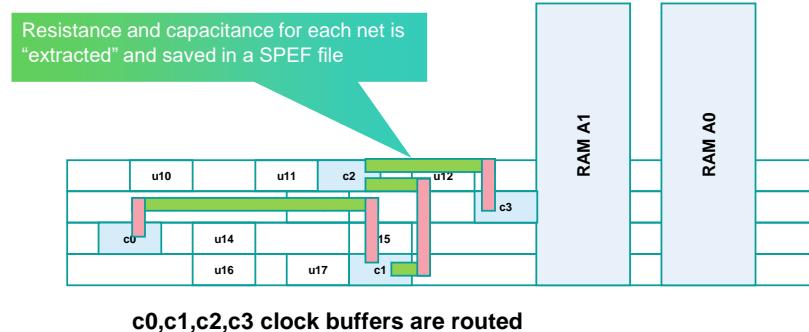
705 © Cadence Design Systems, Inc. All rights reserved.



The input to the extraction is a routed design, physical libraries in LEF format, extraction constraints, and Tcl commands. The output is a Standard parasitic extraction format file with resistance and capacitance information of all the routed nets in the design. The level of accuracy of the output SPEF format depends on the tool and the technique used to extract the design.

Example: Extraction

- When the design has been routed, we can perform a detailed extraction of the resistance and capacitance of the routed nets in the design.
- This RC data will give us a more accurate report of the timing and power of the design.



706 © Cadence Design Systems, Inc. All rights reserved.

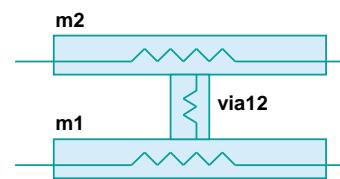


When the design has been routed, we can perform a detailed extraction of resistance and capacitance of routed nets in the design and save them in an SPEF file. Using the RC data of each net will give us a more accurate report of the timing and power of the design.

Resistance and Capacitance

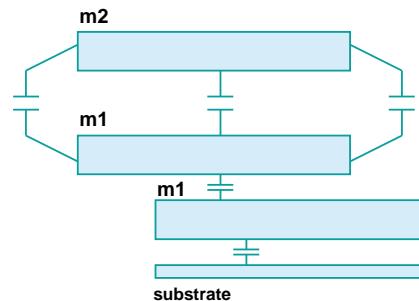
Resistance calculations are typically simple:

- Single layers
- Vias and via arrays



Capacitance calculations can be very complex:

- Multi-layer
- Multi-dimension
- Coupling capacitances:
 - Line-to-ground (net to substrate)
 - Line-to-line (nets on same layer)
 - Crossover (nets on different layers)



Resistance calculations are typically simple calculations involving layer resistance and resistance for vias and via arrays.

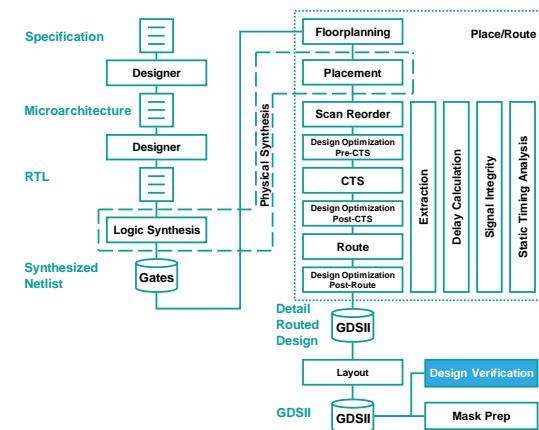
Capacitance calculations can get complex, and the availability of the many choices of 2D, 2.5D, and 3D modeling techniques and tools echo those complexities.

Even though typical resistance calculations are simple, for advanced techniques, like bias, erosion, and metal fill, or for complex metal shapes, resistance calculation may become quite complicated.

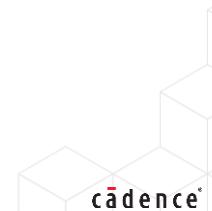
What Is Design/Physical Verification?

Layout versus schematic (LVS), design rule check (DRC), and power (IR drop and EM) are signoff checks run to ensure the integrity, functionality, and manufacturability of the chip.

- LVS is a comparison of transistor-level SPICE netlist vs. GDSII to ensure the connectivity of the design.
- DRC is a detailed check of the physical design against the process technology rules.
- IR drop is a detailed check of the chip's power plan to ensure that the supply voltages do not drop below-accepted levels.
- EM is a detailed check to ensure that the current density in all parts of the design does not exceed acceptable levels.



708 © Cadence Design Systems, Inc. All rights reserved.



Physical verification is the process of comparing the layout versus the schematic, running design rule checks, IR drop analysis, and electromigration analysis. These checks are performed to ensure the reliability, functionality, and manufacturability of the chip. LVS is a comparison of transistor-level SPICE netlist versus GDS2 to ensure the connectivity of the design.

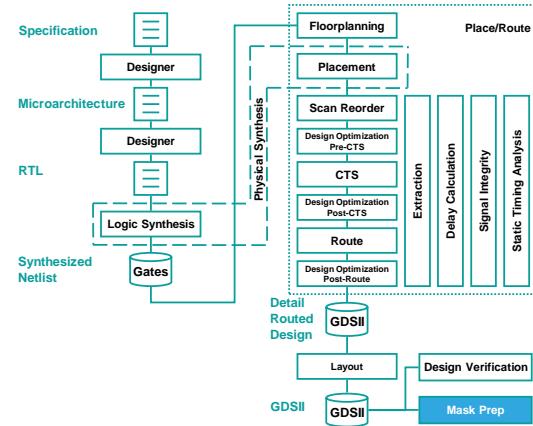
DRC is a detailed check of the physical design against the process technology rules.

IR drop is a detailed check of the chip's power plan to ensure that the supply voltages do not drop below acceptable levels.

EM is a detailed check to ensure that the current density in all parts of the design does not exceed acceptable levels.

What Is Mask Prep?

The process of creating the mask set from the GDSII database to allow chip manufacturing.



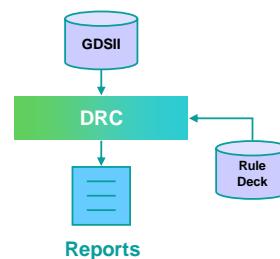
709 © Cadence Design Systems, Inc. All rights reserved.



Mask prep is the final step in the implementation flow, which is the process of creating the mask set from the GDS2 database for manufacturing the chip.

DRC: Input, Output, and File Format

- Input
 - GDSII
 - Rule deck
- Output
 - DRC reports



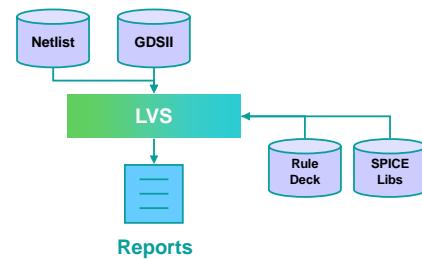
710 © Cadence Design Systems, Inc. All rights reserved.



The input files to Design Rule Check is a design database in GDS2 file format and the rule deck to check the design against. The output is a list of violations that will need to be fixed before the final GDS2 is used for manufacturing. Examples of rules that must be fixed are minimum wire spacing violations and violations related to minimum wire widths and minimum areas.

LVS: Input, Output, and File Format

- Input
 - Netlist
 - GDSII
 - Rule deck
 - SPICE libraries
- Output
 - LVS reports



711 © Cadence Design Systems, Inc. All rights reserved.



The inputs to layout versus schematic or LVS are the logical netlist, the layout in GDS2 format, the rule deck, and SPICE libraries. The SPICE libraries are used to extract the netlist from the layout, and the resulting netlist is compared to the Verilog netlist to make sure that both netlists match. The output of LVS is a report of layout versus schematic comparison violations. Examples of violations include shorted nets and open nets.

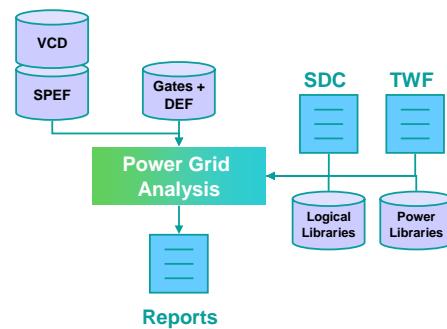
Power Grid Analysis, IR Drop, and EM: Input, Output, and File Format

- Input

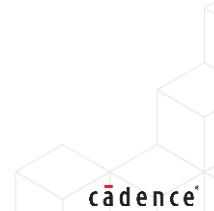
- Gate Level Netlist
- Power characterized libraries in tool-specific format
- Timing libraries in Liberty (.lib) format
- Timing constraints in SDC format
- Extraction data in SPEF format
- Timing windows file (TWF)
- Value-change-dump file (optional)

- Output

- IR drop reports
- EM reports



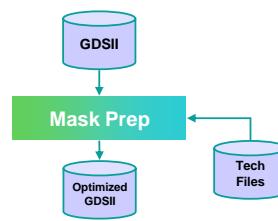
712 © Cadence Design Systems, Inc. All rights reserved.



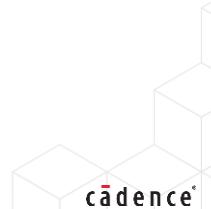
The required inputs for power grid analysis, IR Drop, and EM analysis consist of a gate-level Verilog netlist, power characterized libraries in a tool-specific format, timing libraries or .libs, SDC, LEF, and DEF, SPEF, timing window files, and optional activity information in VCD format. The output is a variety of plots and reports for viewing. Examples of plots and reports are power and rail plots, IR and EM reports.

Mask Prep: Input, Output, and File Format

- Input
 - GDSII
 - Technology specific files
- Output
 - Optimized GDSII



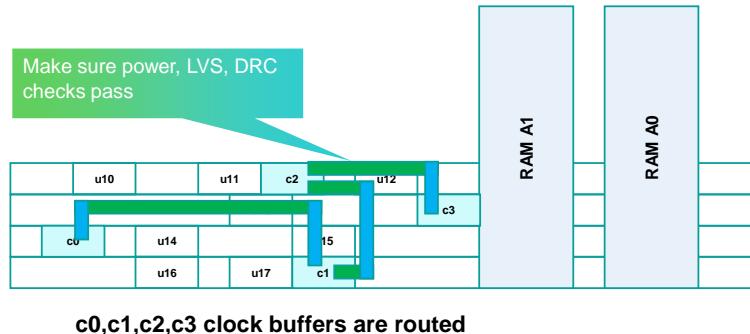
713 © Cadence Design Systems, Inc. All rights reserved.



The input files to mask prep are the physical design in GDS2 format and technology-specific files. The process of mask prep converts the GDS2 file to instructions that a photomask writer uses to generate the masks that are required to manufacture the part.

Example: Physical Verification Before Mask Prep

- Physical verification involves power, LVS, and DRC checks to ensure the integrity of the design.
- When the design passes all of the PV checks, a GDSII is produced, and mask prep can begin. Mask prep involves complex processes such as lithography (the process of creating the masks to create the layers for an integrated circuit), modifications, etc.



714 © Cadence Design Systems, Inc. All rights reserved.



Physical verification involves power verification, LVS, and DRC checks to ensure the integrity of the design.

When the design passes all of the physical verification checks, a GDS2 is produced, and mask prep can begin. Mask prep involves complex processes such as lithography, which is the process of creating the masks to create the layers for an integrated circuit, modifications, etc. In the diagram that is shown, for the cells and the wires in the design, all the validation checks have to pass before design masks are created.

Discussion Questions

- What are the main process steps in the physical design of a chip?
- Which process steps can be done at multiple stages of the flow?
- If you were to lead the design of a chip, how would you organize your resources to handle the various tasks?



What are the main process steps in the physical design of a chip? Which process steps can be done at multiple stages of the flow? If you were to lead the design of a chip, how would you organize your resources to handle the various tasks?

Test Your Understanding

True or False

1. In creating a floorplan, we can gather information to see if our design is routable.
2. If a design does not meet timing after synthesis, it is possible that it can meet timing during placement.
3. When routing a design, it is best to avoid having long parallel routes.
4. Accurate SDC constraints are important to meet timing during static timing analysis.
5. Errors in physical verification are simple to fix.



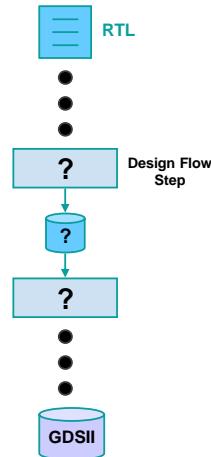
Here are a few questions to test your understanding.

Learning Activity

In this activity, you will:

- Complete a flowchart of the digital design implementation flow.
- Include the design flow steps.
- Include the necessary inputs and outputs.
- Fill in the missing or wrong sections of the flowchart.

10 minutes for debriefing



717 © Cadence Design Systems, Inc. All rights reserved.



Based on this flowchart, what steps should be included that are missing, and what are the input and output files?

Terms and Definitions

Floorplanning	Process of deriving the die size, allocating space for soft blocks, planning power, and macro placement
Placement	Process of placing the standard cells in a floorplanned design
Clock Tree Synthesis	Process of inserting buffers in the clock tree of a digital design
Route	Process of connecting the pins of the standard cells, macros, and I/Os of a digital design to specific metal layers in the process technology to match the schematic
Extraction	Process of calculating the parasitic resistance and capacitance of the interconnect of the physical design
Delay Calculation	Process of computing the delay of interconnect and standard cells in a digital design
Static Timing Analysis	Process of computing the timing of logically related paths for a digital design without regard to large scale functional behavior
Signal Integrity	Unintended effects on digital signals caused by interconnect parasitic resistance or capacitance that causes noise and/or changes delays
Design Optimization	Process of using automated algorithms to improve the quality of a digital design
Physical Synthesis	Process of combining logic synthesis and placement to improve the accuracy of the physical implementation of a digital design
Design Verification	Process of physically verifying the design rules and backend checks of a design
Mask Prep	Process of creating the mask set from the GDSII database to allow chip manufacturing

718 © Cadence Design Systems, Inc. All rights reserved.



This slide contains a list of terms, definitions, and acronyms. Spend a few minutes reading through this slide.

Terms and Definitions (continued)

LEF	Library Exchange Format, Physical Library (metal and via routing rules)
DEF	Design Exchange Format, Physical (floorplanning, placement, routing) and Logical Representation (connectivity)
Liberty	Format for logical libraries, includes timing, area, and power information
SDC	Standard Design Constraints, includes clocks and timing constraints
Clock Skew	Delay difference between clock paths in a design
Clock Latency	Delay from clock source to destination in a design
SPEF	Standard Parasitic Exchange Format, standard format for representing capacitance and resistance for each net
SDF	Standard Delay Format, standard format for representing interconnect and cell delays
LVS	Layout vs. schematic, connectivity checking
DRC	Design Rule Check, physical rule checking
IR Drop	Voltage Drop, measure of power plan integrity
EM	Electromigration, term used to describe failures in wires due to high current
TWF	Timing Windows File, file used in signal integrity analysis to determine the overlap of signals
VCD	Value Change Dump, file used to provide toggle information to power analysis
GDSII	Graphic Data System, standard format for IC layout data exchange
Rule Deck	Technology specific information used by physical verification
Spice Deck	Format to represent circuits, cells, and macros in detail

719 © Cadence Design Systems, Inc. All rights reserved.



This reference slide defines the terms that you have seen so far. Take a few moments to read it.

