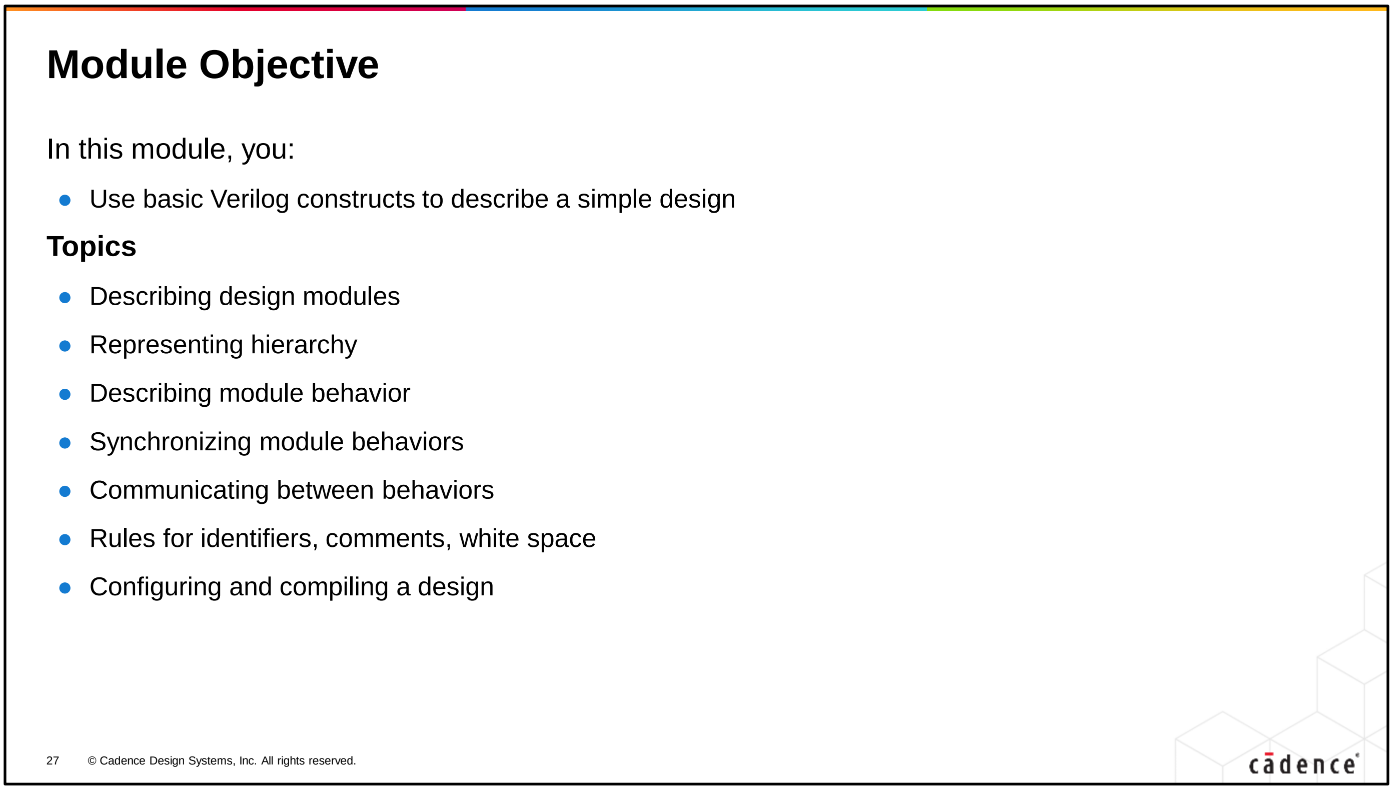
Module 3

Verilog Introduction

This module examines the fundamental language constructs and how you use them to describe a design.



Module Objective

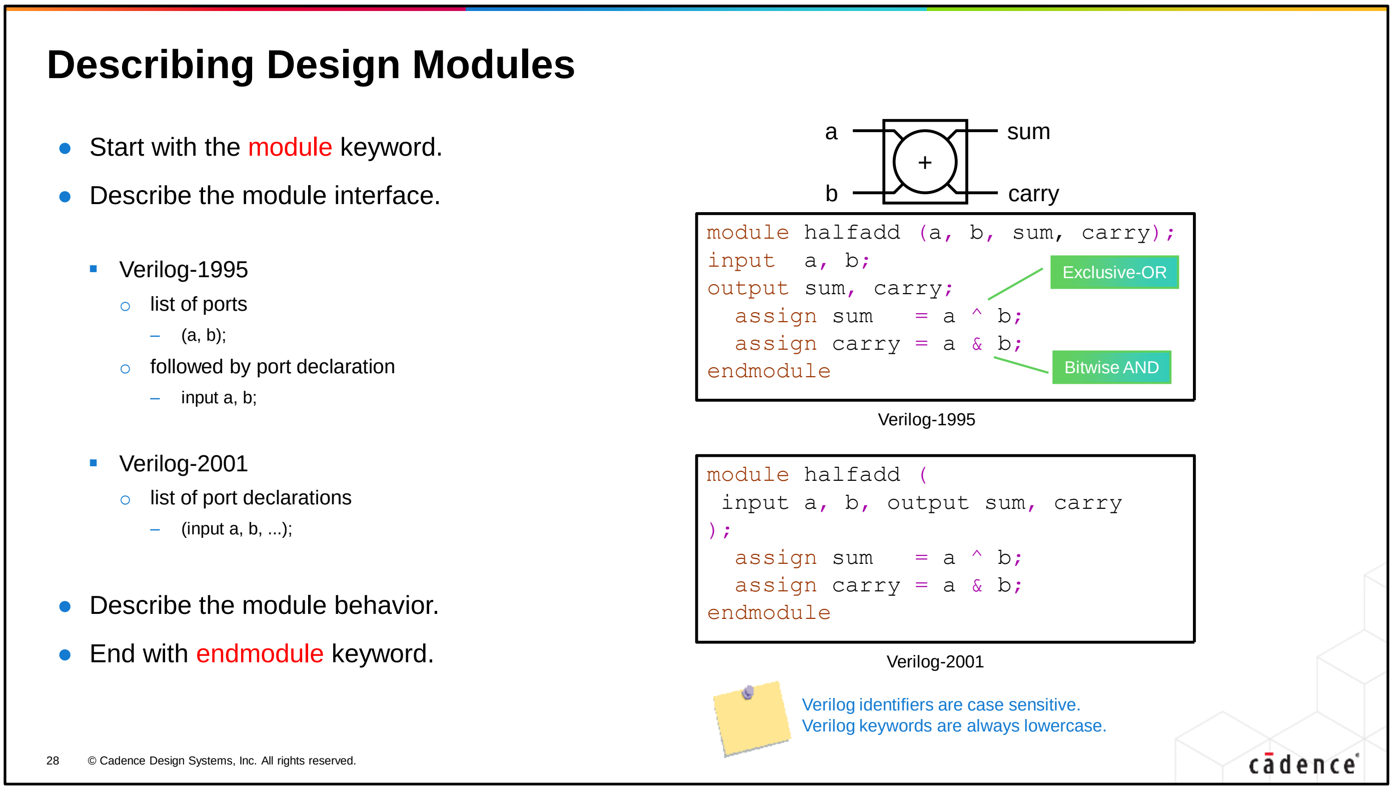
In this module, you:

- Use basic Verilog constructs to describe a simple design

Topics

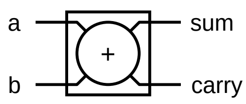
- Describing design modules
- Representing hierarchy
- Describing module behavior
- Synchronizing module behaviors
- Communicating between behaviors
- Rules for identifiers, comments, white space
- Configuring and compiling a design

Your objective is to get started using Verilog to describe the behavior of a digital design. To do that, you need to know some fundamental Verilog language constructs.



Describing Design Modules

- Start with the module keyword.
- Describe the module interface.
  - Verilog-1995
    - list of ports
      - (a, b);
    - followed by port declaration
      - input a, b;
  - Verilog-2001
    - list of port declarations
      - (input a, b, ...);
- Describe the module behavior.
- End with endmodule keyword.



```
module halfadd (a, b, sum, carry);
input  a, b;
output sum, carry;
  assign sum   = a ^ b;
  assign carry = a & b;
endmodule
```

Exclusive-OR

Bitwise AND

Verilog-1995

```
module halfadd (
 input a, b, output sum, carry
);
  assign sum   = a ^ b;
  assign carry = a & b;
endmodule
```

Verilog-2001

Verilog identifiers are case sensitive.
Verilog keywords are always lowercase.

