Module 11

Directing the Compiler

This module describes the Verilog compiler directives.

A compiler directive is a directive to the compiler. It is not syntactically a Verilog declaration or a Verilog statement. The scope of a compiler directive is from the point in the source stream input to the compiler to the point at which the directive is overridden or disabled or the end of the compilation session, probably across multiple files and multiple module descriptions, and sometimes affecting only parts of files or parts of module descriptions.

Verilog compiler directives are prefixed with the “back-tick” (`) character more formally called the accent grave. Wherever you see that character in Verilog code it is always associated with a compiler directive.

This module introduces compiler directives here. Although they are more closely related to lexical structure than to declarations and statements, you will understand them better if you have an understanding of Verilog declarations and statements. The course will later revisit many of these directives in an appropriate context that uses them.

Module Objective

In this module, you:

- Direct the compiler to interpret subsequent source code

Topics

- Substituting text
- Conditionally compiling code
- Including files
- Setting the timescale
- Reserving keywords
- Using pragmas

Your objective is to appropriately and effectively utilize compiler directives. To do that, you need to know what directives are available and what they do and how to use them.

Substituting Text: The `` `define `` Directive

Define a text substitution macro:

- `` `define name[(arguments)] text ``
- Text can use other text macros.

Use a text substitution macro:

- `` `name[(arguments)] ``
- Preprocessor substitutes text literally (WYSIWYG).

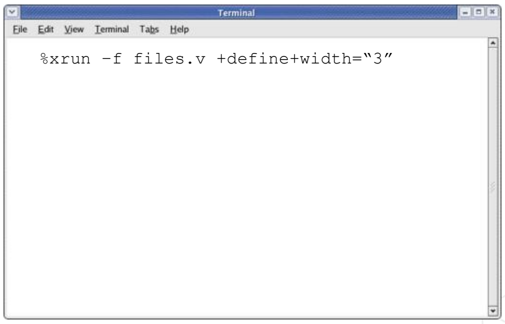
Undefine a text macro:

- `` `undef name ``

```
'define width 7
wire a, b, c;
reg [2:0] f;
reg ['width:0] mem [1:1024];

'define e {b, c, a};

always @ (posedge clk)
  f<= 'e ;  //compiler replaces macro
            // with its definition
```