Module Summary

In this module, you:

- Demonstrated the implementation flow at a high level.
- Described the purpose of logic synthesis and the function of Static Timing Analysis in the logic synthesis flow.
- Identified Design for Test (DFT), built-in self-test, and Joint Test Action Group (JTAG) design techniques.
- Described the following steps in the physical implementation flow:
 - Floorplanning
 - Placement
 - Physical synthesis
 - Static timing analysis
 - Clock tree synthesis
 - Pre- and post-CTS optimization
 - Routing
 - Extraction
 - Delay calculation
 - Timing analysis
 - IR drop analysis
 - Design verification
 - Mask prep

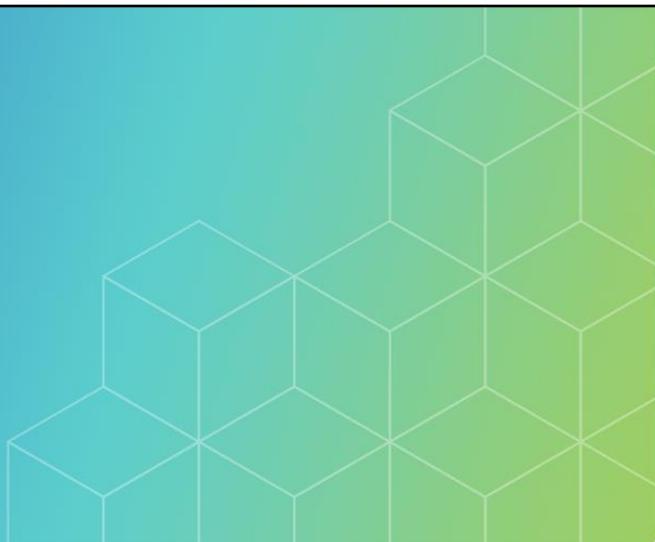
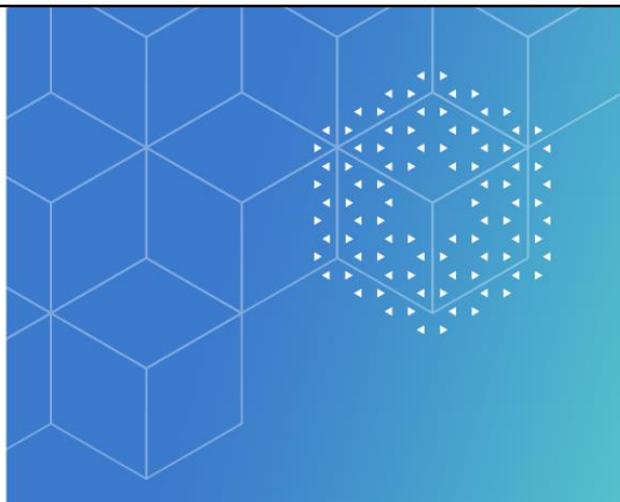
720 © Cadence Design Systems, Inc. All rights reserved.



In this module, we described the implementation flow at a high level and described the purpose of logic synthesis and the function of static timing analysis in the logic synthesis flow.

We identified the Design for Test, built-in self-test, and JTAG design techniques.

We described the following steps in the physical implementation flow; floorplanning, placement, physical synthesis, static timing analysis, clock tree synthesis, pre-and post-CTS optimization, routing, extraction, delay calculation, IR drop analysis, design verification, and finally, mask prep.



Module 7

IC Packaging

cadence®

IC packaging is the final stage of semiconductor device fabrication. The IC is encapsulated in a supporting case that protects the IC from physical damage and corrosion and supports the electrical contacts that connect the device to a higher-level carrier, often a printed circuit board. A very large number of different types of packages exist. Some package types have standardized dimensions and tolerances and are registered with trade industry associations such as JEDEC and Pro Electron. Other types are proprietary designations that may be made by only one or two manufacturers.

Module Objectives

In this module, you will:

- Identify the different IC packaging styles.
- Explain the different steps in the IC package implementation process.
- Check the PDN for DC voltage drop.
- Check the PDN for AC impedance across a frequency spectrum.
- Run a 3D-EM analysis to generate an S-parameter model.
- Analyze the thermal performance of the IC package.

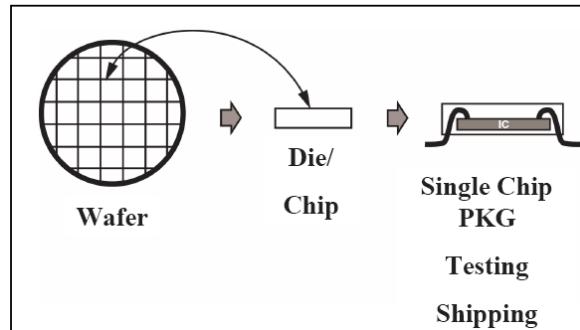


In this module, we will identify the different IC packaging styles that are commonly used and explore the different steps in the IC Package implementation process. We will discuss the need for checking the power delivery network for DC Voltage Drop through the package, as well as the need for a low AC impedance path in the package across a wide frequency bandwidth. We will also discuss 3D-EM simulation and how it is used to generate S-parameter models for the package that can be used in system-level 3D-EM simulations, as well as how the thermal performance of the IC package is important and how it can be analyzed. Finally, we will describe the different types of package models that can be generated for the package so the IC and its package can be included in system-level simulations.

What Is an IC Package?

An IC package provides electrical interconnection, mechanical support, environment protection, and a thermal path between the chip and the next-level board.

Chip (die) + Package (case) = Device
packaging

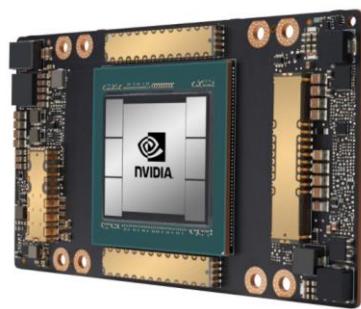
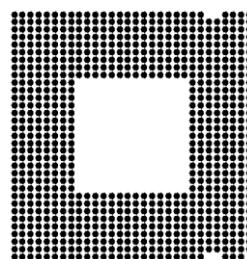
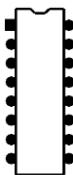


723 © Cadence Design Systems, Inc. All rights reserved.



An IC package provides electrical interconnection, mechanical support, environment protection, and a thermal path between the IC and the next-level carrier, usually a printed circuit board. Integrated circuits are manufactured as an array of ICs on a wafer. After the ICs are manufactured, the wafer is cut down to individual ICs. The ICs are far too small to be assembled on a printed circuit board or large system carrier, so they are usually individually packaged in a device or IC package. Although bare dies are sometimes purchased from an IC vendor, they are more often available in a variety of package styles and sizes that will have different electrical, thermal, and environmental characteristics.

Increasing IC I/O Drives Greater Package I/O



1971

- Intel 4004 microprocessor
- 2250 transistors
- 16-pin dual inline package
- 7.8 mm x 22.3 mm

2005

- Intel Pentium IV 6xx processor
- 169,000,000 transistors
- 775 pin grid array
- 37.45 mm x 37.45 mm

2020

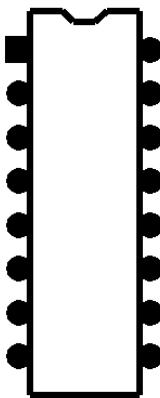
- Nvidia GA100
- 28,000,000,000 transistors
- 826 mm²

724 © Cadence Design Systems, Inc. All rights reserved.

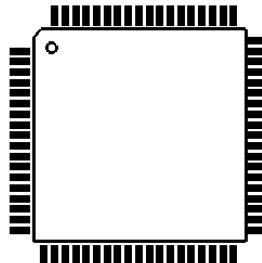


As transistors get smaller, density increases, and the amount of input and output pins required also increases. In the early 1970s, a typical IC had a couple of thousand transistors and 10 to 20 pins. By the early 2000s, a typical processor had over 150 million transistors and required nearly 1000 pins. Today's typical processors have billions and billions of transistors and are often limited by the number of input and output pins that can be put on a package. The shrinking of the transistor has also allowed more and more functionality to be placed on a single die allowing for the advent of an entire system on a single silicon die.

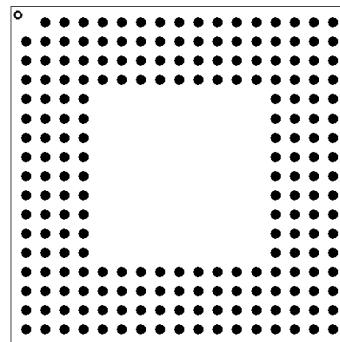
Package Formats



DIP (Dual Inline Package)



Peripheral Pins



Area Pins

Packaging formats have changed over the years, mainly to meet the demand for higher and higher pin count devices. In the beginning, DIP packages with pins on either side of the device had an I/O density great enough to meet the IC demands of the times. As the number of transistors, and therefore the number of inputs and outputs, increased, the packages became mostly peripheral pin devices that had pins on all four sides of the device. Today ICs, with much higher pin counts, require an area of pins under the whole device. The demand for more power to power the incredible number of transistors has also exasperated the problem. DIP packages often have a single power and ground pin, while a typical peripheral pin package may have a couple of power and ground pins on each side of the device. For area-based packages for large power-hungry ICs, it is not uncommon for nearly half of the pins to be dedicated to power and ground.

What Is Package Efficiency?

Package efficiency (PE) is the ratio of the active die and the area required to attach the packaged device to the next level of interconnect (usually the PCB).

$$\frac{\text{Die area}}{\text{Package area}} = \text{PE}$$



Package efficiency is the ratio of the active die and the area required to attach the packaged device to the next level of interconnect, usually the printed circuit board. The requirement for a larger number of I/Os and the need for more power and ground pins prevent the packages from shrinking at the same rate as the silicon, so packaging efficiency decreases.

Test Your Understanding

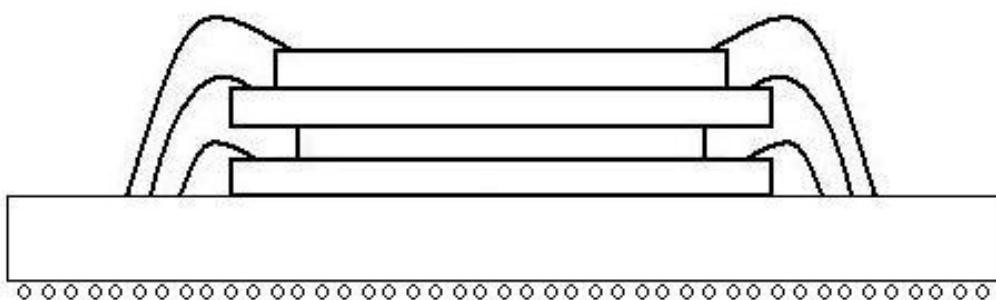
Is it possible to achieve a PE of greater than 1.0?

$$\frac{\text{Die area}}{\text{Package area}} = \text{PE}$$



As the number of I/Os increases and packaging efficiency suffers, new methods are often employed to keep package efficiency in check. However, is it possible to achieve a packaging efficiency of greater than 1.0, where the silicon area inside the device is actually greater than the area used by the packaged device?

Answer: Test Your Understanding



Indeed. Many vendors are starting to employ stacked die devices where multiple ICs are packaged in the same device.

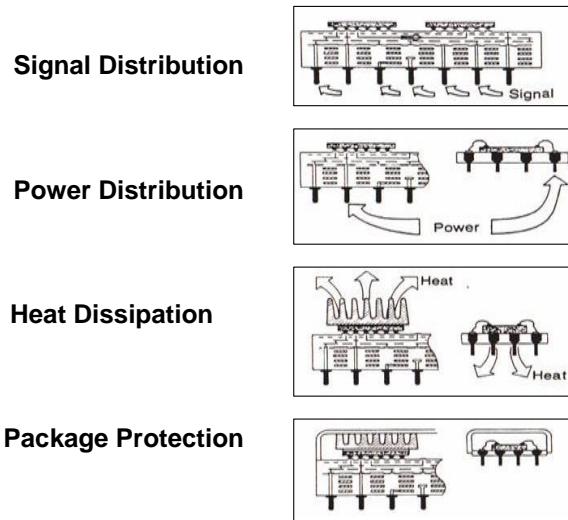
728 © Cadence Design Systems, Inc. All rights reserved.



A packaging efficiency of greater than 1.0 can be achieved by stacking multiple die in the same package. 3D Packaging refers to 3D integration schemes that rely on traditional methods of interconnect, such as wire bonding and flip chip, to achieve vertical stacks.

What Are the Functions of an IC Package?

An IC package provides electrical interconnection, power distribution, mechanical support, environment protection, and a thermal path between the chip and the next level board.



729 © Cadence Design Systems, Inc. All rights reserved.



The IC package that a die is packaged in has many different functions. The main function of an IC package is to distribute the I/O signals from the die to the outside of the package, where they can be accessed for the next higher-level packaging in the system, which is often a printed circuit board. As the power of each die increases, the power distribution of the package plays a more dominant role, and, in some packages, nearly half the pins on the package may be dedicated to supplying power to the IC. Along with higher power comes increased heat, and the package must provide a path for the heat to be dissipated away from the silicon out through the package. Lastly, the package also provides protection to the silicon to protect it from environmental effects such as humidity and corrosion.

What Are the Two Main Die Attachment Methods?

An IC package provides electrical interconnection, mechanical support, environment protection, and a thermal path between the chip and the next level board.

Wire Bond



The IC is packaged with the pins up and small wires are used to connect the IC to the substrate.

Flip Chip



The IC is packaged with the pins down and small solder bumps are used to connect the IC to the substrate.

There are two main types of die attachment methods, wire bonding and flip chip. In a wire-bonded package, the die is placed with the pins up, and small wires are used to make connections from the IC pins to the substrate of the package. These die generally have a peripheral pin configuration where the pins are around the perimeter of the IC. In a flip chip package, the IC is flipped with its pins down so they physically connect to the top of the package substrate. These ICs generally have a gridded arrangement that may cover the entire area of the silicon. This provides a much higher pin density, but the flip chip connections are much harder to inspect and can rarely be repaired.

Three Main Wire Materials

- Gold (Au) is the most commonly used material by a large margin.
- Aluminum (Al) is used for excessive power or temperature cycling applications because of its high *coefficient of thermal expansion*.
- Copper (Cu)
 - Is cheaper than gold
 - Has low resistivity
 - Has high tensile strength
 - Oxidizes readily at elevated temperatures

Properties	Gold	Aluminum	Copper
Thermal conductivity (W/m-K)	319	237	403
Melting point	1064	660	1083
Resistivity (Ohm-m C)	2.3×10^{-8}	2.7×10^{-8}	1.7×10^{-8}
Tensile strength (Pa)	2.1×10^8	4.5×10^7	2.2×10^8
Coefficient of thermal expansion (ppm/C)	14	46	16
Thermal conductivity (W/m-K)	319	237	403



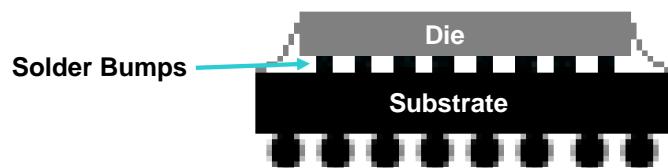
There are three main metals that are commonly used for wire bonding. Gold is by far the most commonly used material because of its great conductive properties, and it is a very malleable metal. For high-power applications, aluminum is sometimes used because its high coefficient of thermal expansion allows the wire bonds to handle the excessive power and temperature cycling. For cost-sensitive packages, copper is sometimes used in place of gold. Copper maintains a reasonably low resistivity and high tensile strength while being more affordable; however, it does oxidize readily at elevated temperatures.

Flip Chip Packaging

Flip chip packaging does not require any wire bonds. Solder is applied directly to the die pads, and the chip is attached to the package with the pins down, making a direct connection from the die to the package.

The flip chip package provides several advantages.

- **Size:** No space for wire bonds are required.
- **Performance:** No wire bond is available to add inductance.
- **Greater I/O:** Package efficiency increases because area array ICs can be accommodated.
- **Mechanical:** There are no wire-bond failures.
- **Cost per connection:** Cost increases are small because the number of I/Os increase.



732 © Cadence Design Systems, Inc. All rights reserved.



Flip chip packaging does not require any wire bonds. Solder is applied directly to the die pads, and the chip is attached to the package with the pins down, making a direct connection from the die to the package. The flip chip package provides several advantages over wire bonding, such as size since no space is required for wire bonds. The flip chip bump connection also provides a performance advantage over wire bonding; the direct IC pin to package connection does not have the extra inductance associated with a wire bond connection. Greater package efficiency can be achieved because an array of pins can be accommodated over just having pins around the perimeter of the silicon. Without the wire bonds, there can be no wire bond failures, and the cost per connection is better because a greater number of inputs and outputs can be accommodated.