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

cādence

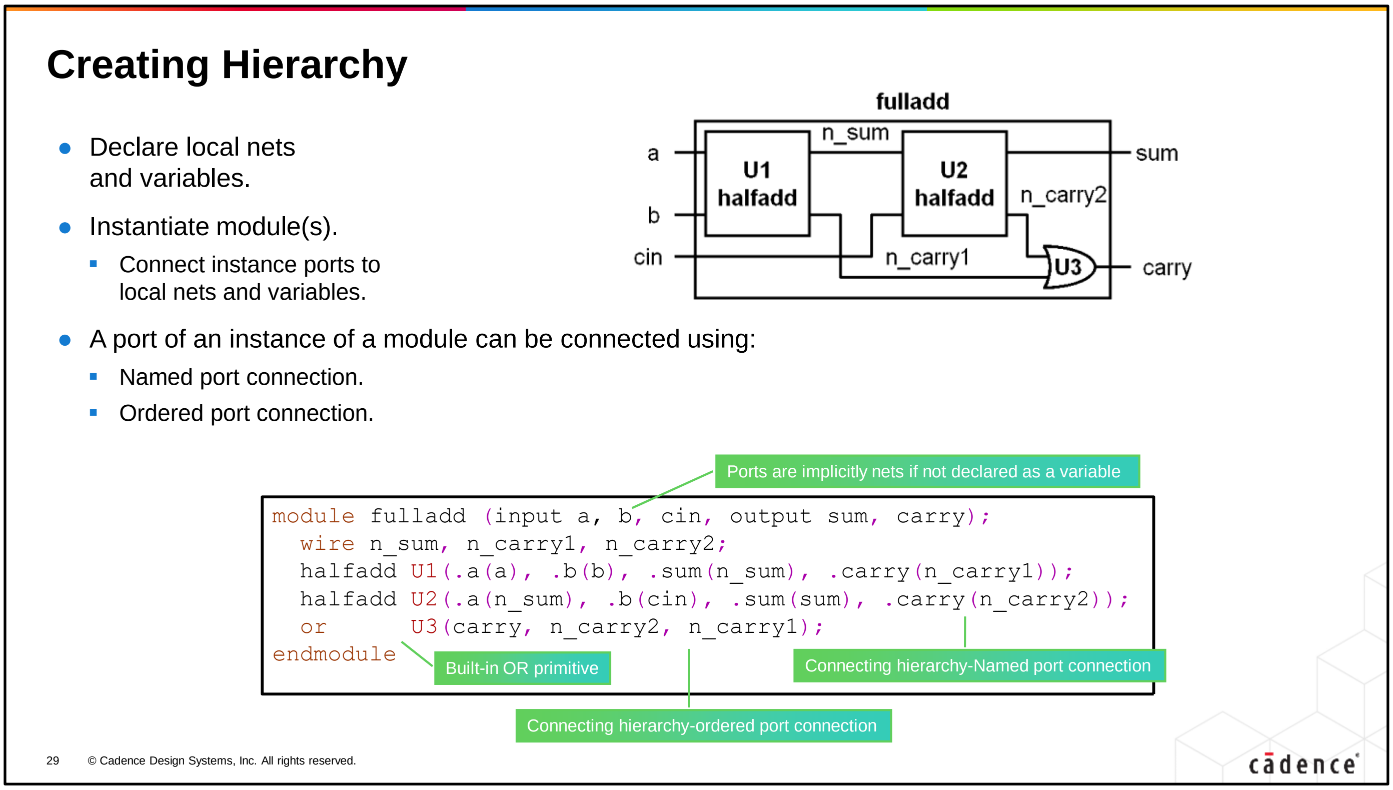
The basic building block of design hierarchy is the module declaration. A module can represent the complete system, its major subblocks such as the CPU, further subblocks such as the ALU, and physical blocks such as an ASIC cell. Each module instance is its own scope in which it can instantiate primitives, nets, variables and other modules.

Each module declaration starts with the *module* keyword followed by a unique definition name for the module. Most module declarations then follow with a list of ports or a list of port declarations. A module declaration typically 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***. The first example uses the Verilog-1995 list of ports syntax to list the ports ***a***, ***b***, ***sum*** and ***carry.*** It also declares the ports later as module items. The second example uses the Verilog-2001 syntax to list the port declarations. You can use either syntax for any module, but you cannot mix the two syntaxes in a single module declaration.

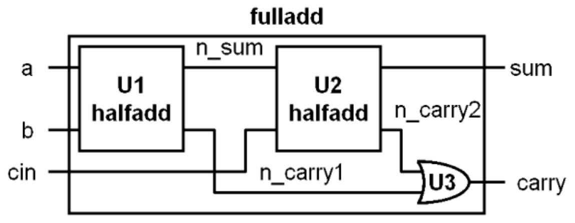
When you declare a port, you are explicitly or implicitly declaring a net or variable of that name and permitting the scope instantiating the module to make a connection to that net or variable. If you do not explicitly declare a net or variable, then a single-bit net is implicitly declared for the port.

The module describes the half-adder functionality by making two continuous assignment statements to the module output port nets. The statements assign the results of Boolean expressions. Continuous assignments are processes, and they react to changes of their inputs. Upon any change of the ***a*** or ***b*** input port net values, the simulator automatically updates the expression values and assigns the new values to the output port nets.



Creating Hierarchy

- Declare local nets and variables.
- Instantiate module(s).
  - Connect instance ports to local nets and variables.
- A port of an instance of a module can be connected using:
  - Named port connection.
  - Ordered port connection.



Ports are implicitly nets if not declared as a variable

```
module fulladd (input a, b, cin, output sum, carry);
  wire n_sum, n_carry1, n_carry2;
  halfadd U1(.a(a), .b(b), .sum(n_sum), .carry(n_carry1));
  halfadd U2(.a(n_sum), .b(cin), .sum(sum), .carry(n_carry2));
  or      U3(carry, n_carry2, n_carry1);
endmodule
```

Built-in OR primitive

Connecting hierarchy-Named port connection

Connecting hierarchy-ordered port connection

You create hierarchy by declaring ports, nets, variables and module instances and connecting ports of the module instances to the locally declared ports, nets, and variables.

You describe a full adder as a module definition that declares two instances of the half adder and an instance of an “or” built-in primitive and makes the appropriate connections.

This module first declares the nets it will use to connect the instance ports. You generally need to declare items before you reference them. Verilog implicitly declares single-bit nets for connections you make to module ports, so for this example you can elect to omit the net declarations.

You give each module an instance name (U1 and U2) that is unique within the module. You can use the instance names to uniquely reference each instance in the design.

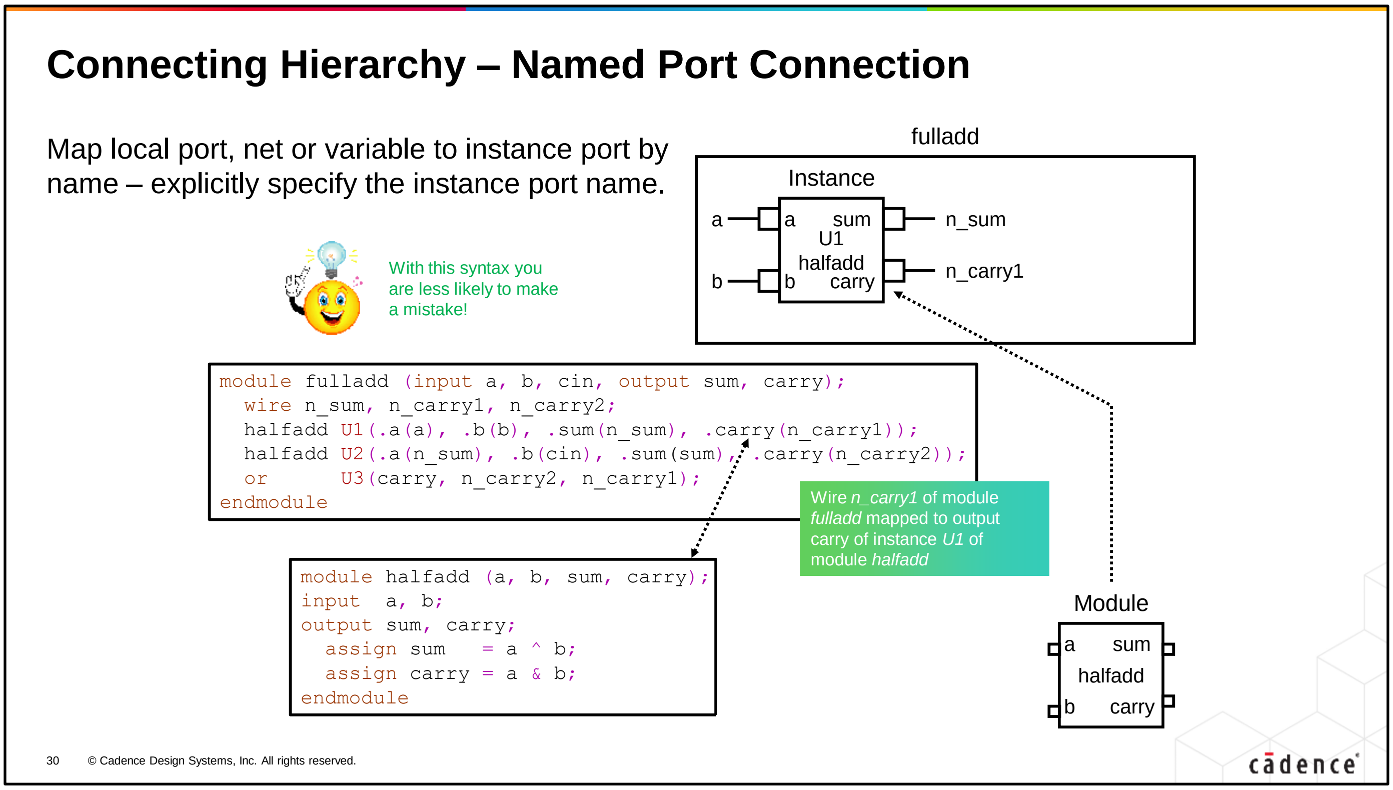
Verilog provides built-in primitives for basic logic functions. This example instantiates (U3) an “or” primitive. For this example you could have alternatively used a continuous assignment.

```
  assign carry = n_carry2 | n_carry1;
```

```
module_instantiation ::=
 module_identifier [ parameter_value_assignment ] module_instance { ,
module_instance } ;

module_instance ::= name_of_instance ([list_of_port_connections])

list_of_port_connections ::=
   ordered_port_connection { , ordered_port_connection }
 | named_port_connection { , named_port_connection }
```



Connecting Hierarchy – Named Port Connection

Map local port, net or variable to instance port by name – explicitly specify the instance port name.