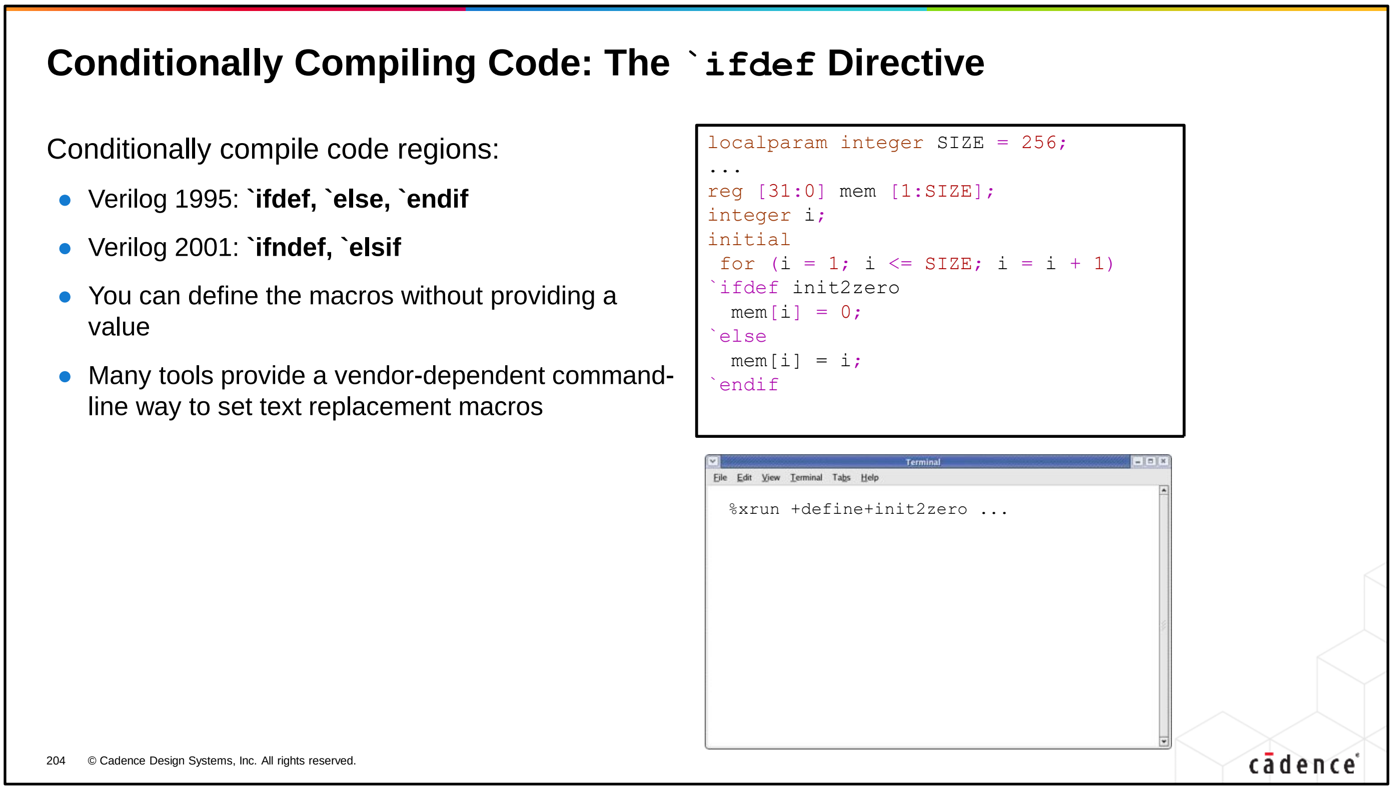
```
%xrun -f files.v +define+width="3"
```

You can use the text substitution facility anywhere that you want to write generic code that you can then easily replace during subsequent compilation sessions without the need to edit the target source file. You can define the text substitution macros in a separate file that you compile first during the compilation session, and many tools also provide a vendor-dependent command-line way to define text substitution macros.

As the scope of a compiler directive depends upon compilation order and thus can change between compilation sessions, you may want to instead use module parameters wherever possible for design constants such as delays and vector widths.

You can place a `define directive anywhere in your source code, but most people by convention place them all in a separate file, or at the top of the file if they are used in only one file.

- Text macros have their own name space. You can define a macro with the same name as a module without causing confusion. You cannot name the text macro with the same name as an existing compiler directive, as that would definitely cause confusion.
- You can include a parenthesized list of formal arguments. The opening parenthesis must appear immediately after the macro identifier with no intervening white space.
- A newline character terminates the macro definition. You can escape the newline with a backslash (\) character to continue the definition on a subsequent line.
- Some constructs you cannot split across text macros. This means that if a macro definition starts the construct then it must complete the construct in that same macro definition. Constructs you cannot split across text macros are comments, identifiers, keywords, numbers, operators and strings.



Conditionally Compiling Code: The `ifdef Directive

Conditionally compile code regions:

- Verilog 1995: **`ifdef, `else, `endif**
- Verilog 2001: **`ifndef, `elsif**
- You can define the macros without providing a value
- Many tools provide a vendor-dependent command-line way to set text replacement macros

```
localparam integer SIZE = 256;
...
reg [31:0] mem [1:SIZE];
integer i;
initial
 for (i = 1; i <= SIZE; i = i + 1)
`ifdef init2zero
  mem[i] = 0;
`else
  mem[i] = i;
`endif
```