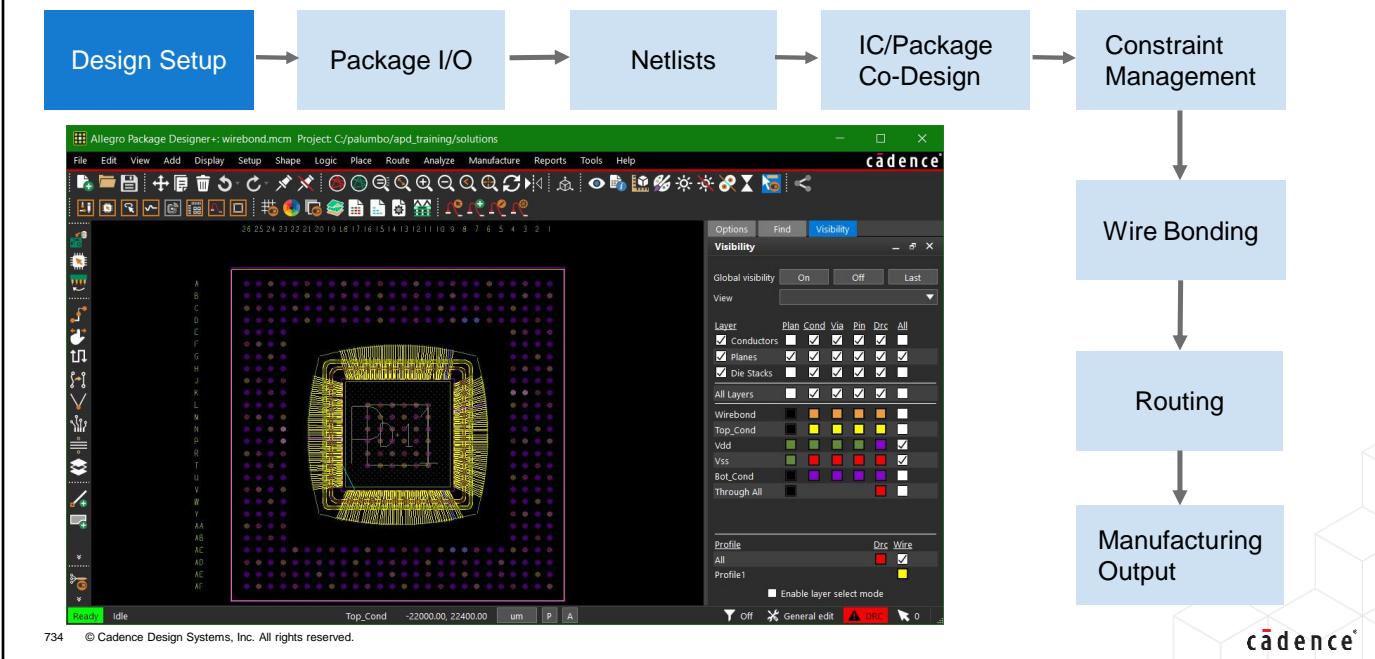
Flip Chip Disadvantages

- Bare dies are difficult to test.
- Bumped chips availability is limited.
- It is a challenge for PCB technology because pitches become very fine and bump counts are high.
- For inspection of hidden joints, X-ray equipment is needed.
- With present-day materials, an underfilling process with a considerable curing time is needed.
- Some substrates have low reliability.
- Repairing is difficult or impossible.



Although flip chip attachment does have many advantages over wire bonding, there are still some disadvantages for flip chip packages. The bare die can be difficult to test before packaging because of the density of the pins, and the bumped chips often have limited availability. The pin pitches that can be achieved by a flip chip package create a challenge for the next level of packaging, the printed circuit board. Inspection of the flip chip devices requires expensive X-ray equipment to find hidden and faulty solder joins. The materials used to attach the chip to the package substrate often have a considerable curing time which slows down the manufacturing process, and repairing defects is often difficult or impossible.

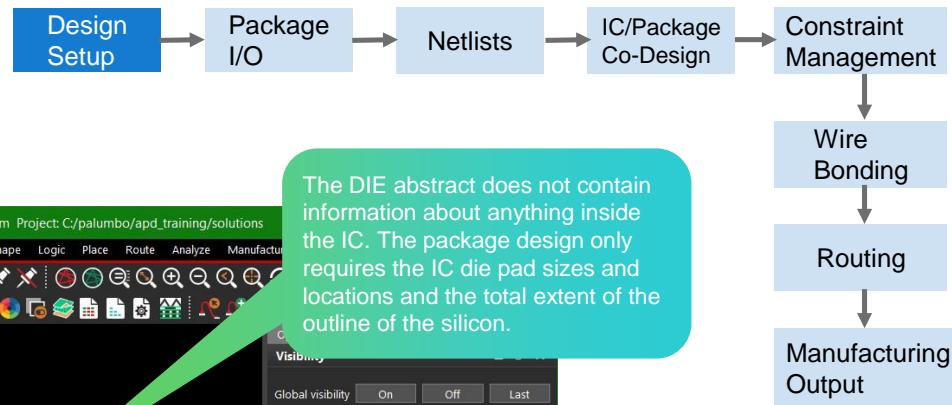
A Typical IC Package Design Flow



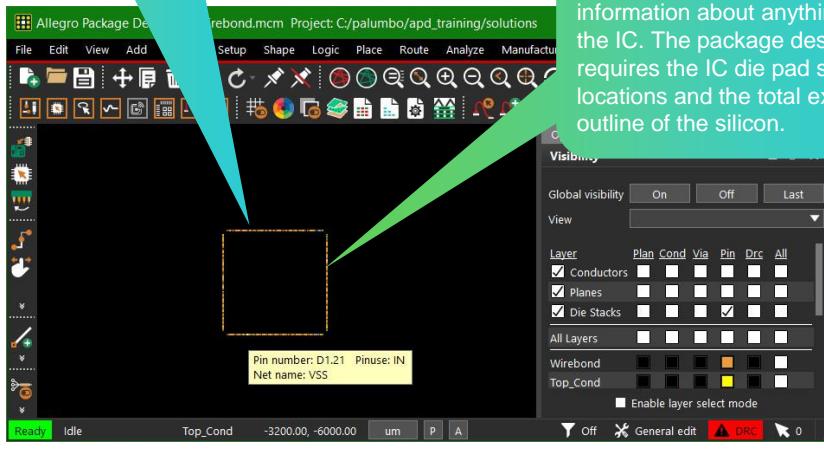
The typical IC packaging design flow is generally done using an intelligent CAD application such as APD+. APD+ will allow you to start and set up a package design by reading in pin information from an IC CAD tool and defining the layers of the IC package substrate. After importing the information for the IC pins and die size, you can define the package I/O that will interface the package to the next level of the carrier in the system, generally a printed circuit board. A netlist is derived during the package design process to define which IC pin should connect to which I/O pin on the bottom of the package. This step is often done as an IC and package co-design process. In some cases, the package design is started before the IC is completed, and optimizations based on packaging can be accommodated through IC pin swaps which can then be fed back to the IC design so the I/O of the silicon can be optimized in the context of the package design. A constraint management system keeps track of all the manufacturing and electrical constraints, and if you violate any spacing rules, the application will automatically and immediately flag a design rule violation.

After all the manufacturing and electrical constraints are set up, you can wire bond the die to the top of the package substrate, if required, and then the design is ready to be routed. A final check by the constraint system will verify the spacing rules and indicate if there are any opens from the IC pins to I/O connections on the bottom of the package. When all violations are eliminated, you can output the manufacturing files that will be used to manufacture the IC package.

IC Package Design Flow: Design Setup



A DIE abstract can be output by the IC tool and read into APD+.



The DIE abstract does not contain information about anything inside the IC. The package design only requires the IC die pad sizes and locations and the total extent of the outline of the silicon.

Manufacturing Output

Wire Bonding

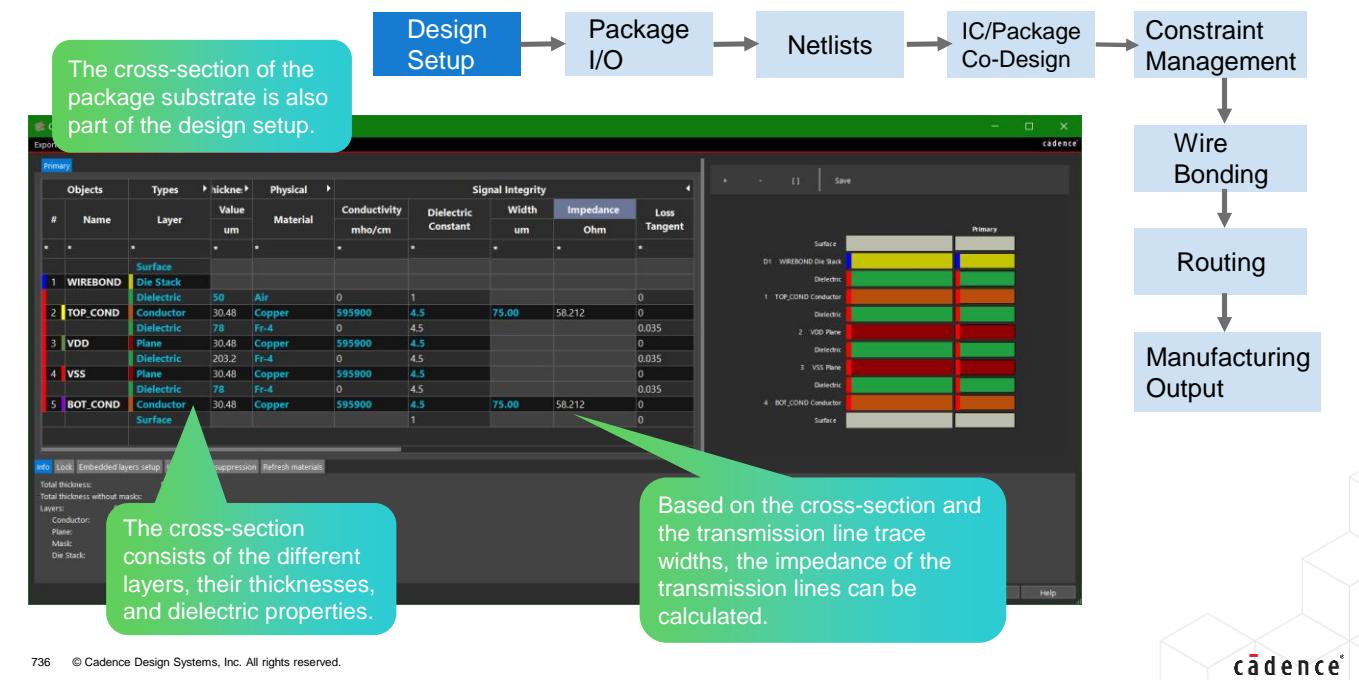
Routing

Manufacturing Output



A package design is often started by reading in a DIE abstract file that describes the die being packaged. The packaging tool does not need to know information about things internal to the IC being packaged; it only needs to know about its interface to the package, the size of the die, the size of the pads, and the exact location of each pad. Many IC tools will export a DIE abstract file that can then be read into APD+ to start the package design process.

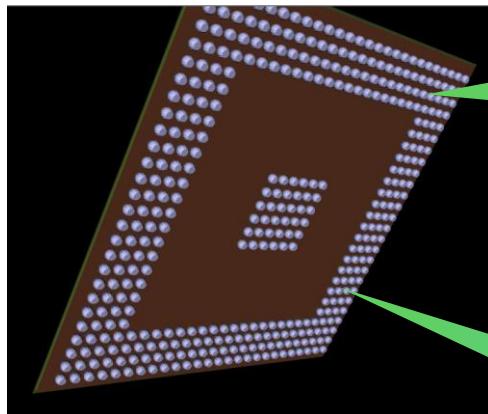
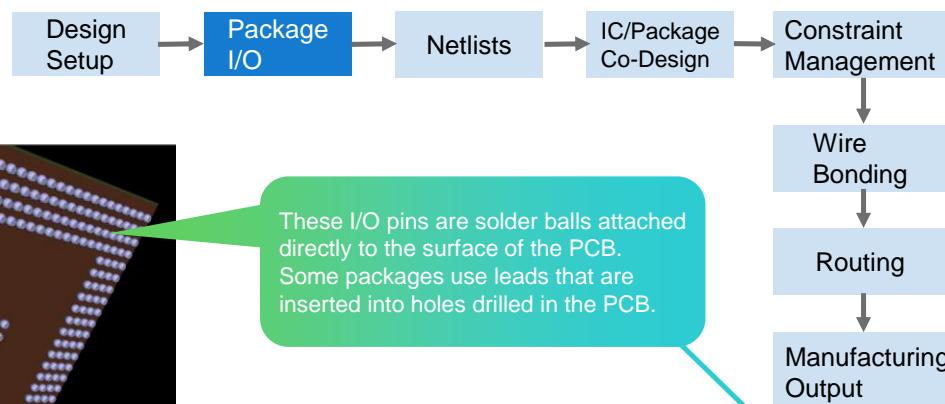
IC Package Design Flow: Design Setup (continued)



736 © Cadence Design Systems, Inc. All rights reserved.

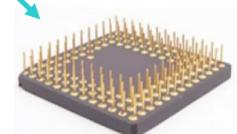
The cross-section of the IC package substrate is also part of the design setup. You define all the different layers of the substrate in the cross-section so the design tool knows how many layers are available for routing the signals in the design. The cross-section also models the thickness of the layers and the materials, including the conductivity for conductor layers and the dielectric constant for insulator layers. Armed with this information, the APD+ application can calculate the impedance of the transmission lines in the design based on the layer the transmission line is on and the trace width of the transmission line.

IC Package Design Flow: Package I/O



These I/O pins are solder balls attached directly to the surface of the PCB. Some packages use leads that are inserted into holes drilled in the PCB.

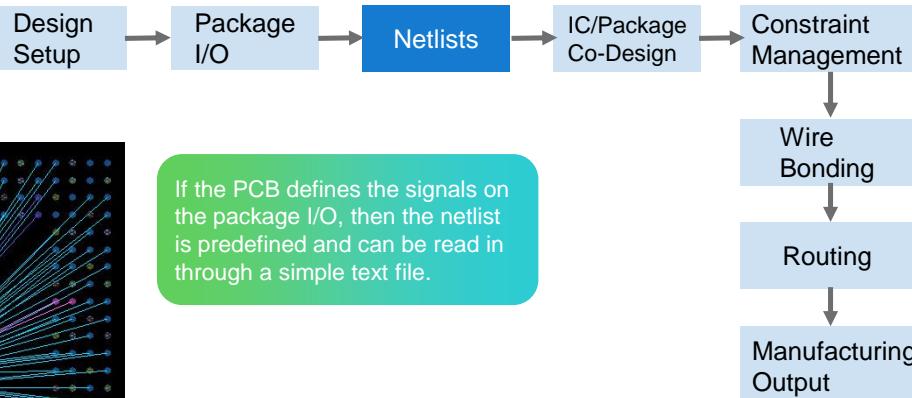
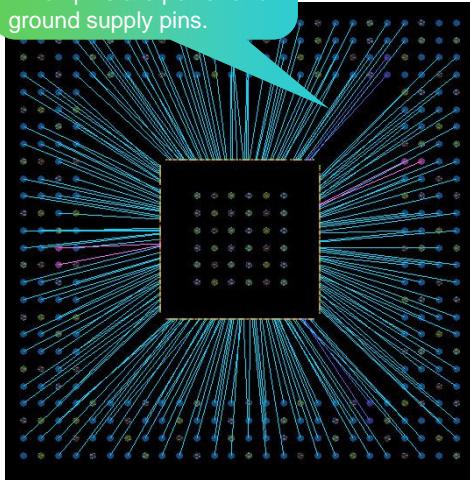
The package I/O is the interface to the next level of packaging, generally the printed circuit board.



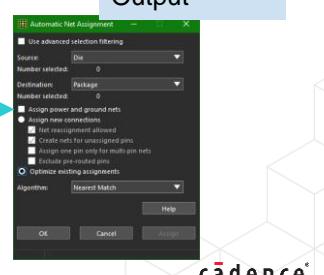
The IC package I/O may be predefined or a custom array of pins. Predefined I/O components are often read in from a text file that defines the exact XY locations of the pins or from libraries so the information can be reused across designs. The package I/O is the interface to the next level of packaging, generally the printed circuit board. Much like the IC inside the package, the PCB physical design is only dependent on this I/O component, the size of the pads connecting to the PCB, and the exact location of the pads. However, electrical modeling does require information about the package connections from the IC down to the package I/O and extraction tools are often used to extract electrical models that model the electrical characteristics of the package. The package I/O pins shown above are solder balls that connect directly to the surface of a PCB, but these are often pins that extend off the package and then are inserted into holes drilled into the PCB for through-hole packages.

IC Package Design Flow: Netlists

The netlist defines which IC pins connect to which package pins as well as which pins are power and ground supply pins.



Often, the package designer has the freedom to select the package I/O pin for each IC pin. Optimization tools help the designer reduce the number of ratsnest crossings and minimize the length of connections.