With this syntax you are less likely to make a mistake!

fulladd
Instance
a
b
a
sum
U1
halfadd
b
carry
n_sum
n_carry1
Module
a
sum
halfadd
b
carry

```
module fulladd (input a, b, cin, output sum, carry);
  wire n_sum, n_carry1, n_carry2;
  halfadd U1(.a(a), .b(b), .sum(n_sum), .carry(n_carry1));
  halfadd U2(.a(n_sum), .b(cin), .sum(sum), .carry(n_carry2));
  or      U3(carry, n_carry2, n_carry1);
endmodule
```

Wire *n_carry1* of module *fulladd* mapped to output carry of instance *U1* of module *halfadd*

```
module halfadd (a, b, sum, carry);
input  a, b;
output sum, carry;
  assign sum   = a ^ b;
  assign carry = a & b;
endmodule
```

To connect a module instance’s port to locally declared ports, nets and variables, you use either the ordered port connection syntax or the named port connection syntax.

This example uses the named port connection syntax, in which you explicitly name the port to which you connect the expression. The expression is often simply a locally declared port, net or variable, but it can also be a concatenation of bit and part selections, and for input ports it can include other operators. You can omit any port connection by simply not naming the port.

Leaving an output port unconnected is common, e.g., for a device that has an active-low version and an active-high version of the same output signal. Leaving an input port unconnected is usually an error. Leaving an input port unconnected feeds a high-impedance value into the module or primitive, which almost all modules and primitives handle as an unknown logic value.

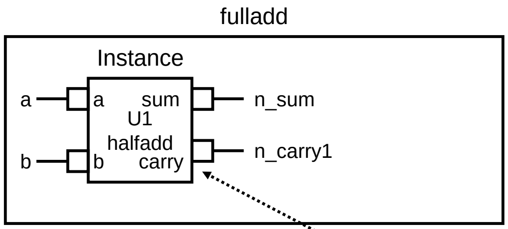
The named port connection syntax explicitly specifies the instance port for the connection. This syntax is verbose, but quickly readable and less likely to cause connection errors.

```
list_of_port_connections ::=
  ordered_port_connection { , ordered_port_connection }
 | named_port_connection { , named_port_connection }

named_port_connection ::=
 { attribute_instance } .port_identifier ( [ expression ] )
```

Connecting Hierarchy – Ordered Port Connection

Map local port, net or variable to instance port by position – in the order the ports are declared.

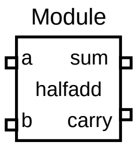


With this syntax you can very easily make a mistake!

```
module fulladd (input a, b, cin, output sum, carry);
  wire n_sum, n_carry1, n_carry2;
  halfadd U1(a, b, n_sum, n_carry1);
  halfadd U2(n_sum, cin, sum, n_carry2);
  or      U3(carry, n_carry2, n_carry1);
endmodule
```

Wire *n_carry1* of module *fulladd* mapped to output *carry* of instance *U1* of module *halfadd*

```
module halfadd (a, b, sum, carry);
input  a, b;
output sum, carry;
  assign sum   = a ^ b;
  assign carry = a & b;
endmodule
```



To connect a port of an instance of a module to locally declared ports, nets and variables, you use either the ordered port connection syntax or the named port connection syntax.

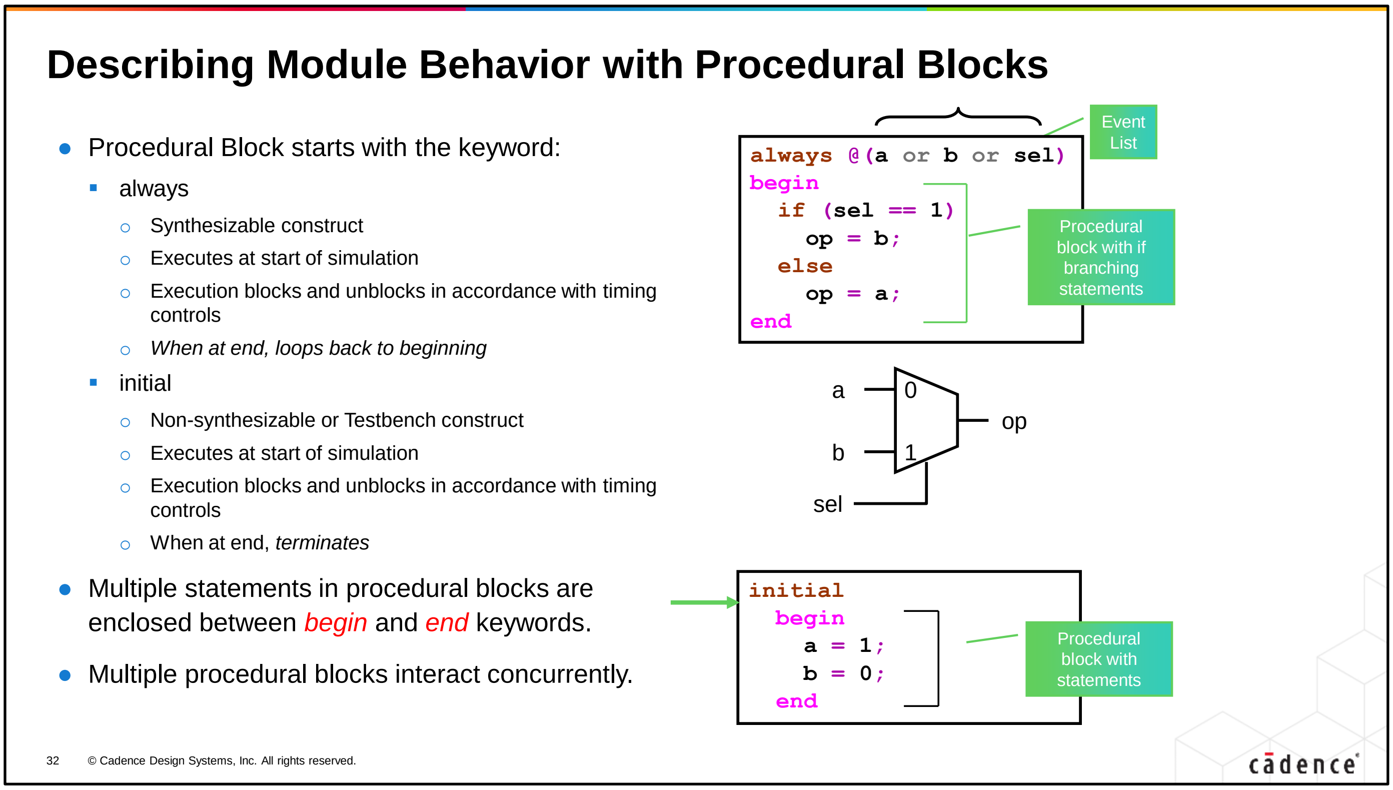
This example uses the ordered port connection syntax, in which you simply list the expressions connected to the instance ports in the same order that the instantiated module or primitive declares the ports. The expression is often simply a locally declared port, net or variable, but it can also be a concatenation of bit and part selections, and for input ports can include other operators. You can omit any port connection expression, but need to retain the comma as a place holder to make any later port connections.

Leaving an output port unconnected is common, e.g., for a device that has an active-low version and an active-high version of the same output signal. Leaving an input port unconnected is usually an error. Leaving an input port unconnected feeds a high-impedance value into the module or primitive, which almost all modules and primitives handle as an unknown logic value.

You and your coworkers may only with difficulty trace a signal path through ordered port connections, as the port ordering is not immediately obvious. You are also very likely to create incorrect connections. The more verbose named port connection syntax solves both problems.

```
list_of_port_connections ::=
  ordered_port_connection { , ordered_port_connection }
 | named_port_connection { , named_port_connection }

ordered_port_connection ::= { attribute_instance } [ expression ]
```



Describing Module Behavior with Procedural Blocks

- Procedural Block starts with the keyword:
  - always
    - Synthesizable construct
    - Executes at start of simulation
    - Execution blocks and unblocks in accordance with timing controls
    - *When at end, loops back to beginning*
  - initial
    - Non-synthesizable or Testbench construct
    - Executes at start of simulation
    - Execution blocks and unblocks in accordance with timing controls
    - When at end, *terminates*
- Multiple statements in procedural blocks are enclosed between *begin* and *end* keywords.
- Multiple procedural blocks interact concurrently.

```
always @(a or b or sel)
begin
  if (sel == 1)
    op = b;
  else
    op = a;
end
```

Event List

Procedural block with if branching statements

```
initial
  begin
    a = 1;
    b = 0;
  end
```

Procedural block with statements

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

Previous examples described the half-adder behavior by making continuous assignment statements.