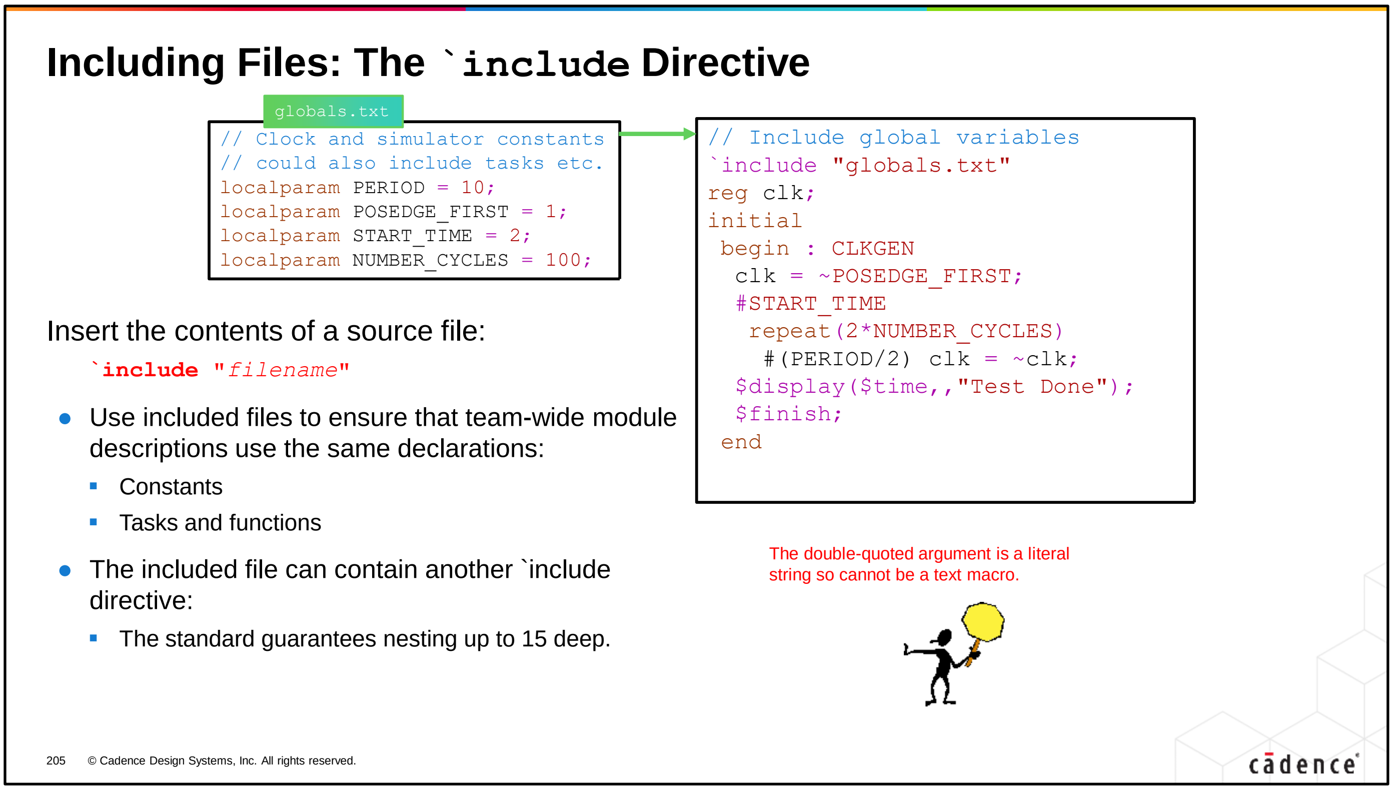
```
%xrun +define+init2zero ...
```

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

You can use the existence of a text macro to conditionally compile lines of source code.

You use these directives similarly in an algorithmic sense to the if-else statement, but of course keep in mind that these macros must each appear on their own line and it is the preprocessor that parses them during compile time.

This example provides the option to initialize the ROM to either all zeros or to values read from a file.



Including Files: The `include Directive

globals.txt

```
// Clock and simulator constants
// could also include tasks etc.
localparam PERIOD = 10;
localparam POSEDGE_FIRST = 1;
localparam START_TIME = 2;
localparam NUMBER_CYCLES = 100;
```

```
// Include global variables
`include "globals.txt"
reg clk;
initial
 begin : CLKGEN
  clk = ~POSEDGE_FIRST;
  #START_TIME
   repeat(2*NUMBER_CYCLES)
    #(PERIOD/2) clk = ~clk;
  $display($time,,"Test Done");
  $finish;
 end
```

Insert the contents of a source file:

```
`include "filename"
```

- Use included files to ensure that team-wide module descriptions use the same declarations:
  - Constants
  - Tasks and functions
- The included file can contain another `include directive:
  - The standard guarantees nesting up to 15 deep.