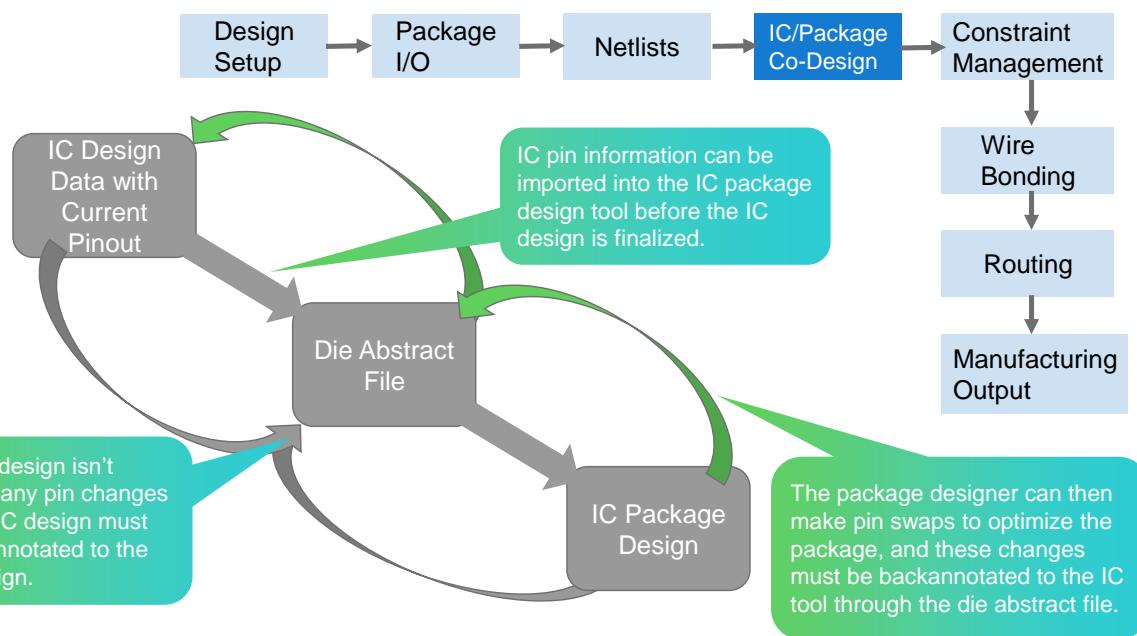


738 © Cadence Design Systems, Inc. All rights reserved.

The netlist defines which IC pins connect to which package pins, as well as which pins are power and ground supply pins.

Having a netlist-driven system allows the package design tool to flag any IC pin to I/O pin connections that have not been made and flag any manufacturing spacing violations or electrical routing rules. If the PCB defines the signals on the package I/O, then the netlist is predefined and can be read in through a simple text file. Often, the package design has the freedom to select the package I/O pin for each IC pin. Optimization tools help the designer reduce the number of ratsnest crossings and minimize the length of connections. When multiple ICs are packaged in the device, the netlist will also define the IC-to-IC connections in the package.

IC Package Design Flow: IC/Package Co-Design

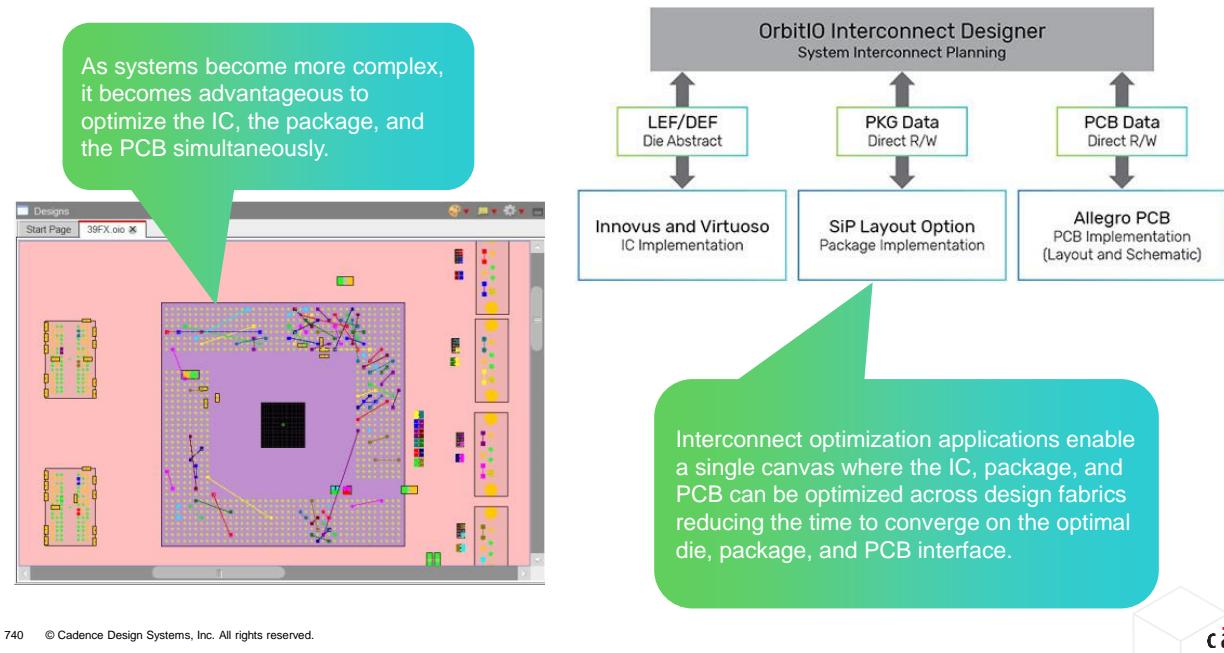


739 © Cadence Design Systems, Inc. All rights reserved.



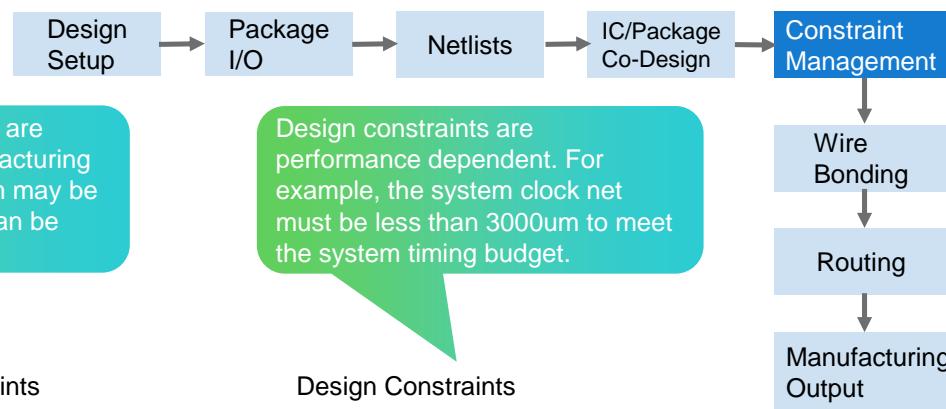
An IC and its package are often designed in parallel. The current IC pinout can be sent to the package design as a die abstract file that only contains information about the interface of the IC, its boundary, and the exact size and location of the IC pins. This can be read into the package design tool, and the IC package design can begin. By designing the IC and package in parallel, it often gives the package designer the opportunity to swap IC pins to optimize the IC package connections. These pin swaps must be backannotated to the IC design, where they will either be accepted or rejected. Since the IC design is not finalized yet, any changes to the interface of the IC, pin swaps, or edited and moved bumps must be forward annotated to the package design through the die abstract file.

IC/Package/Board Design



As systems become more complex, it becomes advantageous to optimize the IC, the package, and the PCB simultaneously. Interconnect optimization applications enable a single canvas where the IC, package and PCB can be optimized across design fabrics reducing the time to converge on the optimal die, package, and PCB interface.

IC Package Design Flow: Constraint Management



Manufacturing constraints are generally driven by manufacturing yields. For example, 90um may be the narrowest trace that can be manufactured.

Design constraints are performance dependent. For example, the system clock net must be less than 3000um to meet the system timing budget.

Manufacturing Constraints

- Minimum trace width
- Minimum trace to trace spacing
- Minimum trace to via spacing
- Minimum via to via spacing
- and dozens more....

Design Constraints

- Maximum trace length
- Matched trace lengths
- Differential pair rules
- Minimum via to via spacing
- and dozens more...

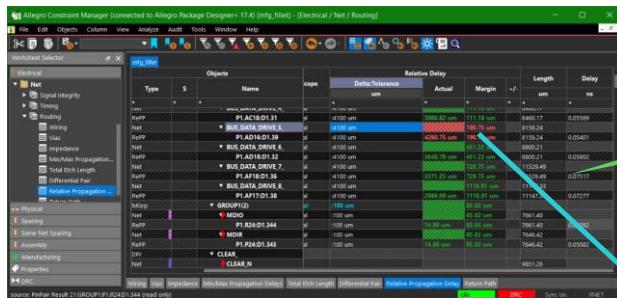
741 © Cadence Design Systems, Inc. All rights reserved.



A typical IC package design has hundreds of design constraints that must be met for thousands of connections. Keeping track of all this information is generally done through a spreadsheet-based Constraint Manager. Many of the constraints are defined by the package manufacturer and may be cost-dependent. For a lower price, you may be required to have minimum traces of 100 microns in the design, but for a high cost, you may be able to go down to 75um traces. Either way, as the IC package is being designed, the constraint system will indicate any violations that you make to the manufacturing and design constraints.

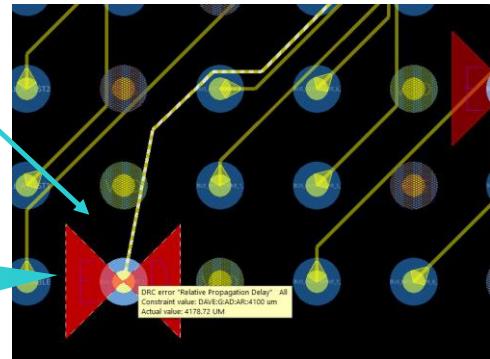
Design constraints are generally performance-driven and dependent on the IC that is being packaged. For example, you may need all the high-resolution nets to be less than 3000 microns through the package to meet the delay budget for those nets through the system. Design-dependent constraints can only be accommodated by a netlist-driven system because the netlist identifies which nets are attached to each IC pin and down to each I/O pin on the package.

Constraint Management (continued)



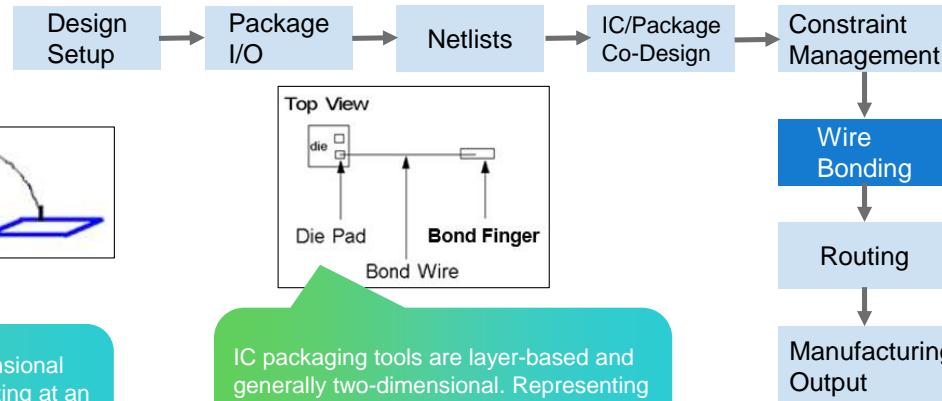
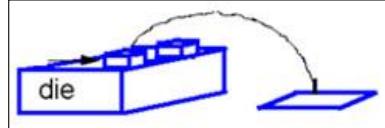
Manufacturing and design constraints can become very complicated, so they are defined in a spreadsheet-based application.

The constraint system dynamically checks constraints as the package is being designed. As soon as the designer violates any of the manufacturing or design constraints, the violation is indicated in both the Constraint Manager and on the design canvas.



Manufacturing and design constraints can become very complicated, so they are defined in a spreadsheet-based application. The constraint system dynamically checks constraints as the package is being designed. As soon as the designer violates any of the manufacturing or design constraints, the violation is indicated in both the Constraint Manager and on the design canvas.

IC Package Design Flow: Wire Bonding



Wire bonding is a three-dimensional process with a wire bond starting at an IC pin and being extruded to a bond finger on the package.

Top View
die
Die Pad
Bond Finger
Bond Wire

IC packaging tools are layer-based and generally two-dimensional. Representing objects that changed in all three dimensions was traditionally difficult.

The package design tools couldn't report the correct lengths for the wire bonds, and even worse, they couldn't check wire-to-wire interference in three dimensions.

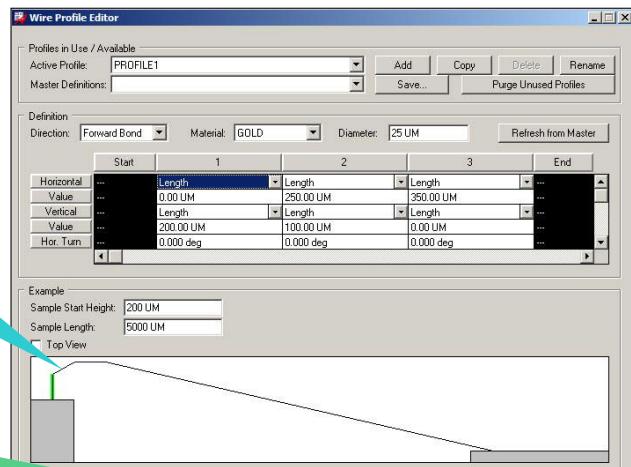
Wire bonding is a three-dimensional process with a wire bond starting at an IC pin and being extruded to a bond finger on the package surface. Traditional IC packaging tools are layer-based, and objects are only represented in the X and Y dimension. The Z dimension is handled by placing objects on different layers of the design. Representing objects that change in all three dimensions, such as a wire bond, was very inaccurate at best. The package design tools couldn't report correct lengths for the wire bonds; they could only report the point-to-point distance from the die pad to the bond finger as a straight line in the XY plane.

Wire Bond Profiles

Wire bond profiles allow for accurate representation of the three-dimensional wire bonds.

A design may have many different wire profiles, and the profiles can be defined with an unlimited number of segments for accuracy.

Wirebond Report		
Pin Number	2D Wire Length (microns)	3D Wire Length (microns)
D1.245	3545.42	3848.85
D1.175	3360.65	3666.35
D1.179	3300.12	3606.62
D1.181	3291.57	3598.19
D1.185	3222.28	3529.87
D1.178	3307.56	3613.96
D1.174	3365.83	3671.45
D1.184	3227.02	3534.54



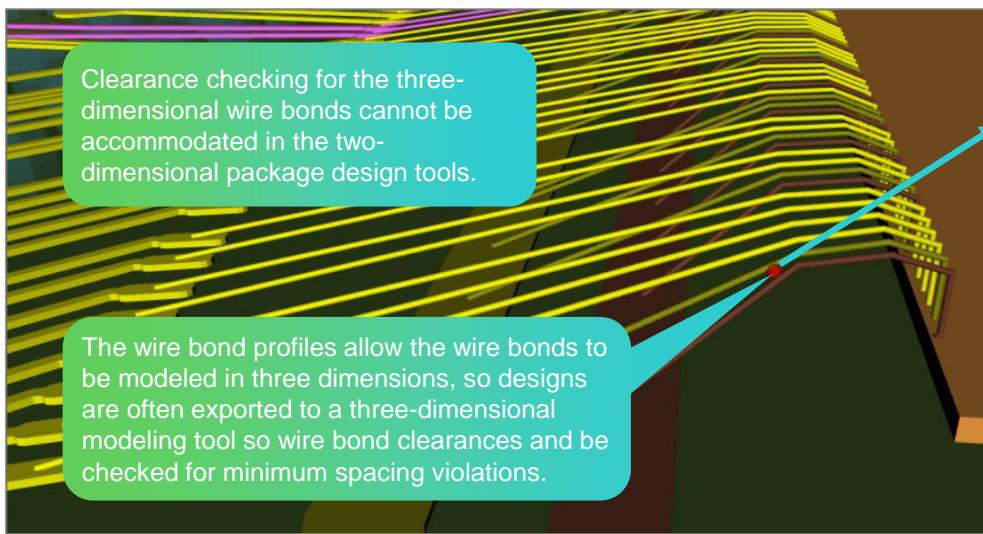
Wire bond profiles allow the package design tool to report accurate wire bond lengths.

744 © Cadence Design Systems, Inc. All rights reserved.

cadence®

To accommodate the three-dimensional wire bonds, the package design application allows you to create and use wire bond profiles. The wire bond profile models the three-dimensional nature of the wire bonds. A design may have many different wire profiles, and the profiles can be defined with an unlimited number of segments for accuracy. Many wire bonding machine vendors supply wire bond profiles that represent the wire bonds of their bonding equipment. Wire bond profiles allow the package design tool to report accurate wire bond lengths rather than the simple XY distance from the IC pin to the bond finger.