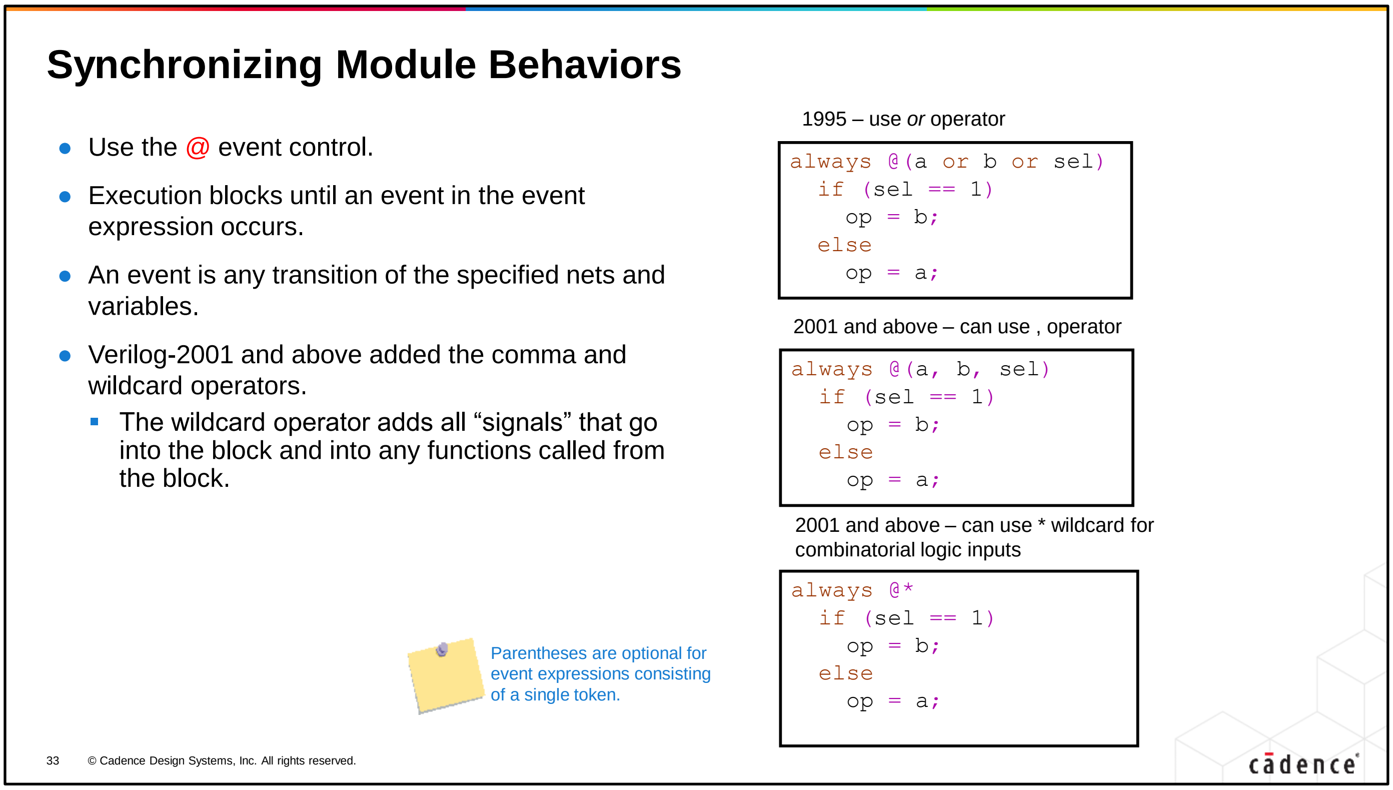
To describe complex behavior, Verilog provides procedural constructs. You place procedural constructs within a module declaration. Within the procedural constructs, you place procedural statements similar to the statements of a programming language.

Verilog provides two procedural blocks:

- The always construct starts executing at the start of simulation. Execution suspends and resumes in accordance with timing controls. Upon executing the last statement, execution loops back to the beginning of the construct. The always construct is ideal for describing hardware that always reacts to one or more of its inputs. The always construct always has at least one timing control to keep the construct from "hogging" all the execution cycles.
- The initial construct starts executing at the start of simulation. Execution suspends and resumes in accordance with timing controls. Upon executing the last statement, execution of that block terminates. The initial construct is ideal for describing a testbench that administers the test a finite, number of times, usually just once. The initial construct does not need any timing controls, but in a testbench at least one will likely have timing controls in order to "step" through the test.

Within a procedural block, you can choose to execute a group of statements sequentially in their order of appearance, as shown here, and you can choose to execute a group of statements in parallel in no particular order.

No execution order is implied between the procedural blocks. Procedural blocks scheduled to run in the same simulation cycle can execute in any order.



Synchronizing Module Behaviors

- Use the @ event control.
- Execution blocks until an event in the event expression occurs.
- An event is any transition of the specified nets and variables.
- Verilog-2001 and above added the comma and wildcard operators.
  - The wildcard operator adds all “signals” that go into the block and into any functions called from the block.

1995 – use *or* operator

```
always @(a or b or sel)
  if (sel == 1)
    op = b;
  else
    op = a;
```

2001 and above – can use , operator

```
always @(a, b, sel)
  if (sel == 1)
    op = b;
  else
    op = a;
```

2001 and above – can use * wildcard for combinatorial logic inputs

```
always @*
  if (sel == 1)
    op = b;
  else
    op = a;
```

Parentheses are optional for event expressions consisting of a single token.

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

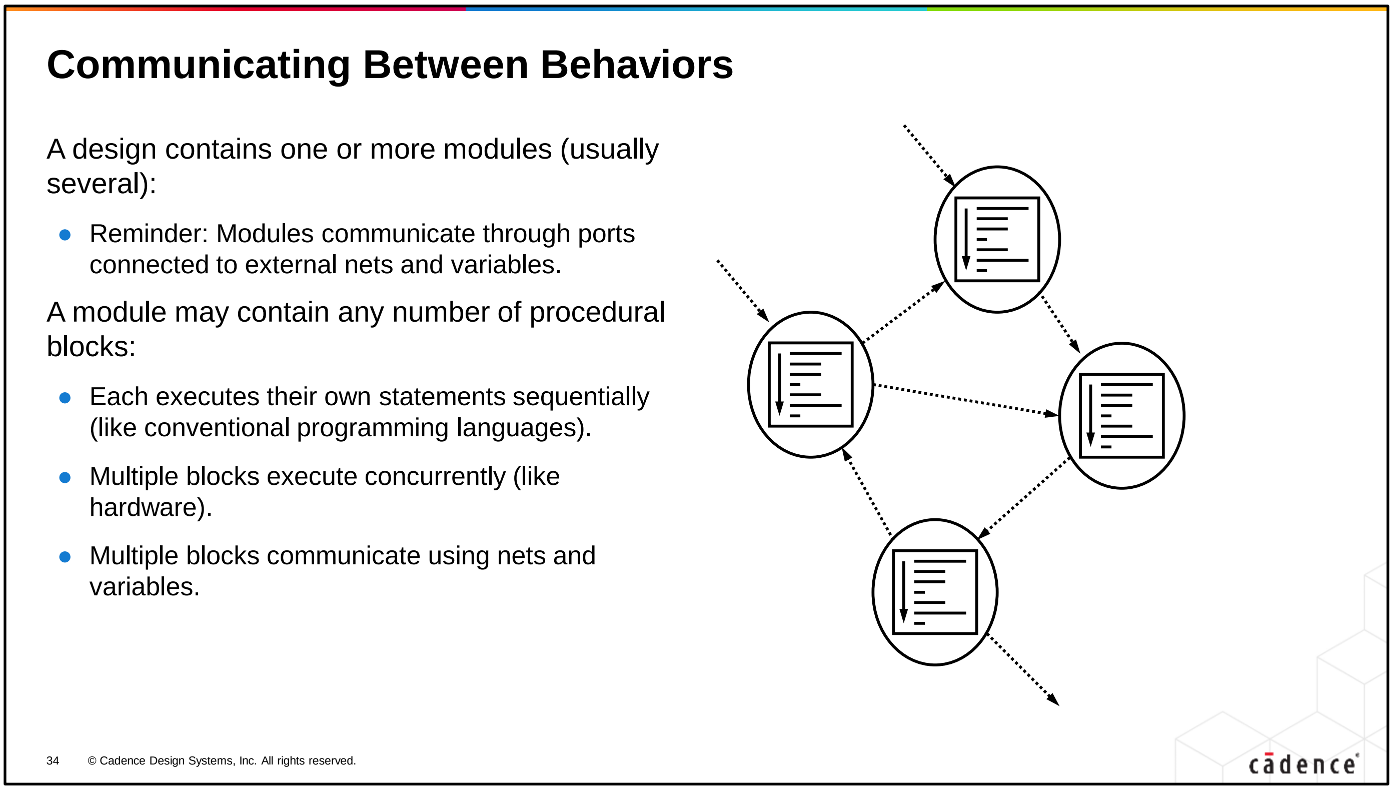
cādence

Verilog provides procedural timing controls for “stepping” execution of procedural blocks. The most common of these is the event control. An event control starts with the *at* (@) character 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 the *or* keyword or by the comma (**,**) character – both are shown here. The *or* keyword in an event expression is a separator between event expressions and is not an operator in the usual sense. You can further qualify an expression with the ***posedge*** or ***negedge*** keywords that you will see later.