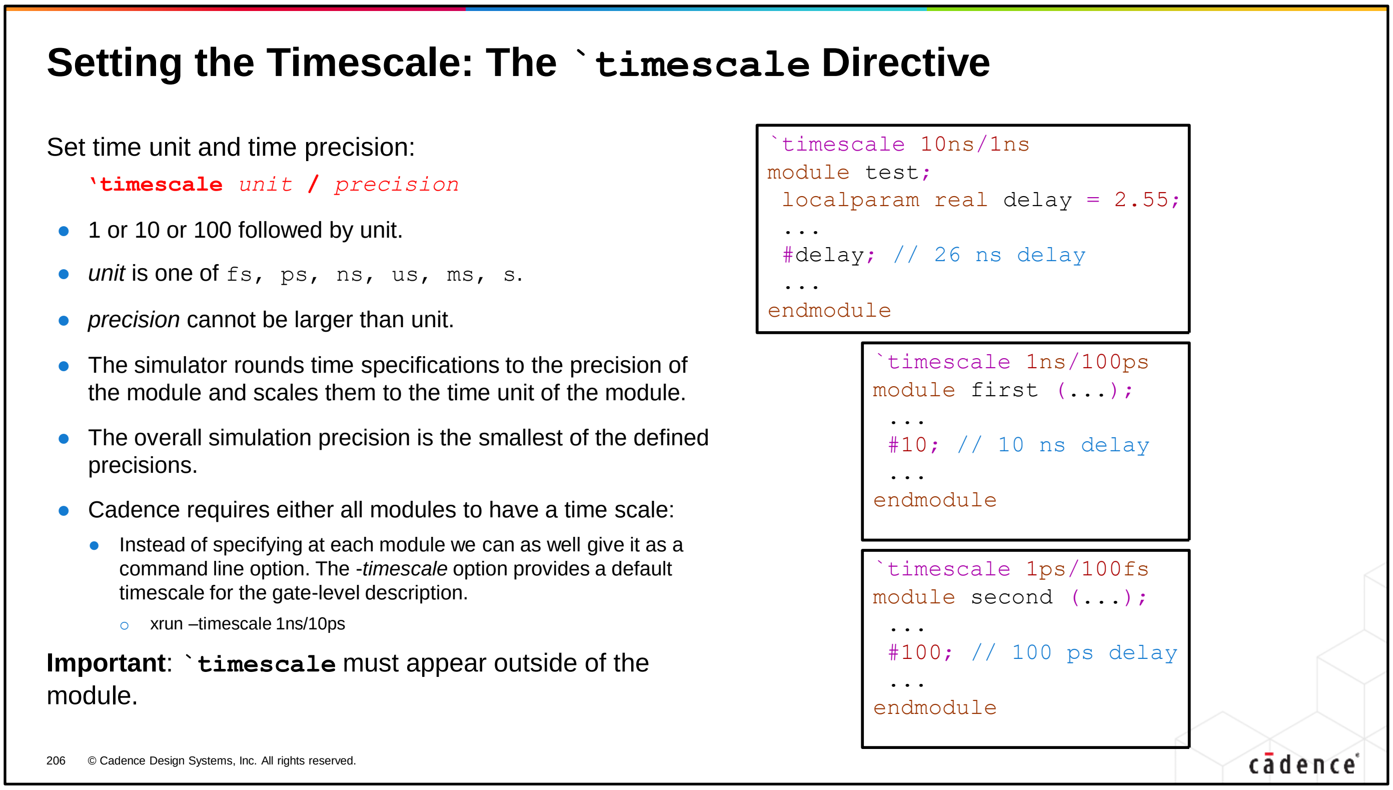
The double-quoted argument is a literal string so cannot be a text macro.

The `include directive includes the contents of a file at the point where the directive appears. The effect is exactly as if the file contents were copied into the source at that point to replace the `include directive.