Wire Bond Clearance Checking

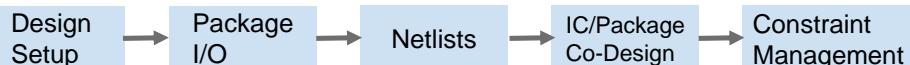
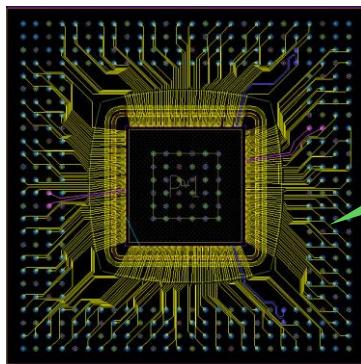


OBJECT: DRC_ERROR
LAYER: DRC_ERROR
RULE: 1
NAME: spacing90
CONSTRAINT: 60.00
VALUE: 51.00
INPUTS: [WIRE] [WIRE]
LAYERS: DRC_ERROR, PROFILE1

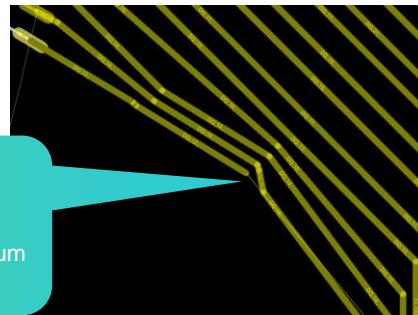
OBJECT: DRC_ERROR
LAYER: DRC_ERROR
RULE: 1
NAME: spacing90
CONSTRAINT: 60.00
VALUE: 51.00
INPUTS: [WIRE] [WIRE]
LAYERS: DRC_ERROR, PROFILE1

Clearance checking for the three-dimensional wire bonds can not be accommodated in the two-dimensional package design tools. The wire bond profiles allow the wire bonds to be models in three dimensions, so designs are often exported to a three-dimensional modeling tool so wire bond clearances can be checked for minimum spacing violations.

IC Package Design Flow: Routing



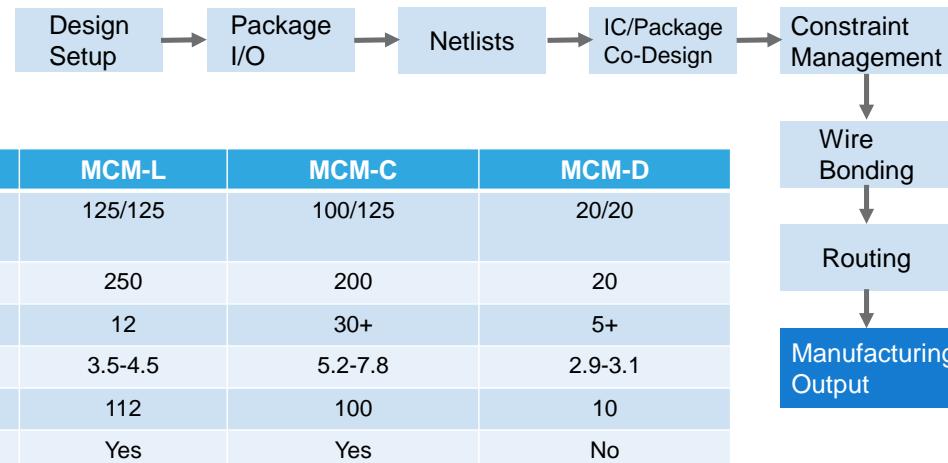
Each IC pin must be routed from the IC pins in the top center of the package to the package I/O pins, which are spread out across the bottom of the package.



Both the automatic router and the interactive routing environment will consult the constraint system and prevent you from creating design rule violations. They also push and shove existing traces to maintain minimum trace-to-trace spacing and optimize design real estate.

Each IC pin must be routed from the IC pins in the top center of the package to the package I/O pins, which are spread out across the bottom of the package. IC packaging autorouters are specialized routers that solve this type of radial routing problem. Most IC routers and PCB routers route in an orthogonal manner, trying to solve routing problems in the X and the Y direction rather than in the radial or diagonal direction. Both the automatic router and the interactive routing environment will consult the constraint system and prevent you from creating design rule violations. They also push and shove existing traces to maintain minimum trace-to-trace spacing and optimize design real estate.

Comparison of MCM Types



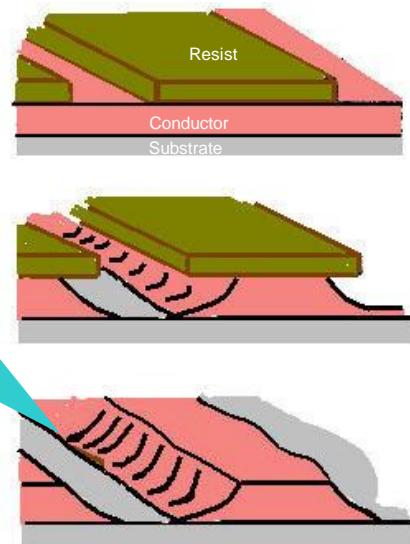
747 © Cadence Design Systems, Inc. All rights reserved.



Most packages can be broken down into three different manufacturing types. MCM-L uses a laminate process very similar to a printed circuit board type process. This process is generally the least expensive and most widely available because it is similar to the PCB process. The MCM-C process uses a ceramic substrate rather than a laminate substrate but generally gives more consistent electrical characteristics. The MCM-D process is done in silicon and is a deposited process similar to the IC manufacturing process. The MCM-D process provides the smallest features because it is similar to the IC manufacturing process.

MCM-L

The MCM-L process is a subtractive process. The chemical etching is not linear, and the resulting transmission lines are non-rectangular.

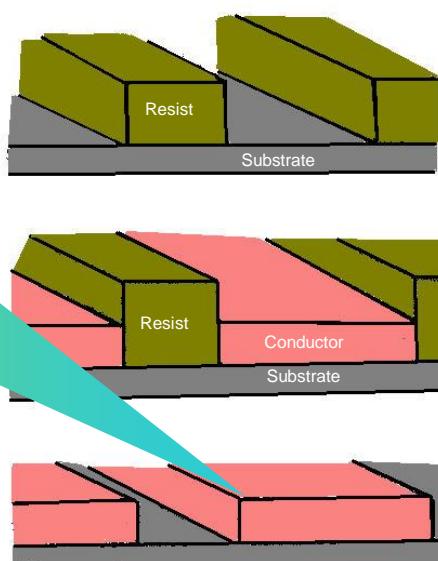


1. A layer of thick metal is deposited, and a pattern of photoresist is applied.
2. Unprotected metal is removed by an etching chemical.
3. The layer of photoresist is removed.

The MCM-L process is a subtractive process. The chemical etching is not linear, and the resulting transmission lines are non-rectangular. A layer of thick metal is deposited on a substrate, and then a pattern of photoresistive material is applied to the conductor that should remain. Unprotected metal is removed with an etching chemical, and the non-linear nature of the etching produces non-rectangular traces.

MCM-C

The MCM-C process is an additive process. Resist is applied to areas that should have no conductor, and when the resist is removed, the resulting traces are much more rectangular.



1. A thick layer of resist is applied and patterned to enable selective plating.

2. Unprotected areas are plated.

3. The layer of photoresist is removed, and the conductive traces remain.

The MCM-C process is an additive process. Resist is applied to areas that should have no conductor, and when the resist is removed, the resulting traces are much more rectangular. In the MCM-C manufacturing process, a thick layer of resist is applied with a mask to the areas of the ceramic substrate that should not be plated. The unprotected areas can then be plated, and when the layer of photoresist is removed, only the conductive traces remain. This additive process tends to give much more rectangular traces than the subtractive etching process.

MCM-D

The thin-film substrate, or MCM-D, technology emulates the photolithographic aspects of IC wafer fabrication. Light is used to transfer geometric patterns from a photomask to a light-sensitive chemical on a substrate with the following advantages:

- Very small feature sizes that result in high wiring density.
- High performance because of high-conductivity conductor materials.
- Metallization systems that can be tailored to accommodate either wire bonding or flip chip attachment.
- Low-temperature processing.



The thin-film substrate, or MCM-D, technology emulates the photolithographic aspects of IC wafer fabrication. Light is used to transfer geometric patterns from a photomask to a light-sensitive chemical on a substrate with the following advantages:

- Very small feature sizes that result in high wiring density.
- High performance because of high-conductivity conductor materials.
- Metallization systems that can be tailored to accommodate either wire bonding or flip chip attachment.
- Low-temperature processing.

Packaging Disciplines

IC packaging requires many different engineering disciplines. Engineers must solve problems in the electrical, mechanical, and materials domain:

Electrical

- Controlled impedance
- Crosstalk
- Reflection
- Radiation

Materials

- Dielectric control
- Conductor control
- Via formation
- Joining materials

Mechanical

- Interface stresses
- Warpage
- Corrosion
- Via cracking

751 © Cadence Design Systems, Inc. All rights reserved.



The different functions that the IC package provides often require input from many different engineering disciplines to design an effective IC package. Most important is the electrical performance, where controlled impedance is employed to reduce signal reflections and mitigate radiation. Clean return paths and transmission line spacings must be used to help reduce the chance of crosstalk problems. The mechanical disciplines are used to mitigate interface stresses that occur as the device heats and cools and make sure the warpage does not cause openings in the package connections.

What Is DC Voltage Drop?

Voltage for an IC is supplied by power pins at the bottom of the IC package, and it is carried through the package to the power pins of the IC through a conductor. The resistive losses in the conductor cause the voltage at the IC pins to be lower than the voltage supplied at the package pins, which is the DC voltage drop.



A clean DC voltage of 1.8 volts may be supplied at the power pins of the package.

Voltage for an IC is supplied by power pins at the bottom of the IC package, and it is carried through the package to the power pins of the IC through a conductor. The resistive losses in the conductor cause the voltage at the IC pins to be lower than the voltage supplied at the package pins, which is the DC voltage drop. Low supply voltages at the power pins can cause intermittent problems that are hard to find, as well as the complete failure of the device.

Causes of Voltage Drop

There are various causes of voltage drop. One of the main causes is the conductive properties of the conductor which include:

- **Type of material from which the conductor is made:** Copper conducts electricity better than aluminum and will cause less voltage drop than aluminum for a given length and conductor size.
- **Cross-sectional area of the conductor:** Conductors with a larger cross-sectional area will result in less voltage drop than conductors with a smaller cross-sectional area.
- **Conductor Length:** Shorter conductors will have less voltage drop than longer conductors for the same conductor cross-sectional area.
- **Temperature of the conductor:** Most conductive materials will increase their resistance with an increase in temperature.
- **Current being carried by the conductor:** Voltage drop increases on a conductor with an increase in the current flowing through the conductor.



There are various causes of voltage drop. One of the main causes is the conductive properties of the conductor which include:

- **Type of material from which the conductor is made:** Copper conducts electricity better than aluminum and will cause less voltage drop than aluminum for a given length and conductor size.
- **Cross-sectional area of the conductor:** Conductors with a larger cross-sectional area will result in less voltage drop than conductors with a smaller cross-sectional area.
- **Conductor Length:** Shorter conductors will have less voltage drop than longer conductors for the same conductor cross-sectional area.
- **Temperature of the conductor:** Most conductive materials will increase their resistance with an increase in temperature.
- **Current being carried by the conductor:** Voltage drop increases on a conductor with an increase in the current flowing through the conductor.

Calculating Voltage Drop

- DC resistance of a conductor is given by:

$$R_{DC} = \frac{\rho l}{A}$$

where ρ is the resistivity of the conductor in ohms/meter, l is the length in meters and A is the cross-sectional area in m^2 . The resistivity of copper is about 1.68×10^{-8} at room temperature. A typical PCB trace may be 35um thick and 3mm wide for an area of $105 \times 10^{-9}m$. If the trace is 100mm long, then the total DC resistance would be about 16 mOhms.

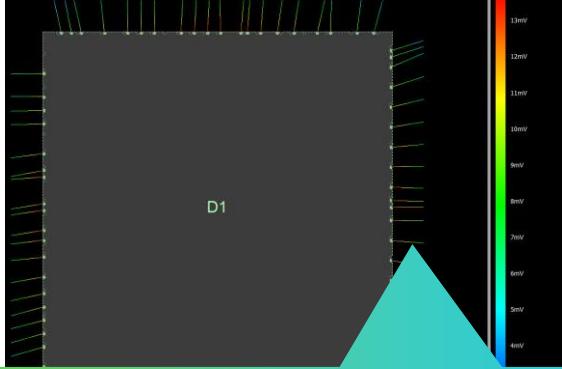
- Ohm's law, $V=IR$ can then be used, and if 100mA were flowing through the trace, then the voltage drop in the trace would be about 1.5mV.
- It is important to note that we did not account for the conductor resistivity change that would occur, which in turn would increase the voltage drop in the trace.
- The voltage drop across power and ground planes is dependent on its sheet resistance which for 1oz copper on the printed circuit board is approximately .5mOhms/square.



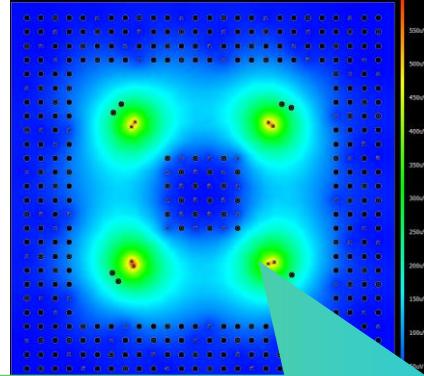
The DC resistance of a conductor is given by the resistivity of the conductor, in ohms per meter, times the length of the conductor in meters, divided by the cross-sectional area of the conductor in meters squared. The resistivity of copper is about 1.68 times 10 to the minus 8 at room temperature, and a typical PCB trace may be 35 microns thick and 33 millimeters wide for an area of 105 nanometers. If the trace is 100 millimeters long, then the total DC resistance would be about 16 milliohms. Ohm's law can then be used and if 100 milliamps were flowing through the trace, then the voltage drop in the trace would be about 1.5 millivolts. It is important to note that we did not account for the conductor resistivity change that would occur, which in turn would increase the voltage drop in the trace. The voltage drop across the power and ground planes is dependent on its sheet resistance which for 1-ounce copper on the printed circuit board is approximately .5 milliohms per square.

Analyzing Voltage Drop

Analyzing the voltage drop for a package by hand would be an impossible task. Analysis tools like PowerDC™ can take the physical design and based on the voltages on the rails, and the amount of current drawn by the IC, can show the voltage drop across the different objects in the package design.



Voltage drop through the wire bonds that attach the die pins to the package substrate.

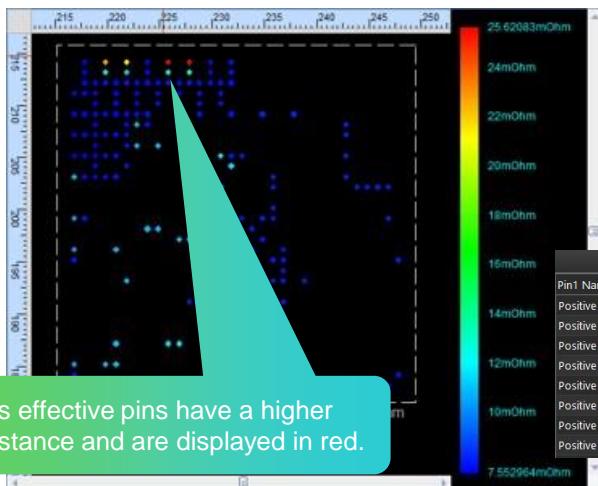


Just as the power planes see a voltage drop across them, the ground planes will see a voltage rise because of the resistivity of the conductor.

Analyzing the voltage drop for a package by hand would be an impossible task. Analysis tools like PowerDC can take the physical design and, based on the voltages on the rails and the amount of current drawn by the IC, can show the voltage drop across the different objects in the package design.

Analyzing Pin Effectiveness

The Pin Location Effectiveness analysis helps to quickly detect which pins are not effective by displaying high resistance from the package I/O voltage pins to each voltage pin of the IC. While simulating, designers may want to check which pins (power or ground) are ineffective because of the way routing or layout is done.



A sortable table view also allows the designer to identify the least effective pins.

Pin1 Name	Pin1 Net	Pin2 Name	Pin2 Net	Resistance (Ohm)
Positive Pin	VSS	Node146!99:VSS	VSS	0.0825936
Positive Pin	VSS	Node86!1:VSS	VSS	0.0820998
Positive Pin	VSS	Node147!105:VSS	VSS	0.0806635
Positive Pin	VSS	Node148!109:VSS	VSS	0.0805884
Positive Pin	VSS	Node149!113:VSS	VSS	0.0805619
Positive Pin	VSS	Node137!188:VSS	VSS	0.0799532
Positive Pin	VSS	Node92!129:VSS	VSS	0.0798586
Positive Pin	VSS	Node150!118:VSS	VSS	0.0797568

756 © Cadence Design Systems, Inc. All rights reserved.