A timing control is not itself a statement, but precedes a statement. In the case of an event control, it blocks the execution of that statement and subsequent sequential statements until one of those events occurs.

```
event_control ::=
  @ event_identifier
| @ ( event_expression )
| @*
| @ (*)

event_expression ::=
  expression
| hierarchical_identifier
| posedge expression
| negedge expression
| event_expression or event_expression
| event_expression , event_expression
```



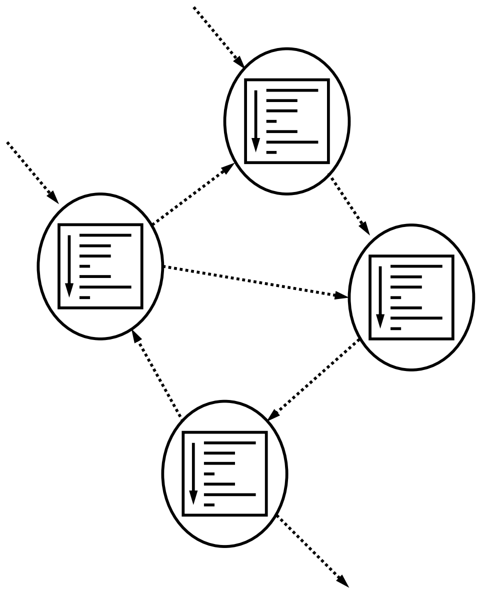
Communicating Between Behaviors

A design contains one or more modules (usually several):

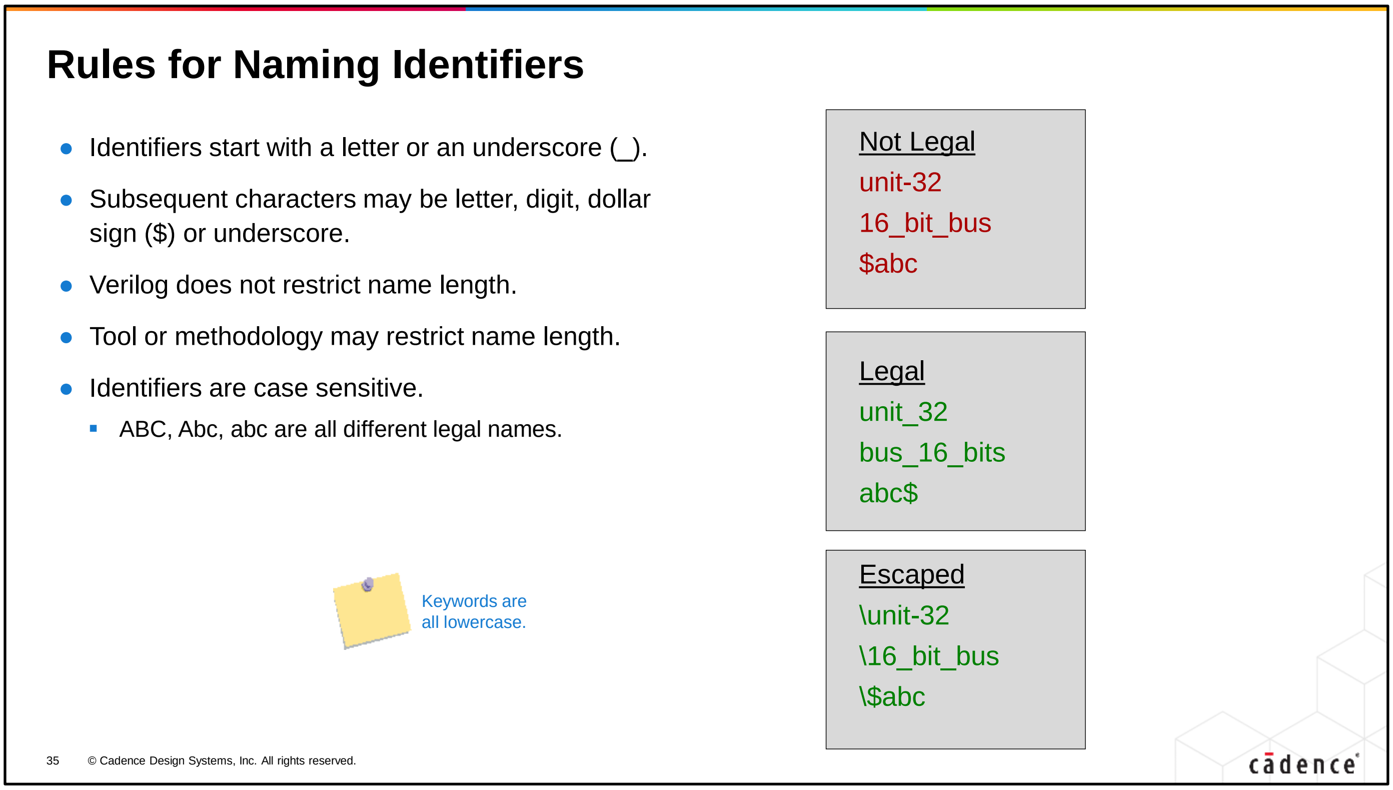
- Reminder: Modules communicate through ports connected to external nets and variables.

A module may contain any number of procedural blocks:

- Each executes their own statements sequentially (like conventional programming languages).
- Multiple blocks execute concurrently (like hardware).
- Multiple blocks communicate using nets and variables.



This capability to have many procedural blocks communicating concurrently with each other through variables is the basic model which Verilog uses to describe hardware. The procedural blocks can be in a single module or partitioned into a number of modules in a hierarchical structure. However, the basic model remains the same.



Rules for Naming Identifiers

- Identifiers start with a letter or an underscore (_).
- Subsequent characters may be letter, digit, dollar sign ($) or underscore.
- Verilog does not restrict name length.
- Tool or methodology may restrict name length.
- Identifiers are case sensitive.
  - ABC, Abc, abc are all different legal names.

Not Legal
unit-32
16_bit_bus
$abc

Legal
unit_32
bus_16_bits
abc$

Escaped
\unit-32
\16_bit_bus
\$abc

Keywords are all lowercase.

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

cadence

An identifier is a name you provide for Verilog objects and instances.

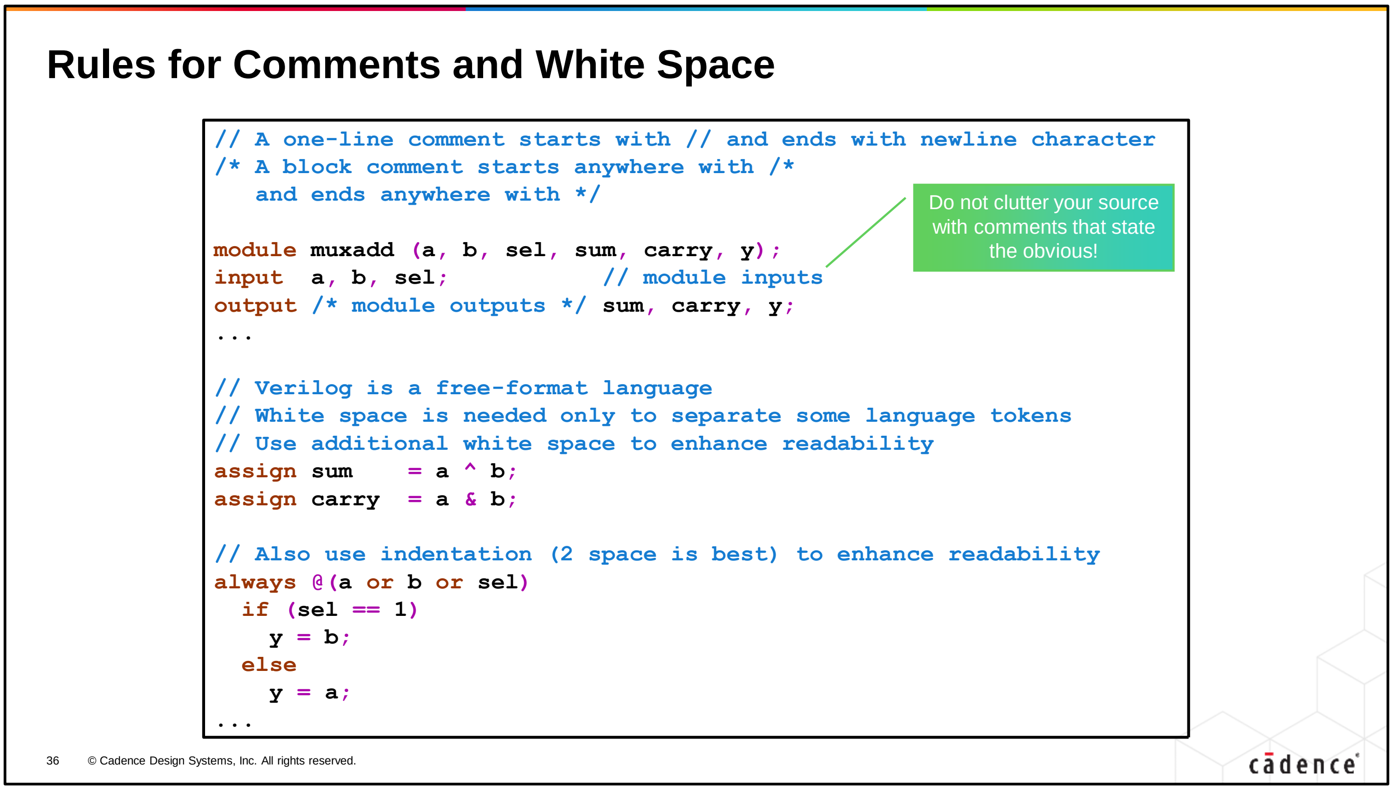
You must start an identifier with an alphabetical character (a-z, A-Z) or an underscore (_) character. Your later characters can include any alphanumeric character, the dollar sign ($) character, and the underscore character.