People use this directive to include declarations that are used by all team members and so for the sake of consistency should be maintained in one place.

The standard guarantees that you can nest included files at least 15 deep.

The argument to the `include directive is a quoted file path name and not a compiler directive or string literal. No standard way exists to generate the file name to be included.



Setting the Timescale: The `` `timescale `` Directive

Set time unit and time precision:

`'timescale` *unit* / *precision*

- 1 or 10 or 100 followed by unit.
- *unit* is one of `fs, ps, ns, us, ms, s.`
- *precision* cannot be larger than unit.
- The simulator rounds time specifications to the precision of the module and scales them to the time unit of the module.
- The overall simulation precision is the smallest of the defined precisions.
- Cadence requires either all modules to have a time scale:
  - Instead of specifying at each module we can as well give it as a command line option. The *-timescale* option provides a default timescale for the gate-level description.
    - xrun –timescale 1ns/10ps

Important: `` `timescale `` must appear outside of the module.

```
`timescale 10ns/1ns
module test;
 localparam real delay = 2.55;
 ...
 #delay; // 26 ns delay
 ...
endmodule
```

```
`timescale 1ns/100ps
module first (...);
 ...
 #10; // 10 ns delay
 ...
endmodule
```

```
`timescale 1ps/100fs
module second (...);
 ...
 #100; // 100 ps delay
 ...
endmodule
```

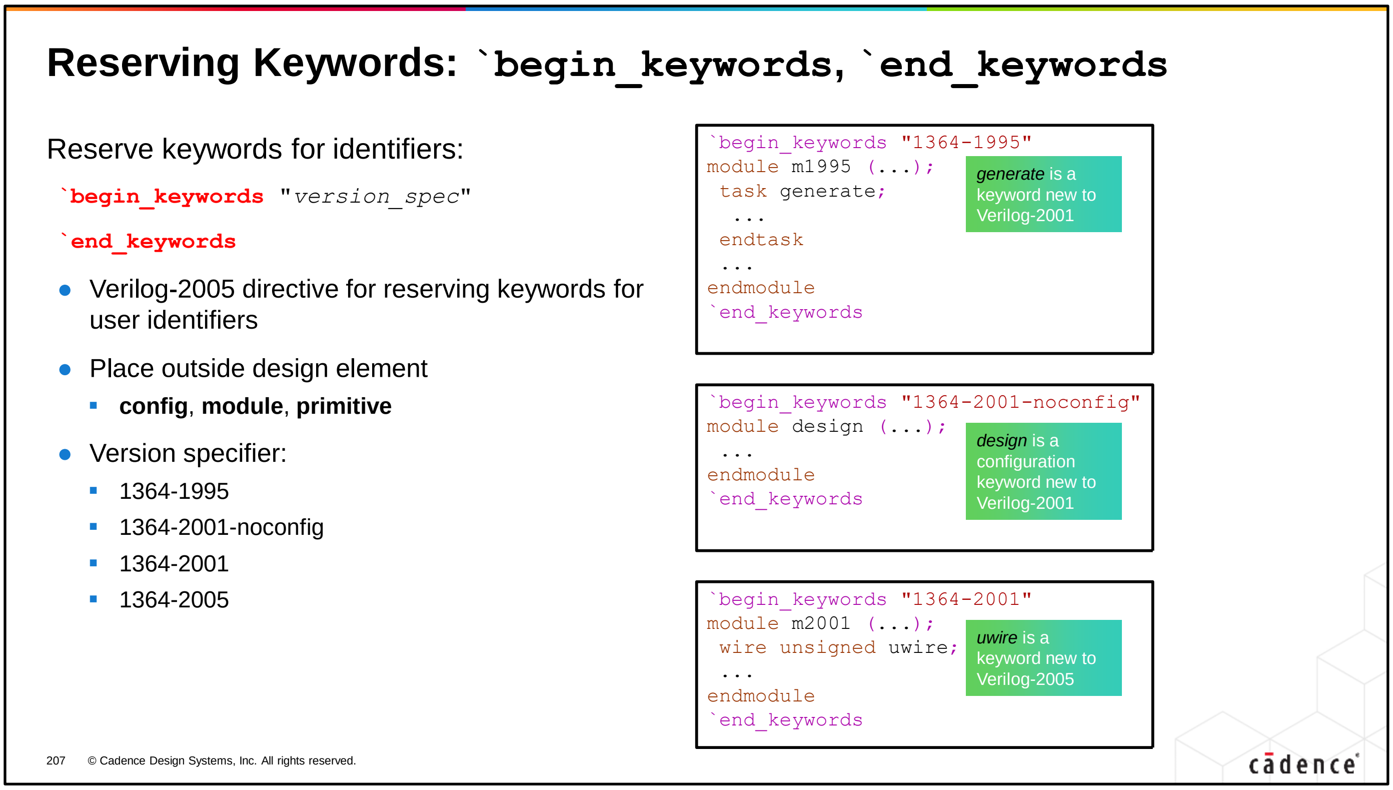
206 © Cadence Design Systems, Inc. All rights reserved.