The Pin Location Effectiveness analysis helps to quickly detect which pins are not effective by displaying high resistance from the package I/O voltage pins to each voltage pin of the IC. While simulating, designers may want to check which pins (power or ground) are ineffective. It is often a result of the way the routing or layout is done.

What Is Target Impedance?

Target impedance establishes a limit to the highest impedance the power rail on the die should see looking into the PDN. If the PDN impedance stays below this limit, even the worst-case transient current from the die will generate an acceptably low rail voltage noise.

$$Z_{\text{target}} = \frac{V_{\text{dd}} \times \text{tolerance}}{I_{\text{max}} - I_{\text{min}}}$$

For example:

$$\frac{1.2V \times 0.05}{7A - 2A} = 10 \text{ mOhm}$$



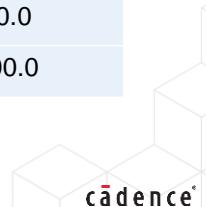
Target impedance establishes a limit to the highest impedance the power rail on the die should see looking into the PDN. If the PDN impedance stays below this limit, even the worst-case transient current from the die will generate an acceptably low rail voltage noise.

Target Impedance Chart

Target impedance is dropping by a factor of five for every computer generation.

From the table, you can conclude that each year's challenge is to create a power distribution system that maintains an ever-decreasing impedance for an ever-increasing frequency!

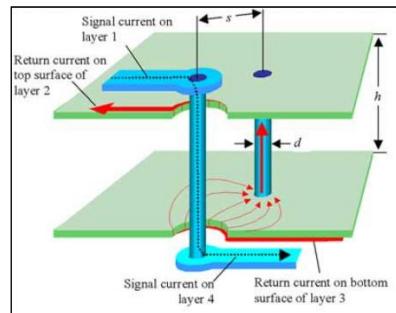
Voltage (volts)	Power Dissipated (watts)	Current (amps)	Ztarget (mOhms)	Frequency (MHz)
5.0	5.0	1.0	250	16.0
3.3	10	3.0	54.0	66
2.5	30.0	12.0	10	200.0
1.8	90.0	50.0	1.8	600.0
1.2	180.0	150.0	0.4	1200.0



The design of the Power Distribution System (PDS) is becoming an increasingly difficult challenge for modern CMOS technology. As CMOS technology is scaled to give smaller and faster transistors, the power supply voltage must decrease. As clock rates rise and more functions are integrated into microprocessors and application-specific integrated circuits (ASICs), the power consumed must increase. These trends are summarized in the table shown as we can see that even as the voltage decreases, the current is increasing at a faster rate, which in turn, increases the power dissipated and continues to lower the target impedance.

Power and Ground Plane Pairs

- Are key elements in power distribution system design.
- Are the only source of pure decoupling capacitance.
- Provide target impedance at frequencies too high for ceramic capacitors.
- Are modeled using SPICE-based simulation methods.

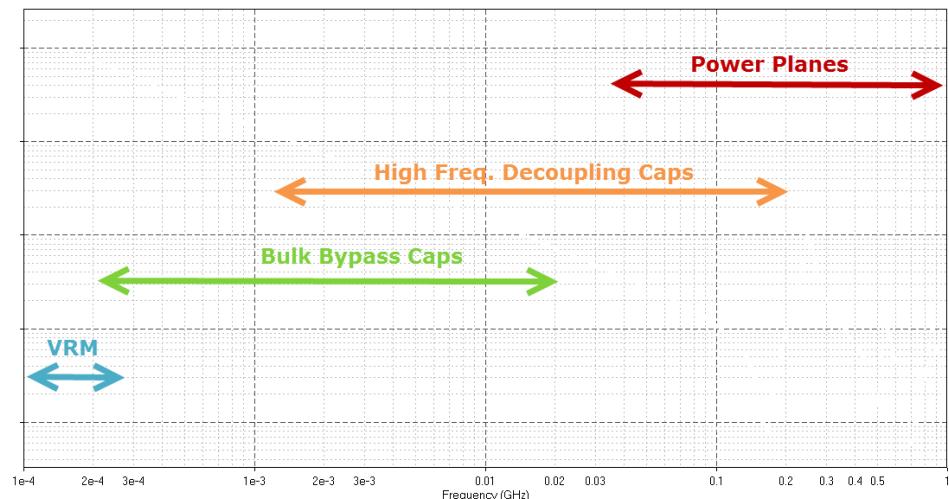


759 © Cadence Design Systems, Inc. All rights reserved.



Power planes are used to deliver power to both core logic and I/O circuits in modern computer systems. The amount of power required is ever-increasing with each computer generation. The current delivery requirements for the power planes have gone up greatly, and the tolerance for noise has gone down. The power delivery system is required to be low impedance. Power planes are capacitive at low frequencies, then develop resonances according to their cavity dimensions at high frequencies. The final components in the power distribution system are the power planes themselves. The efficiency of the power planes for high-speed decoupling is highly dependent on the characteristics of the printed circuit board stackup. This refers to the closeness of the conductive planes to each other, the thickness of the layers, the dielectric constant of the insulating layer, the shape and size of the planes, and the number of holes in the planes due to vias. The power plane layers do not provide significant decoupling capacitance below a few hundred megahertz. Above 400 MHz, the power planes provide the only significant decoupling at the printed circuit board level.

Keeping Impedance Low at All Frequencies



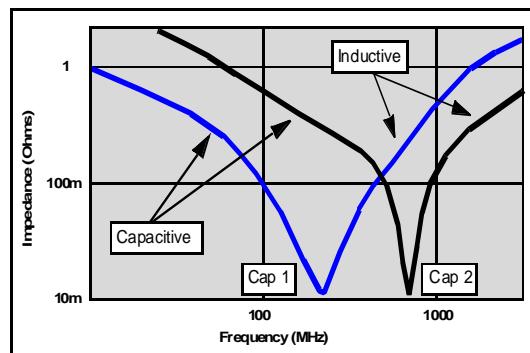
760 © Cadence Design Systems, Inc. All rights reserved.



In general, different parts of the power delivery system are responsible for keeping the impedance low across the entire frequency spectrum. The VRM maintains a low impedance at the low end of the spectrum but does not work at higher frequencies because of the relatively low switching speeds of the VRM. Bulk capacitors are responsible for maintaining the low impedance above frequencies the VRM is not capable of accommodating. As frequencies increase, the high-frequency decoupling capacitors become more important, and, finally, at the high end of the frequency spectrum, the power planes provide the low impedance required.

Viewing Impedance Profiles

The impedance profile for a capacitor comes down on a capacitive line. It bottoms out at the effective series resistance (ESR), then goes up on the line associated with the inductance. Cap 1 and Cap 2 have similar capacitive values, but Cap 2 has less inductance. Cap 2 has a higher Q because it is more peaked at the resonant frequency.



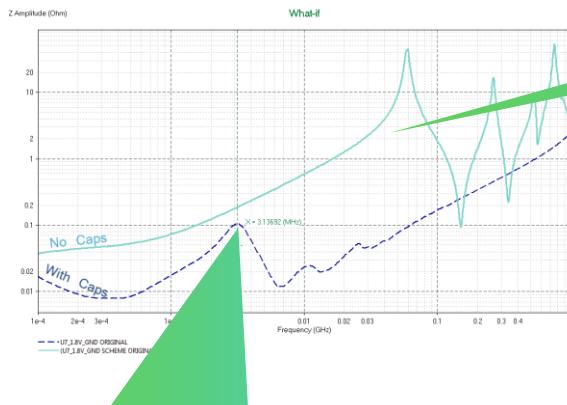
761 © Cadence Design Systems, Inc. All rights reserved.



This graphic shows impedance versus frequency for Cap 1 and Cap 2 (which has less inductance). Notice that the impedance minimum for the less inductive capacitor is at a much higher frequency. The impedance is reduced at all higher frequencies because of the low inductance. The effectiveness of high-frequency decoupling capacitors is greatly increased when the inductance is minimized.

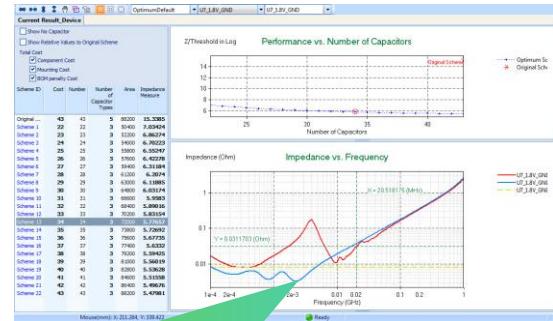
Effective series resistance (ESR) is a term that refers to resistive losses in a capacitor. This loss consists of the distributed plate resistance of the metal electrodes, the contact resistance between internal electrodes, and the external termination points. Note that the skin effect at high frequencies increases this resistive value in the leads of the component. Thus, the high-frequency ESR is higher than the DC ESR.

Turning Out Anti-Resonances



Adding capacitors by “rule of thumb” methods may leave frequencies where the impedance remains high.

With no capacitors, the impedance of the PDN is fairly high in the middle of the frequency spectrum.



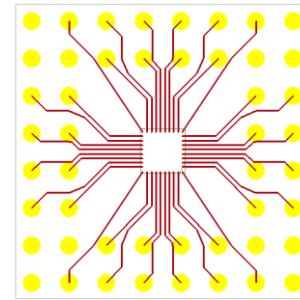
Capacitor optimization tools like OptimizePI™ run hundreds of simulations and provide an optimized set of capacitor values and placements.

With no capacitors, the impedance of the PDN is fairly high in the middle of the frequency spectrum. Many IC package designers use “rule of thumb” methods to add a selection of capacitors to the substrate to lower the impedance of the power delivery network. However, these rule-of-thumb methods often leave many frequencies with an anti-resonance point where the target impedance is not met. Capacitor optimization tools like OptimizePI run hundreds of simulations and provide an optimized set of capacitor values and placements.

What Are Transmission Lines In and IC Package?

In many electric circuits, the length of the wires connecting the components can mostly be ignored; that is, the voltage on the wire at a given time can be assumed to be the same at all points.

However, when the voltage changes in a time interval comparable to the time it takes for the signal to travel down the wire, the length becomes important, and the wire must be treated as a transmission line.



The two main characteristics of transmission lines are:

- Characteristic impedance
- Propagation delay

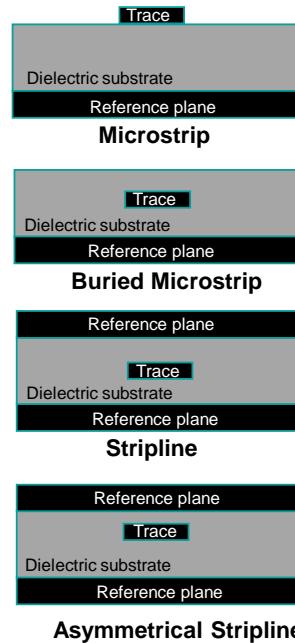
In many electric circuits, the length of the wires connecting the components can mostly be ignored; that is, the voltage on the wire at a given time can be assumed to be the same at all points. However, when the voltage changes in a time interval comparable to the time it takes for the signal to travel down the wire, the length becomes important, and the wire must be treated as a transmission line. The two main characteristics of transmission lines are Characteristic impedance and Propagation delay.

Board and Package Transmission Lines

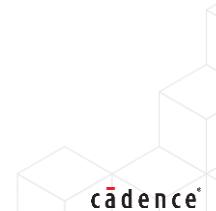
Characteristic impedance of a transmission line is the ratio of the amplitudes of a single pair of voltage and current waves propagating along the line in the absence of reflections.

For all these transmission lines, characteristic impedance is controlled by:

- Trace thickness
- Trace width
- The distance to the reference plane
- The dielectric constant of the insulating layer



764 © Cadence Design Systems, Inc. All rights reserved.

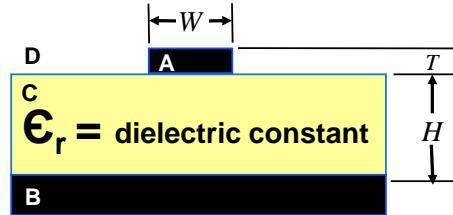


Characteristic impedance of a transmission line is the ratio of the amplitudes of a single pair of voltage and current waves propagating along the line in the absence of reflections. A microstrip trace is routed on the top or the bottom of the PCB or package with dielectric and a reference plane on one side of the trace. A buried microstrip is on an internal layer of the package or PCB with a dielectric material above and below the trace but still only has a reference plane on a single side. The stripline traces are embedded in the dielectric medium with a reference plane both above and below it. If the trace is equidistant to the two reference planes, then it is referred to as stripline, while the asymmetrical stripline trace is not equidistant to the two reference planes.

Microstrip Impedance

In a microstrip configuration, the conductor (A) is separated from the reference plane (B) by the dielectric substrate (C). Typically, the upper dielectric (D) is air.

$$Z_o = \frac{87}{\sqrt{\epsilon_r + 1.41}} \ln\left(\frac{5.98H}{0.8W + T}\right) \text{ Ohms}$$



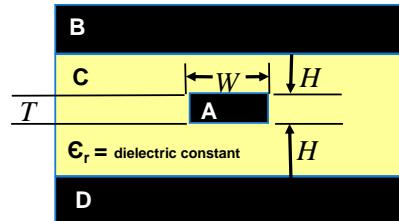
$$\text{when } 0.1 < \frac{W}{H} < 3.0, 1 < \epsilon_r < 15$$

There are empirical formulas for calculating the impedance of transmission lines on a PCB and IC package, and from the equations, we can see that as the thickness of the trace or the width of the trace increase, the impedance of the transmission line will decrease. Increasing the thickness of the dielectric, the distance to the reference plane will increase the impedance of the transmission line, while using a substrate with a lower dielectric constant will also increase the impedance of the transmission line.

Stripline Impedance

In a stripline configuration, the conductor (A) is sandwiched between the ground planes (B and D). The structure is supported by the dielectric (C). The diagram illustrates the dimensions of the stripline:

$$Z_o = \frac{60}{\sqrt{\epsilon_r}} \ln\left(\frac{1.9(2H+T)}{0.8W+T}\right) \text{ Ohms}$$



when $0.1 < \frac{W}{H} < 2.0, \frac{T}{H} < 0.25, 1 < \epsilon_r < 15$

The empirical formula for the stripline transmission line also shows that as the thickness of the trace or the width of the trace increase, the impedance of the transmission line will decrease. Increasing the thickness of the dielectric, the distance to the reference plane will increase the impedance of the transmission line, while using a substrate with a lower dielectric constant will also increase the impedance of the transmission line. The stripline has a reference plane both above and below the transmission line, which helps isolate it from crosstalk and emitting EMI.

Propagation Delay and Time Delay

- **Propagation delay:** The inverse of the propagation velocity, usually measured in seconds per meter.
- **Transmission-line time delay:** The amount of time it takes a signal to travel down the transmission line, that is, the propagation delay of the transmission line times the length of the transmission line.

$$\text{Propagation Delay} = \frac{1}{v} = \frac{\sqrt{\epsilon_r}}{c}$$

c = speed of light
 ϵ_r = dielectric constant

$$\text{Time Delay} = \frac{x\sqrt{\epsilon_r}}{c}$$

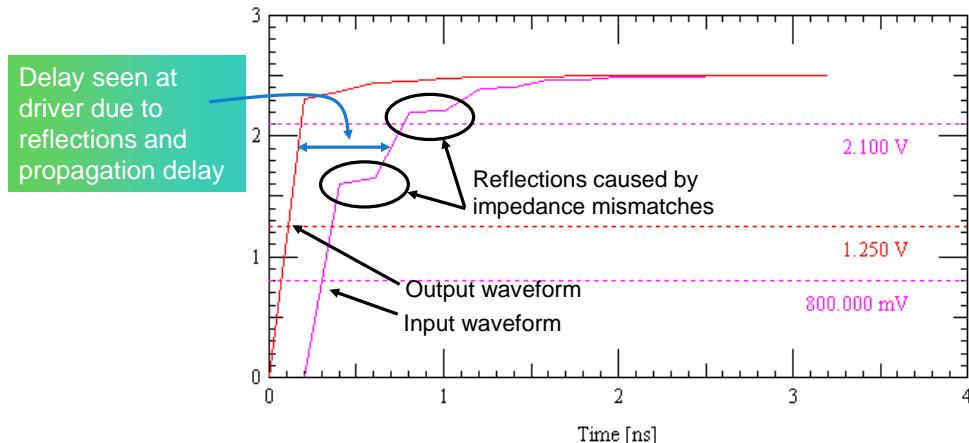
x = length of transmission line
 v = propagation velocity



The propagation velocity of a signal in a transmission line is determined by the dielectric constant of the material the transmission line is placed in, and the propagation delay is the inverse of this propagation velocity. The transmission line time delay, the amount of time it takes a signal to travel down the transmission line, is the propagation delay of the transmission line times the length of the transmission line.