You will later see that built-in and user-defined system tasks and system functions start with a dollar sign character. These are not user identifiers in the normal sense.

The Verilog 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 lower case. The case of your identifier characters is exactly as you specify. You can declare two identifiers that differ only in case, and you can declare an identifier that differs from a keyword only in the case of one or more of its characters. These are both extremely poor practice.

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 exist to provide compatibility with tools and technology libraries. Some tools, for example, break vector variables into scalar variables that include their bracketed index in their name. The name must be escaped to allow the brackets. Some libraries, for example, start cell names with a digit to indicate the number of inputs. The name must be escaped to allow the initial digit. You should in general not declare escaped identifiers yourself because they make your code less readable and there is seldom good reason to do so.



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 (a, b, sel, sum, carry, y);
input  a, b, sel;            // module inputs
output /* module outputs */ sum, carry, y;
...

// Verilog 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 or b or sel)
  if (sel == 1)
    y = b;
  else
    y = a;
...
```

Do not clutter your source with comments that state the obvious!

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

Verilog is a free-format language. You can use white space to organize the code to enhance readability. Verilog ignores these characters except where needed to separate other language tokens.

Verilog has single-line comments and block comments:

- Single-line comments start with two consecutive 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 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.

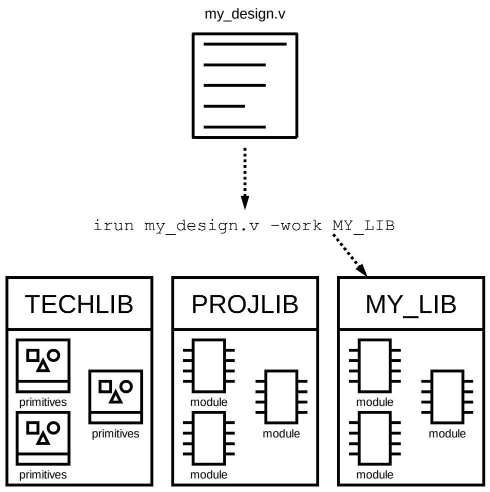
You should in general not clutter your source with unneeded comments. Describe your module in a comment header at the top of the file and otherwise use comments only where absolutely necessary to clarify obscure fragments of your code.

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.

Using Design Libraries

Some Verilog tools use compilation libraries:

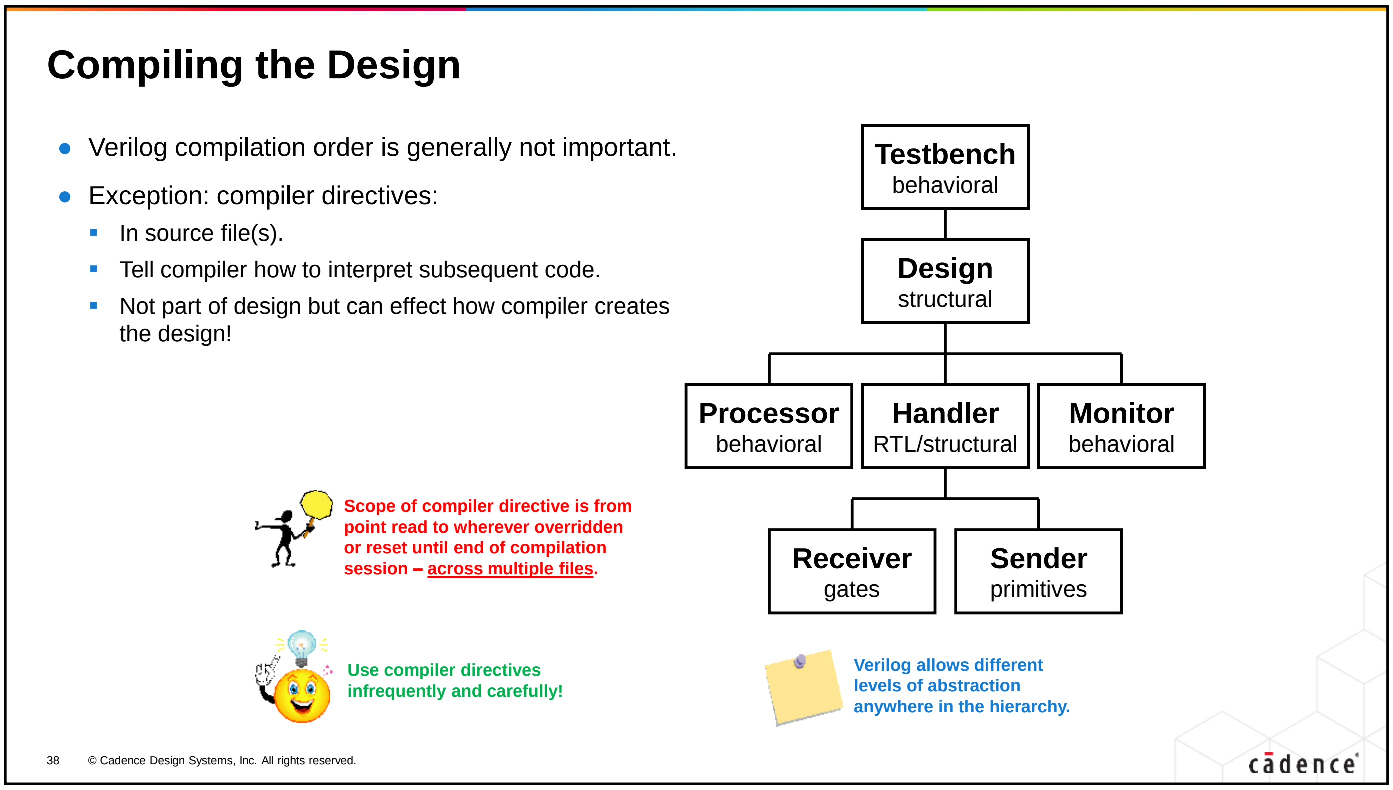
- A collection of already-compiled modules or primitives.
- Stored in file system.
- Referred to by library name.
- Library map file maps name to location.
  - Library name -> directory name
- Compiler places output in “WORK” library.
  - “WORK” library specified by user for each compilation session.
- Configuration in Verilog provides a prioritized list of libraries from which to bind compiled cell descriptions to cell instances.



Not all Verilog simulators use compilation libraries.

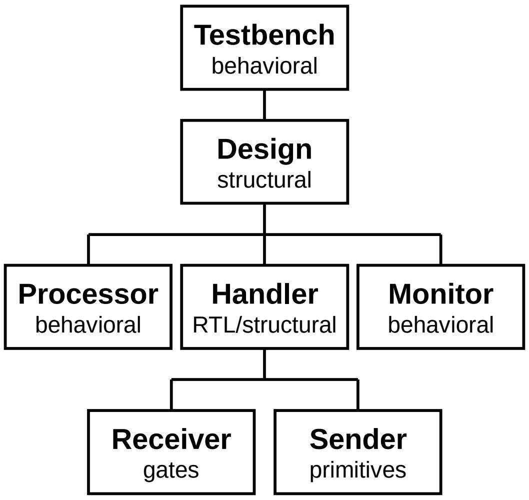
Simulating an HDL design and testbench is a process that first converts the source files into a binary form that the elaborator and simulator can recognize. This process of checking the syntax and producing the binary file is known as ***compilation***.

Verilog tools that use libraries compile a design into a file system data structure known as a library. A vendor-specific mechanism selects one of those libraries as the ***current working library***, and compilation by default places the results of compilation into that library. The elaboration process later selects design and testbench components from the libraries to construct the design and test configuration for simulation.



Compiling the Design

- Verilog compilation order is generally not important.
- Exception: compiler directives:
  - In source file(s).
  - Tell compiler how to interpret subsequent code.
  - Not part of design but can effect how compiler creates the design!



Scope of compiler directive is from point read to wherever overridden or reset until end of compilation session – across multiple files.

Use compiler directives infrequently and carefully!

Verilog allows different levels of abstraction anywhere in the hierarchy.

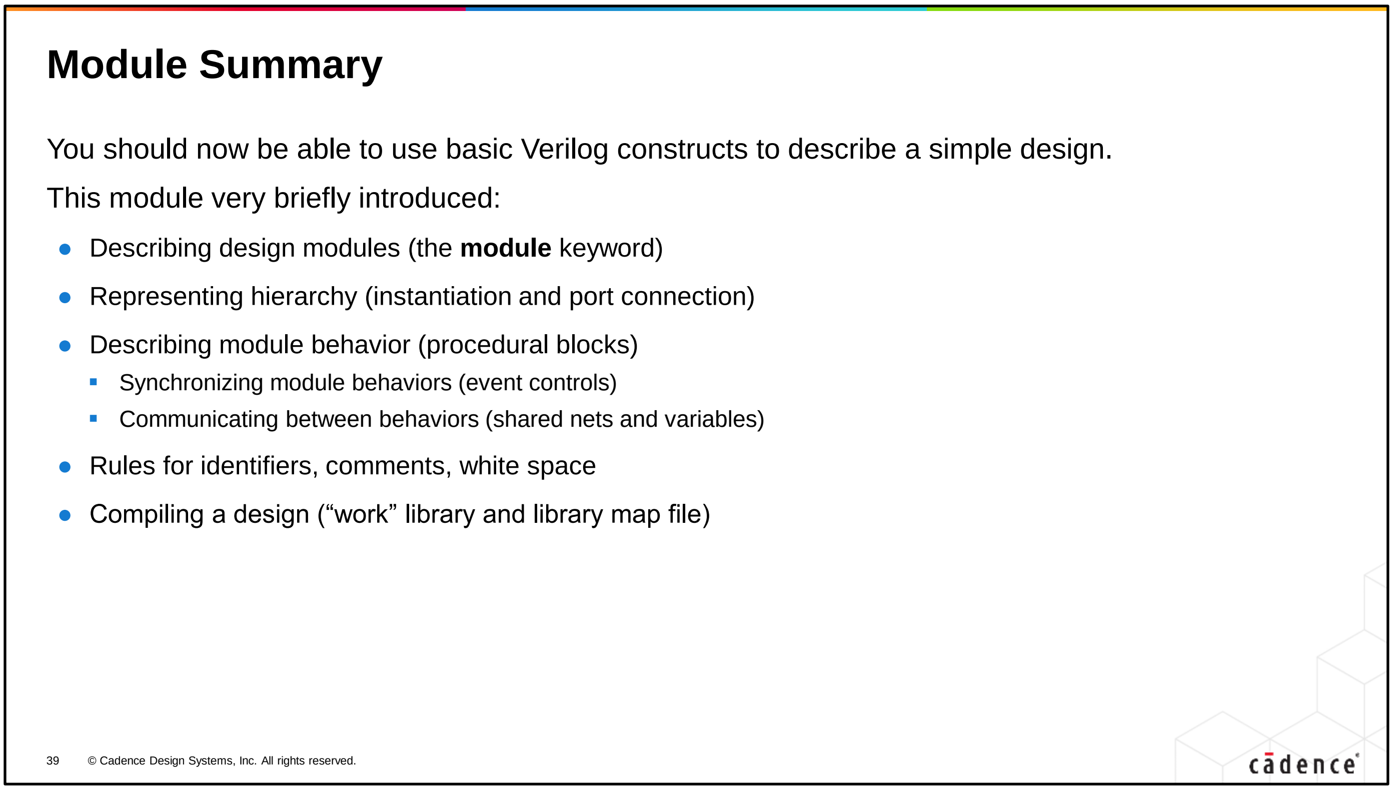
38 © Cadence Design Systems, Inc. All rights reserved.

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.
  - The behavioral processor module fully describes the processor functionality by itself.
  - The behavioral monitor module fully describes the monitor functionality by itself.
  - The RTL handler module partially describes the handler functionality in synthesizable code and instantiates two submodules to complete the description. The receiver submodule is an already-synthesized netlist of ASIC vendor macros and the sender submodule is a hand-crafted netlist of Verilog primitives.

You can normally compile these modules in any order. The exception is that if any file contains compiler directives like ‘ifdef, ifndef etc. that another file depends upon, then you must compile the files together and compile first the file containing the directives. Creating such cross-file dependencies is considered poor practice.

The elaborator later links together the compiled module descriptions to configure a design and testbench for simulation.



Module Summary

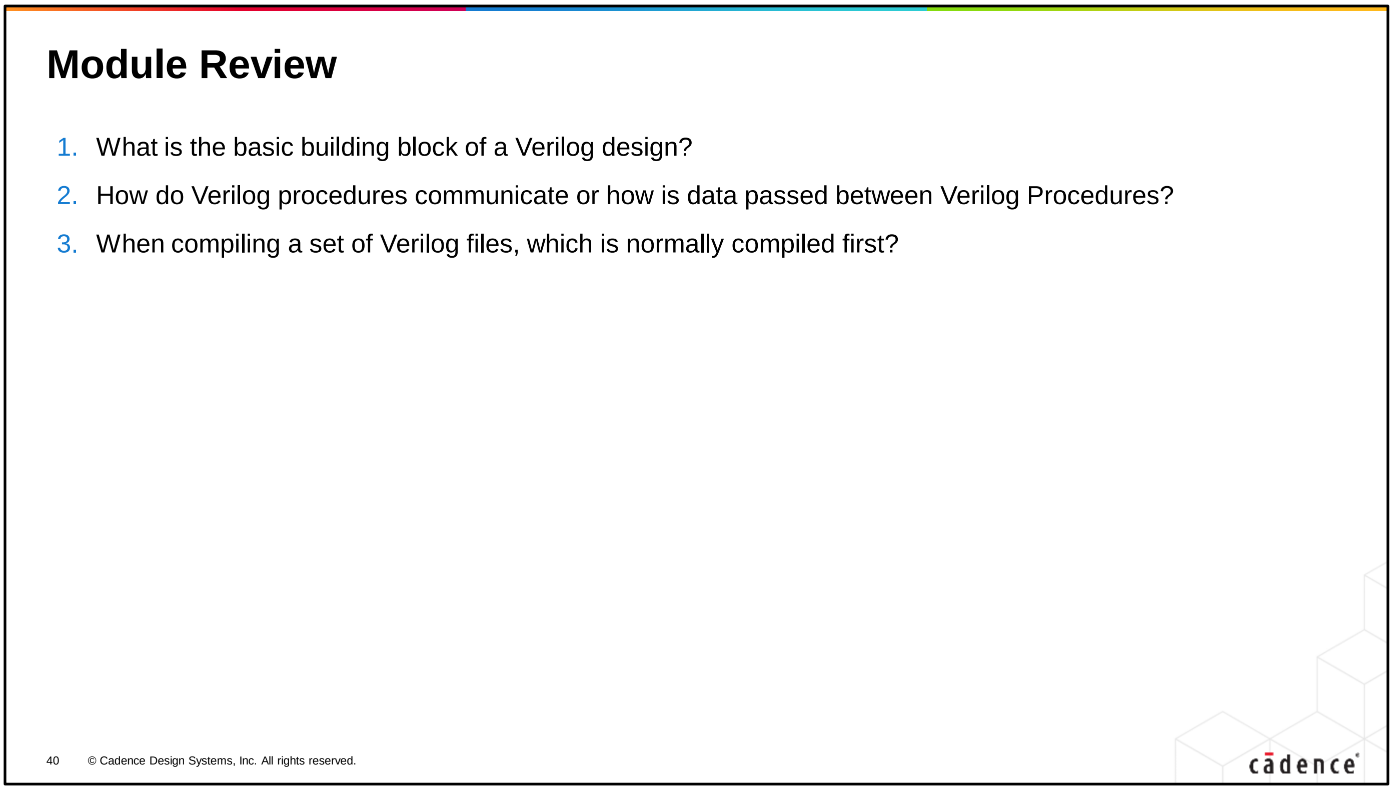
You should now be able to use basic Verilog constructs to describe a simple design.

This module very briefly introduced:

- Describing design modules (the **module** keyword)
- Representing hierarchy (instantiation and port connection)
- Describing module behavior (procedural blocks)
  - Synchronizing module behaviors (event controls)
  - Communicating between behaviors (shared nets and variables)
- Rules for identifiers, comments, white space
- Compiling a design ("work" library and library map file)

This module examined the fundamental language constructs and how you use them to describe a design:

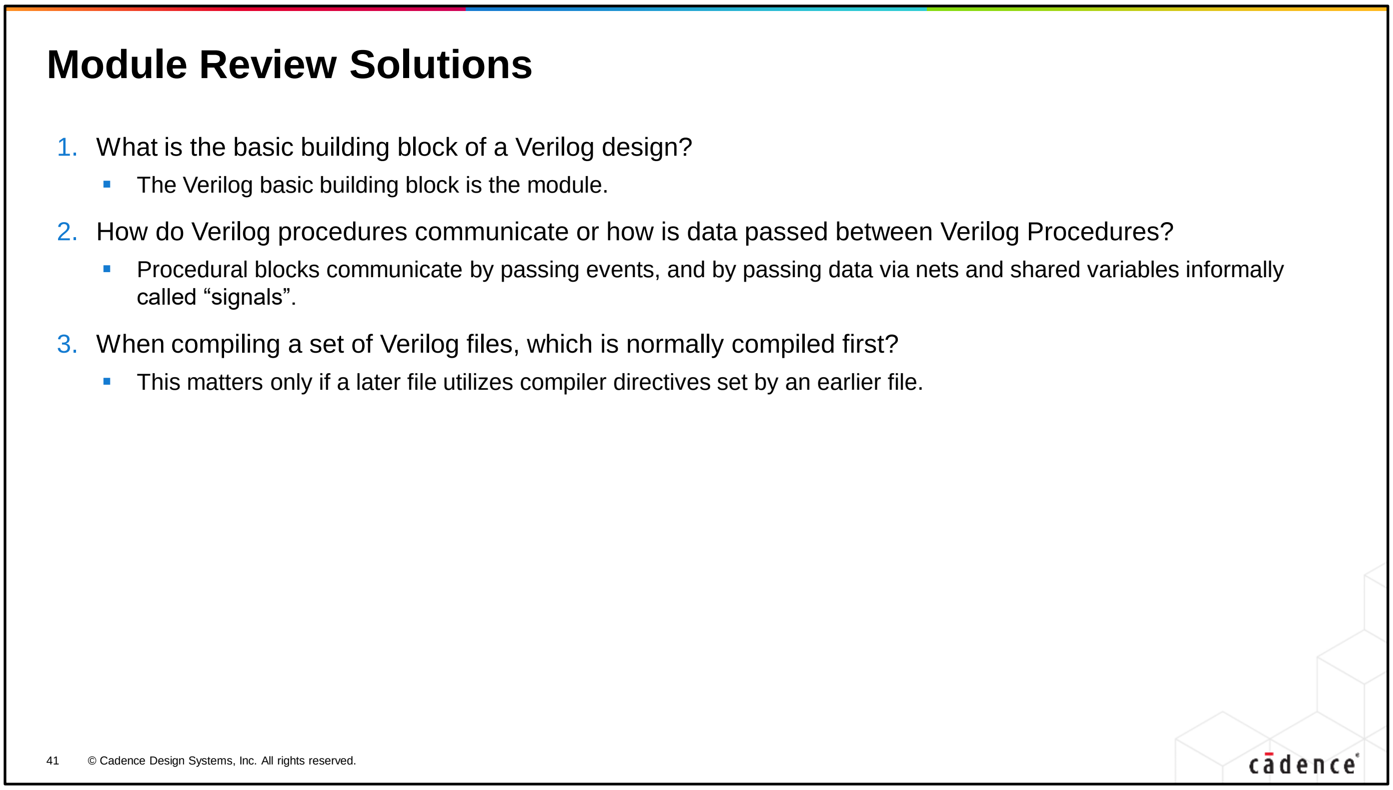
- The basic building block of design hierarchy is the module declaration.
- You create hierarchy by declaring ports, nets, variables and module instances and connecting ports of the module instances to the locally declared ports, nets, variables.
- To describe complex behavior, Verilog provides procedural constructs.
- Verilog provides procedural timing controls for *stepping* execution of procedural blocks. The most common of these is the event control.
- An identifier starts with an alphabetical character (a-z, A-Z) or an underscore (_) character and can contain alphanumeric characters, the dollar sign ($) character, and the underscore character.
- Single-line comments start with two consecutive slash (/) characters and terminate at the end of the line.
- A block comment starts with a slash character and following asterisk (*) character (/*) and terminates with the reverse – an asterisk character followed by a slash character (*/).
- Verilog is a free-format language. You can use white space to organize the code to enhance readability.
- The Verilog 2001 update provides a standard means to specify a design configuration.



Module Review

1. What is the basic building block of a Verilog design?
2. How do Verilog procedures communicate or how is data passed between Verilog Procedures?
3. When compiling a set of Verilog files, which is normally compiled first?

This page does not contain notes.



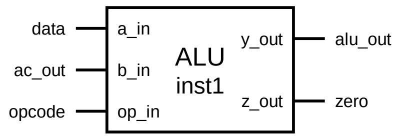
Module Review Solutions

1. What is the basic building block of a Verilog design?
   - The Verilog basic building block is the module.
2. How do Verilog procedures communicate or how is data passed between Verilog Procedures?
   - Procedural blocks communicate by passing events, and by passing data via nets and shared variables informally called “signals”.
3. When compiling a set of Verilog files, which is normally compiled first?
   - This matters only if a later file utilizes compiler directives set by an earlier file.

This page does not contain notes.

Module Exercise

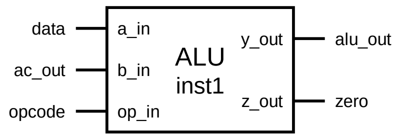
Write a module instantiation statement to instantiate this module:



This page does not contain notes.

Module Exercise Solution

Write a module instantiation statement to instantiate this module:



Solution:

```
ALU inst1
(
   .a_in   (data)     ,
   .b_in   (ac_out)   ,
   .op_in  (opcode)   ,
   .y_out  (alu_out)  ,
   .z_out  (zero)
) ;
```

This page does not contain notes.

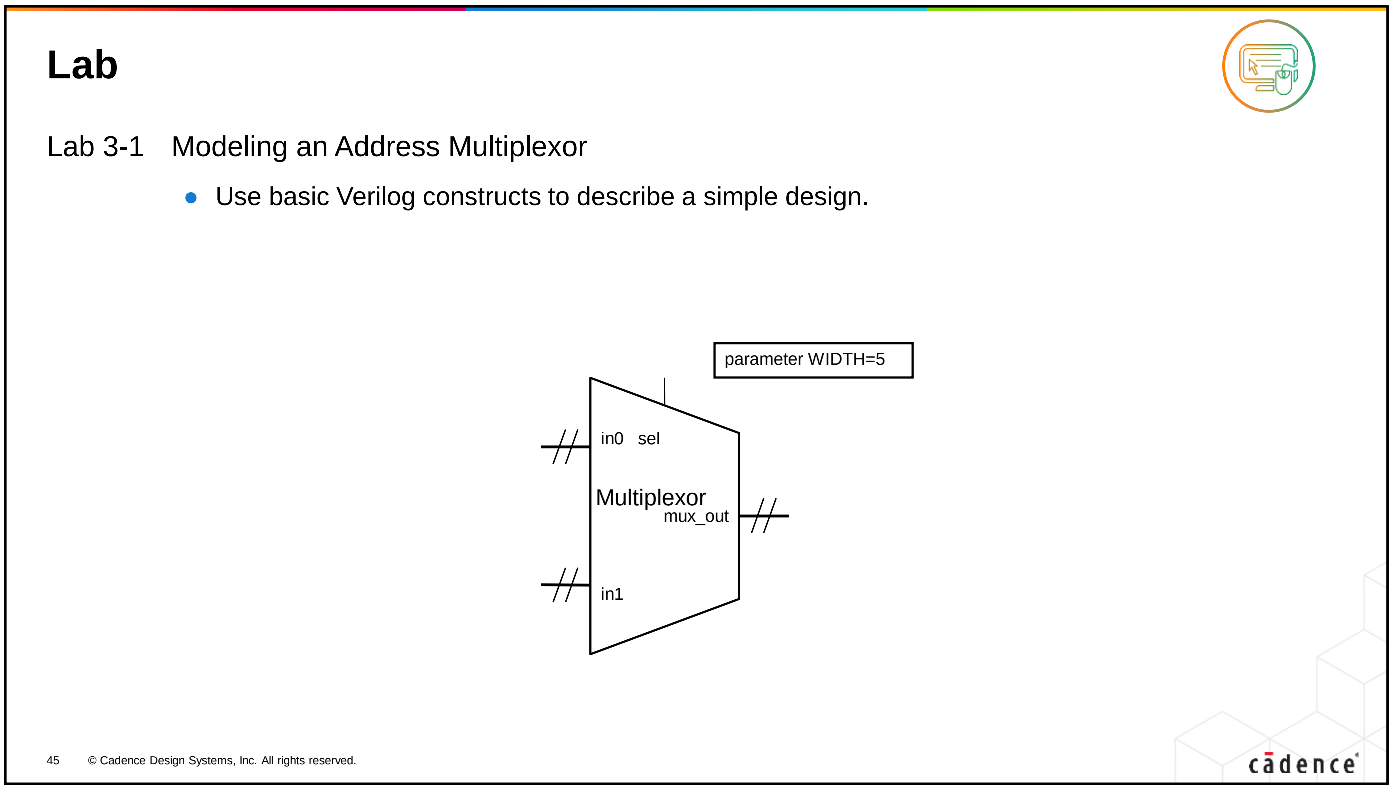
Video: Simulation Using Xcelium

Basic Verilog Simulation using Xcelium™

Manisha Pradhan, Lead Education Application Engineer
Education Services Team

cadence

This page does not contain notes.



Lab

Lab 3-1 Modeling an Address Multiplexor

- Use basic Verilog constructs to describe a simple design.

parameter WIDTH=5
in0
sel
Multiplexor
mux_out
in1

Your objective is to use basic Verilog constructs to describe a simple design.

For this lab, you use basic Verilog constructs to describe a parameterized-width two-to-one multiplexor. The lab instructions explain how to declare module parameters and vector ranges.