Transmission Line Reflections

When a signal is being transmitted from one device to another, the propagation delay and impedance of the transmission line affect the integrity of the signal at the receiver.

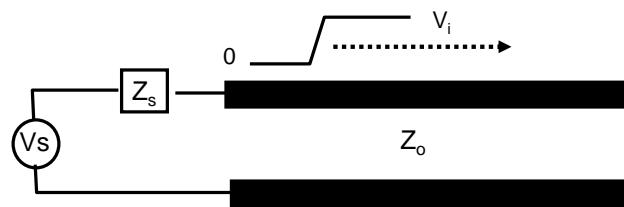


768 © Cadence Design Systems, Inc. All rights reserved.



When a signal is being transmitted from one device to another, the propagation delay and impedance of the transmission line affect the integrity of the signal at the receiver. Each time the impedance of the transmission line changes a portion of the signal is reflected back to the driver and only a portion of the signal continues on to the receiver. These reflections degrade the integrity of the signal and can cause incorrect information to be received at the receiver.

Transmission Line Reflections: Initial Stimulus

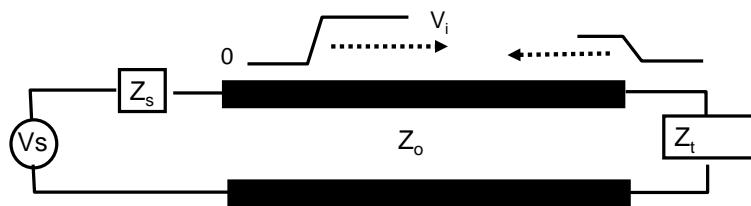


An initial driver signal is determined by a voltage divider of the driver impedance and the transmission line impedance.

$$V_i = V_s \frac{Z_o}{Z_o + Z_s}$$

The initial driver signal is determined by a voltage divider of the driver impedance and the transmission line impedance. As we will see later, reflections continue to reflect back and forth up and down the transmission line, and having a driver impedance that matches the transmission line impedance will prevent signal reflections from the receiver from being re-reflected back down the transmission line.

Transmission Line Reflections: Impedance Mismatches



- If the input impedance of the receiving device does not match the characteristic impedance of the transmission line, then a portion of the initial signal is reflected back on the transmission line toward the driver.
- When the input impedance of the receiving device matches the characteristic impedance of the transmission line, no portion of the signal is reflected, and the transmission line has a *matched termination*.

If the input impedance of the receiving device does not match the characteristic impedance of the transmission line, then a portion of the initial signal is reflected back on the transmission line toward the driver. When the input impedance of the receiving device matches the characteristic impedance of the transmission line, no portion of the signal is reflected, and the transmission line has a *matched termination*.

Transmission Line Reflections: Reflection Coefficient

The *reflection coefficient* is the ratio of the voltage reflected back to the incident voltage seen at an impedance discontinuity.

$$\rho = \frac{V_{reflected}}{V_{incident}} = \frac{Z_t - Z_o}{Z_t + Z_o}$$

Where:

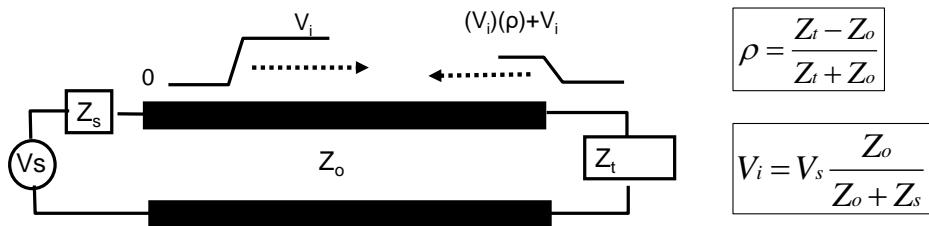
- Z_o is the characteristic impedance of a transmission line.
- Z_t is the impedance of the transmission line discontinuity.
- $V_{incident}$ is the initial voltage of the signal traveling on the transmission line.
- $V_{reflected}$ is the voltage reflected back on the transmission line because of the discontinuity.

If the input impedance of the receiver matches the impedance of the transmission line ($Z_t == Z_o$), then ρ is 0, and there is no voltage reflected back on the transmission line.



The *reflection coefficient* is the ratio of the voltage reflected back to the incident voltage seen at an impedance discontinuity. If the input impedance of the receiver matches the impedance of the transmission line, then the reflection coefficient is 0, and there is no voltage reflected back on the transmission line. Even if the receiver impedance matches the transmission line impedance, other impedance discontinuities, such as via transitions or connectors, will cause reflections back toward the driver.

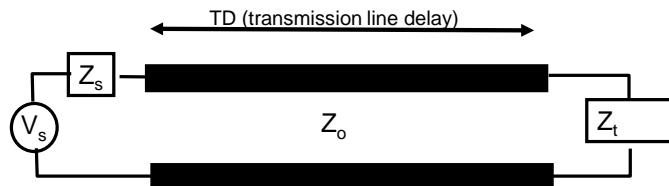
Transmission Line Reflections: First Reflection



- When the incident voltage, V_i , hits the termination with impedance Z_t , a portion of the signal $(V_i)\rho$ is reflected back toward the source and is added to the incident wave to produce a total magnitude of $(V_i)\rho + V_i$.
- When the reflected signal travels back to the source, if the source impedance does not match the line impedance, a second reflection is created toward the receiver. This creation of reflections continues until the line reaches a stable condition.

When an incident signal hits the receiver, if the receiver impedance does not match the transmission line impedance, a portion of the signal is reflected back toward the source and is added to the incident wave to produce a higher magnitude voltage. When the reflected signal travels back to the source, if the source impedance does not match the transmission line impedance, a second reflection is created back towards the receiver. This creation of reflections continues until the line reaches a stable condition.

Transmission Line Reflections: Multiple Reflections



At time $t = 0$	Source transitions to V_s . Initial voltage, V_i , at source is $(V_s)(Z_0)/(Z_0+R_s)$.
At time $t = TD$	Initial voltage, V_i , arrives at load. Reflection with magnitude, $(\rho_b)(V_i)$ is sent back to the source. Voltage at load is $(\rho_b)(V_i) + V_i$. (ρ_b is the reflection coefficient looking into the load.)
At time $t = 2TD$	First reflection $(\rho_b)(V_i)$ reaches the source. Reflection with magnitude, $(\rho_a)(\rho_b)(V_i)$ is sent to the load. Voltage at source is $V_i + (\rho_b)(V_i) + (\rho_a)(\rho_b)(V_i)$. (ρ_a is the reflection coefficient looking into the source.)

773 © Cadence Design Systems, Inc. All rights reserved.

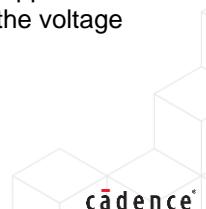


If TD is the transmission line delay, the time it takes the signal to travel down the transmission line, then at time 0, the voltage at the driver will be the initial voltage based on the voltage of the driver and a voltage divider created by the impedance difference between the driver and the transmission line. At time TD , the time it takes for the signal to travel the transmission line, the voltage at the receiver is the initial voltage plus the portion of the signal that will be reflected back to the driver due to a mismatch between the transmission line and the receiver. At time $2TD$, the reflection reaches the driver, and the voltage at the driver will be the initial voltage plus the amount of the signal that got reflected from the receiver. A portion of that reflected signal is then sent back down the transmission line and reaches the receiver at time $3TD$, and this continues until the signal reaches a steady state.

Lattice Diagrams

A lattice diagram is a technique used to solve the multiple reflections on a transmission line with linear loads. The lattice diagram on the next slide shows the following:

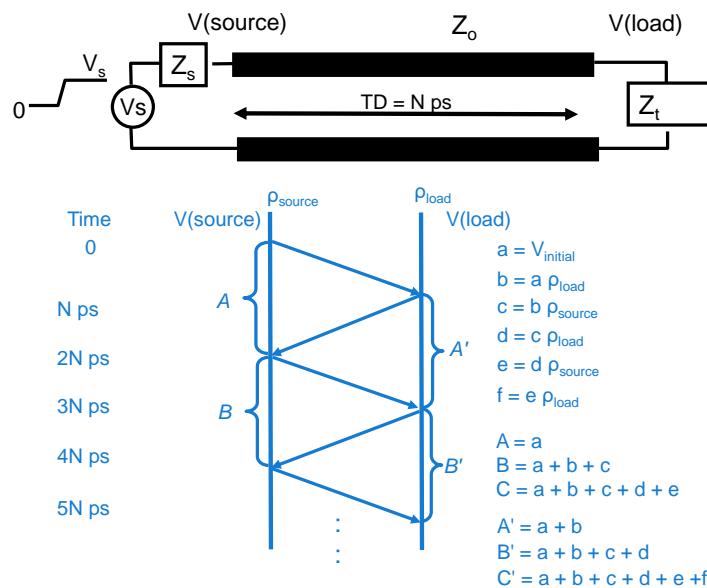
- The left and right vertical lines represent the source and load ends of the transmission line.
- The diagonal lines contained between the vertical lines represent the signal bouncing back and forth between the source and the load.
- The diagram progressing from top to bottom represents increasing time. The time increment is equal to the time delay of the transmission line.
- The vertical bars are labeled with reflection coefficients at the top of the diagram. These reflection coefficients represent the reflection between the transmission line and the load (looking into the load from the line) and the reflection coefficient looking into the source.
- The lowercase letters represent the magnitude of the reflected signal traveling on the line, the uppercase letters represent the voltages seen at the source, and the primed uppercase letters represent the voltage seen at the load end of the line.



A lattice diagram is a technique used to solve the multiple reflections on a transmission line with linear loads. The lattice diagram on the next slide shows the following:

- The left and right vertical lines represent the source and load ends of the transmission line.
- The diagonal lines contained between the vertical lines represent the signal bouncing back and forth between the source and the load.
- The diagram progressing from top to bottom represents increasing time. The time increment is equal to the time delay of the transmission line.
- The vertical bars are labeled with reflection coefficients at the top of the diagram. These reflection coefficients represent the reflection between the transmission line and the load (looking into the load from the line) and the reflection coefficient looking into the source.
- The lowercase letters represent the magnitude of the reflected signal traveling on the line, the uppercase letters represent the voltages seen at the source, and the primed uppercase letters represent the voltage seen at the load end of the line.

Lattice Diagrams (continued)



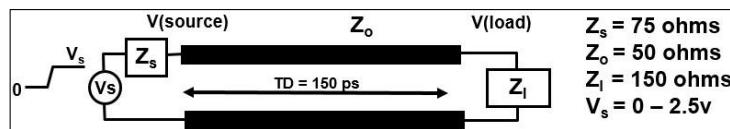
775 © Cadence Design Systems, Inc. All rights reserved.



A lattice diagram is a technique used to solve the multiple reflections on a transmission line with linear loads:

- The left- and right-hand vertical lines represent the source and load ends of the transmission line.
- The diagonal lines contained between the vertical lines represent the signal bouncing back and forth between the source and the load.
- The diagram progressing from top to bottom represents increasing time. The time increment is equal to the time delay of the transmission line.
- The vertical bars are labeled with reflection coefficients at the top of the diagram. These reflection coefficients represent the reflection between the transmission line and the load (looking into the load from the line) and the reflection coefficient looking into the source.
- The lowercase letters represent the magnitude of the reflected signal traveling on the line, the uppercase letters represent the voltages seen at the source, and the primed uppercase letters represent the voltage seen at the load end of the line.

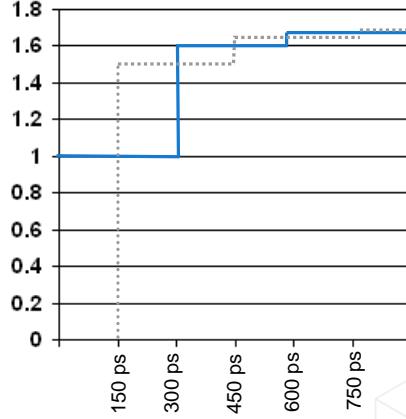
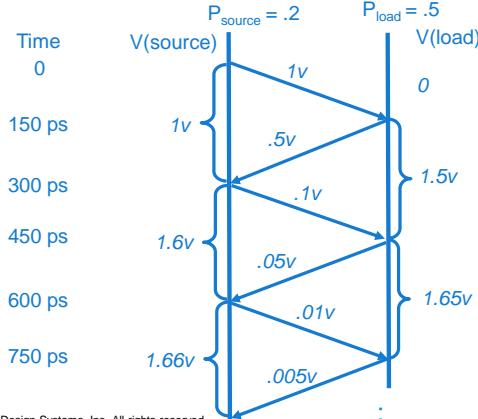
Example: Lattice Diagrams



$$V_{initial} = V_s \frac{Z_o}{Z_s + Z_o} = (2.5) \left(\frac{50}{75+50} \right) = 1$$

$$\rho_{source} = \frac{Z_s - Z_o}{Z_s + Z_o} = \frac{75 - 50}{75 + 50} = .2$$

$$\rho_{load} = \frac{Z_l - Z_o}{Z_l + Z_o} = \frac{150 - 50}{150 + 50} = .5$$



cadence®

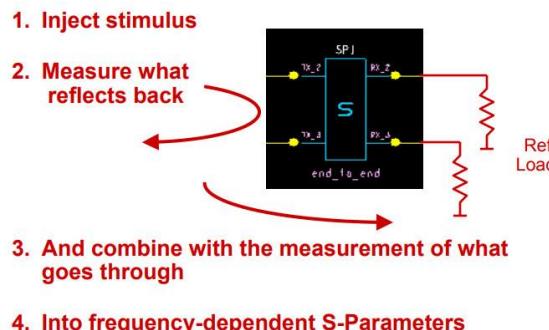
776 © Cadence Design Systems, Inc. All rights reserved.

In this example, the driver is 2.5 volts, but because the impedance of the driver does not match the impedance of the transmission line, the initial voltage at the driver is 1 volt while the initial voltage at the receiver is still 0 volts. In the graph, the solid line represents the voltage at the driver, and the dotted line represents the voltage at the receiver. The delay of our transmission line is 150 picoseconds, so at 150 picoseconds, the 1 volt reaches the receiver. The impedance discontinuity between the receiver and the transmission line is .5, so at 150 picoseconds, the voltage at the receiver is 1 volt plus the .5 volts that are now being reflected back to the driver. So, as shown in the graph at the right, at 150 picoseconds, the voltage at the receiver is 1.5 volts, and the voltage at the driver is 1 volt. The reflection takes another 150 picoseconds to get back to the driver, so at 300 picoseconds, the voltage at the driver is the initial 1 volt plus the .5 volt reflection. However, the driver has a reflection coefficient of .2, so one-fifth of the .5-volt reflection is reflected back toward the receiver. So, at 300 picoseconds, the voltage at the driver is the 1-volt initial signal, plus the .5 volts that were reflected back from the receiver, plus one-fifth of that original reflection, which is now being re-reflected back to the receiver for a voltage of 1.6 volts at the driver. That .2 volt reflection then reaches the receiver at 450 picoseconds, which is added to the voltage at the receiver plus .5 of the last reflection, which gets sent back to the driver. This continues as the reflections get smaller and smaller and the signal reaches a steady state.

What Are S-Parameters?

S-parameters, or scatter parameters, measure at each frequency the percent of energy that is seen at each port when a port is excited with a voltage and a current.

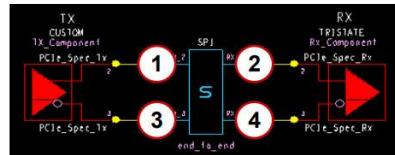
Calculating the reflections on an ideal transmission line is possible, but calculating all the reflections on a real transmission line is not. Each transition through a via, different trace widths, and different routing layers make it impossible to calculate all the reflections by hand. At high frequencies, these reflections can have a large effect on the operation of a circuit.



Calculating the reflections on an ideal transmission line is possible, but calculating all the reflections on a real transmission line is not. Each transition through a via, different trace widths, and different routing layers make it impossible to calculate all the reflections by hand. At high frequencies, these reflections can have a large effect on the operation of a circuit. S-parameters, or scatter parameters, measure at each frequency of the percent of energy that is seen at each port when a port is excited with a voltage and a current.

Can S-Parameters Be Used for Differential Pairs?

S-parameters can model differential pair transmission lines as well as single-ended transmission lines. S-parameters for single-ended traces have one port at the package I/O and one port at the die pin. Differential pair S-parameters have 4 ports, and the S-parameters also model the coupling or crosstalk of the transmission lines.



- On differential nets, nodes are typically numbered as shown
- “ S_{21} ” is what appears at **2** when stimulus is applied at **1**
- As such, for stimulus applied at **1** if you look at:
 - 1** = S_{11} = Reflection (often called “return loss”)
 - 2** = S_{21} = Transmission Loss (sometimes called “insertion loss”)
 - 3** = S_{31} = Near-end Crosstalk
 - 4** = S_{41} = Far-end Crosstalk

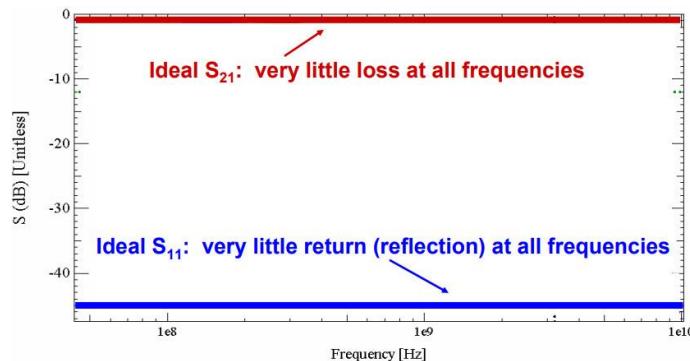
778 © Cadence Design Systems, Inc. All rights reserved.



S-parameters can model differential pair transmission lines as well as single-ended transmission lines. S-parameters for single-ended traces have one port at the package I/O and one port at the die pin. Differential pair S-parameters have 4 ports, and the S-parameters also model the coupling or crosstalk of the transmission lines.

S-Parameters for a Short Ideal Transmission Line

A short, ideal (and unrealistic) transmission line would have no loss from port 2 to port 1 across the entire frequency range; all of the energy would get from port 1 to port 2. Conversely, none of the energy input to port 1 would be reflected back to port 1, so there would be very little return loss across the entire frequency range.



- Remember:
 - S_{21} describes the end-to-end transmission
 - S_{11} describes what reflects back based on what you put in

A short, ideal (and unrealistic) transmission line would have no loss from port 2 to port 1 across the entire frequency range; all of the energy would get from port 1 to port 2. Conversely, none of the energy input to port 1 would be reflected back to port 1, so there would be very little return loss across the entire frequency range.

How Are S-Parameter Models Created?

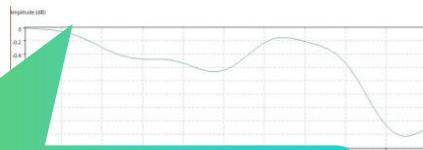
3D Field Solvers like the Clarity™ 3D Workbench are used to simulate and generate S-parameter models for traces and packages.

When a signal is attached to a port, a portion of that signal is reflected back to the port due to losses in the transmission mechanism. This is the return loss of the port.



Only a small portion of the energy is returned at low frequencies, but the return loss increases as the frequency increases.

When a signal is attached to a port, only a portion of the signal reaches the port on the other end due to losses in the transmission line. This is the insertion loss.

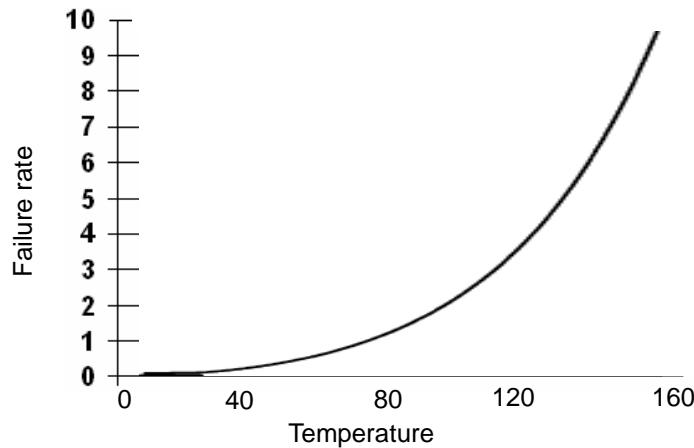


Nearly all of the energy is transferred from one port to the other at low frequencies, but the insertion loss increases as the frequency increases.

3D Field Solvers like the Clarity™ 3D Workbench are used to simulate and generate S-parameter models for traces and packages.

Why Is Thermal Management Important?

As the power requirements of ICs increase and the area of the silicon decreases, the need to dissipate heat away from the IC increases. Failure rates increase dramatically as operating temperatures increase.



781 © Cadence Design Systems, Inc. All rights reserved.



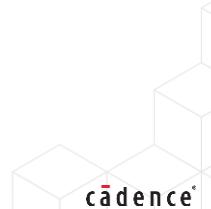
As the power requirements of ICs increase and the area of the silicon decreases, the need to dissipate heat away from the IC increases. Failure rates increase dramatically as operating temperatures increase.

Thermal-Transport Modes

Heat removal from the active regions of an IC might require the use of several mechanisms to transport the heat generated by the chip to the surrounding environment.

There are three basic thermal-transport modes:

- Conduction
- Convection
- Radiation



Heat removal from the active regions of an IC might require the use of several mechanisms to transport the heat generated by the chip to the surrounding environment.

There are three basic thermal-transport modes: Conduction, Convection, and Radiation.

What Is Thermal Conduction?

Thermal conduction is the spontaneous transfer of thermal energy through matter from a region of higher temperature to a region of lower temperature. Consequently, thermal conduction acts to balance temperature differences.



Conduction is governed by the Fourier equation: $q = -kA \frac{dT}{dx}$
where:

- q is the heat flow (W).
- k is the thermal conductivity (W/mK).
- A is the cross-sectional area for heat flow (m^2).
- dT/dx is the temperature gradient in the direction of heat flow (K/m).

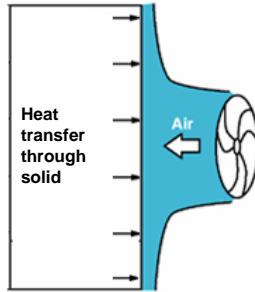
Thermal conduction is the spontaneous transfer of thermal energy through matter from a region of higher temperature to a region of lower temperature. Consequently, thermal conduction acts to balance temperature differences.

What Is Thermal Convection?

Convection is the transfer of heat from a solid to a fluid or gas in motion.

Heat transfer by convection includes two mechanisms:

- The exchange among nearly stationary molecules adjacent to the solid surface, as occurs in heat conduction.
- The transport of heat away from the solid surfaces by the bulk motion of the fluid or gas.



784 © Cadence Design Systems, Inc. All rights reserved.



Convection is the transfer of heat from a solid to a fluid or gas in motion. Heat transfer by convection includes two mechanisms. The exchange among nearly stationary molecules adjacent to the solid surface occurs in heat conduction and the transport of heat away from the solid surfaces by the bulk motion of the fluid or gas.

What Is Thermal Radiation?

Radiation heat transfer occurs as a result of the emission and absorption of the energy contained in electromagnetic waves or photons.

There are three main properties that characterize thermal radiation:

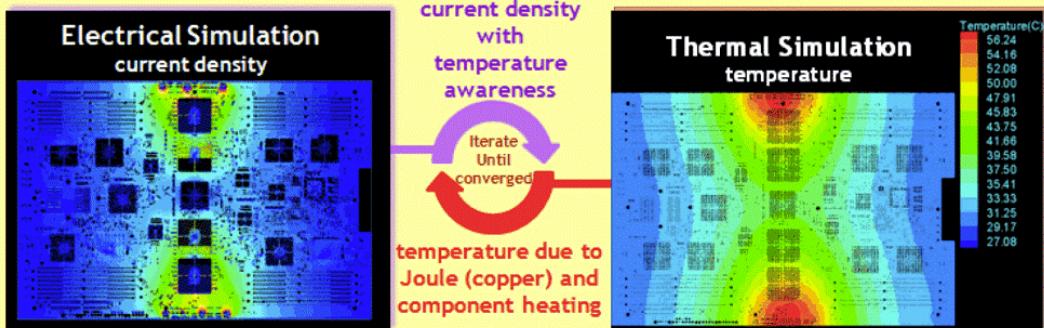
- Thermal radiation, even at a single temperature, occurs at a wide range of frequencies.
- The main frequency of the emitted radiation increases as the temperature increases.
- The total amount of radiation of all frequencies increases rapidly as the temperature rises.
(It grows as T^4 , where T is the absolute temperature of the body.)



Radiation heat transfer occurs as a result of the emission and absorption of the energy contained in electromagnetic waves or photons. There are three main properties that characterize thermal radiation. Thermal radiation, even at a single temperature, occurs at a wide range of frequencies. The main frequency of the emitted radiation increases as the temperature increases, and the total amount of radiation of all frequencies increases rapidly as the temperature rises. It grows as T to the fourth power, where T is the absolute temperature of the body.

Electrical and Thermal Co-Simulation

Celsius Thermal Solver

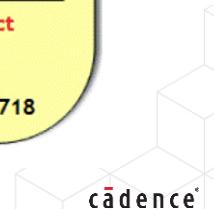


- Electrical resistance increases at higher temperatures
- Component leakage power dissipation increases at higher temperatures

Copper (Joule) heating will affect temperature distributions

Patent pending, US Application #13/158,718

786 © Cadence Design Systems, Inc. All rights reserved.



Electrical resistance and current distributions depend on the temperature. The temperature distributions, in turn, are affected by the heat generated by the current-carrying metal planes, metal traces, vias, solder balls, and solder bumps. The heat generated by the current-carrying metals is often referred to as the Joule heating or copper heating. In Celsius Thermal Solver, the Single-Board/Package and Multi-Board/Package E/T Co-Simulation workflows are provided to enable the electrical/thermal co-simulation function.

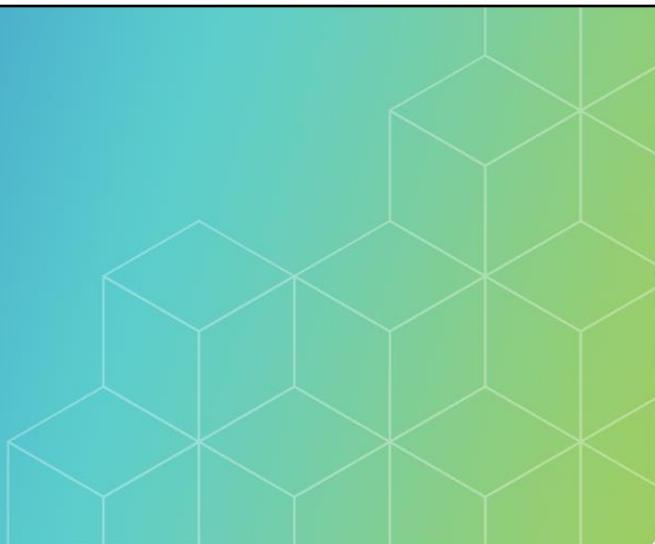
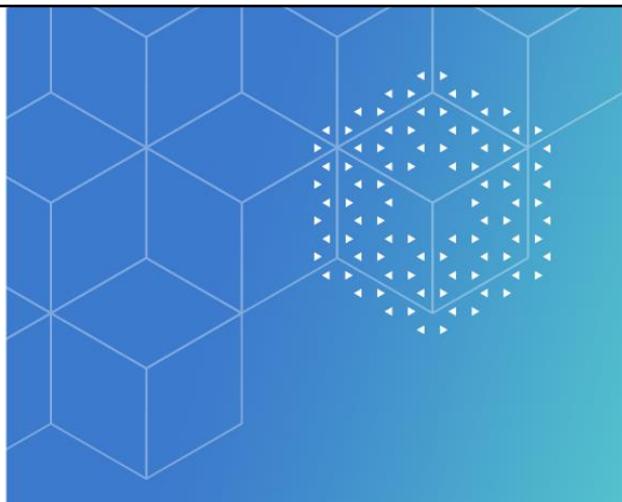
Module Summary

In this module, you:

- Identified the different IC packaging styles.
- Explained the different steps in the IC package implementation process.
- Described checking the PDN for DC voltage drop.
- Described checking the PDN for AC impedance across a frequency spectrum.
- Explored running a 3D-EM analysis to generate an S-parameter model.
- Discussed analyzing the thermal performance of an IC package.



In this module, we identified the different IC package styles and explained the different steps in the IC package implementation process. We described the checking of the Power Delivery Network for DC voltage drop as well as checking the PDN for AC impedance across a frequency spectrum. Finally, we explored running a 3D-EM analysis to generate an S-parameter model and discussed analyzing the thermal performance of an IC package.



Module 8

Next Steps

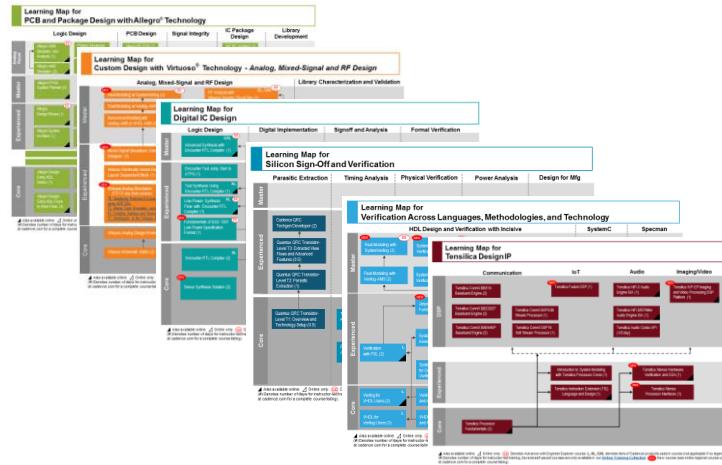
cadence®

This page does not contain notes.

Learning Maps

Cadence® Training Services learning maps provide a comprehensive visual overview of the learning opportunities for Cadence customers.

Click [here](#) to see all our courses in each technology area and the recommended order in which to take them.



789 © Cadence Design Systems, Inc. All rights reserved.



Go here to view the learning maps:

http://www.cadence.com/Training/Pages/learning_maps.aspx

Cadence Learning and Support

The screenshot shows the Cadence Learning and Support website. At the top, there's a navigation bar with links for Cases, Tools, IP, Resources, Learning, Software, My Support, and Contribute Content. To the right of the navigation are a bell icon and a user profile icon. The main header features the Cadence logo and the text "LEARNING & SUPPORT". Below the header is a search bar with the placeholder "Start your search here..." and a magnifying glass icon. There are also buttons for "View History" and "Documents Liked". A large play button icon is overlaid in the center of the page. The background has a dark, futuristic circuit board design. At the bottom, there's a row of six icons with labels: Installation & Licensing, Product Manuals, Training Courses, What's New, Troubleshooting Information, and Video Library.

Cadence Support now includes over 2000 product/language/methodology videos ("Training Bytes")!

790 © Cadence Design Systems, Inc. All rights reserved.



Click [here](#) to view the demo of COS.

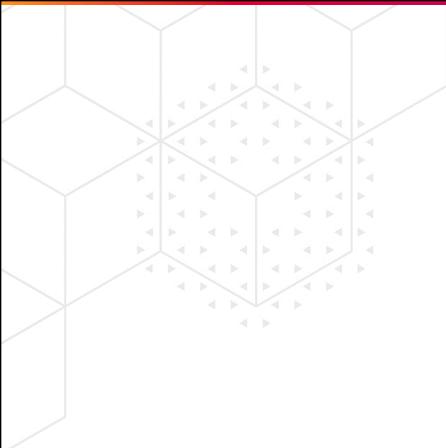
Wrap Up

- Complete Post Assessment, if provided
- Complete the Course Evaluation
- Get a Certificate of Course Completion

Thank you!



This page does not contain notes.



cadence®

© Cadence Design Systems, Inc. All rights reserved worldwide. Cadence, the Cadence logo, and the other Cadence marks found at <https://www.cadence.com/go/trademarks>, are trademarks or registered trademarks of Cadence Design Systems, Inc. Accellera and SystemC are trademarks of Accellera Systems Initiative Inc. All Arm products are registered trademarks or trademarks of Arm Limited (or its subsidiaries) in the US and/or elsewhere. All MIPI specifications are registered trademarks or service marks owned by MIPI Alliance. All PCI-SIG specifications are registered trademarks or trademarks of PCI-SIG. All other trademarks are the property of their respective owners.

This page does not contain notes.