cādence

The `timescale directive specifies the time unit and time precision for subsequent module declarations. Any expression used in a context to mean simulation time will assume the time unit of the current module and be rounded to the time precision of the current module. System tasks that display simulation time will scale the display to the time unit of the current module.

As the standard does not prescribe a default time scale, for interoperability either all modules must be subject to a time scale or no module can be subject to a time scale. To ensure that all modules are subject to a time scale, you should get used to specifying a time scale at the top of every file, even if you not use time specifications in the file. To modify a time scale, you need to edit the file and recompile that file and all other files that utilized that directive. No method exists to modify a time scale during simulation.

In this example, the file containing the test module specifies that until further notice all subsequent time specifications are in units of 10ns and the time precision is 1ns. The 2.55 real value when used as a delay then means 25.5ns and is rounded to 26ns. The file containing the “second” module specifies that until further notice all subsequent time specifications are in units of 1ps and the time precision is 100fs. As this is the finest precision that any of these modules uses, if no other module requires a finer precision, this is the precision of the simulation.



Reserving Keywords: `` `begin_keywords ``, `` `end_keywords ``

Reserve keywords for identifiers:

```
`begin_keywords "version_spec"
`end_keywords
```

- Verilog-2005 directive for reserving keywords for user identifiers
- Place outside design element
  - **config**, **module**, **primitive**
- Version specifier:
  - 1364-1995
  - 1364-2001-noconfig
  - 1364-2001
  - 1364-2005

```
`begin_keywords "1364-1995"
module m1995 (...);
 task generate;
  ...
 endtask
 ...
endmodule
`end_keywords
```

generate is a keyword new to Verilog-2001

```
`begin_keywords "1364-2001-noconfig"
module design (...);
 ...
endmodule
`end_keywords
```

design is a configuration keyword new to Verilog-2001

```
`begin_keywords "1364-2001"
module m2001 (...);
 wire unsigned uwire;
 ...
endmodule
`end_keywords
```

uwire is a keyword new to Verilog-2005

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

cadence

The `begin_keywords directive tells a parser to reserve for user identifiers all new keywords defined by a later standard. These keywords can appear only outside design elements, such as outside configuration, module, or primitive declarations.

If you have a legacy module that uses the generate word as a user identifier, then you can direct a Verilog 2005 compliant compiler to not use any keywords more recent than the 1995 standard for that module.

If you have a legacy module that uses the design word as a user identifier, you can direct a Verilog 2005 compliant compiler to not use any Verilog 2001 configuration keywords or any other keywords more recent than the 2001 standard for that module.

If you have a legacy module that uses the uwire word as a user identifier, you can direct a Verilog 2005 compliant compiler to not use any keywords more recent than the 2001 standard for that module.

Keywords new to IEEE Std 1364-2001

```
automatic cell config design endconfig endgenerate generate genvar incdir
include instance liblist library localparam noshowcancelled
pulsestyle_onevent pulsestyle_ondetect showcancelled signed unsigned use
```

Keywords new to IEEE Std 1364-2005

```
uwire
```

Using Pragmas: The `pragma Directive

Utilize a pragma:

```
`pragma pragma_name
  [  pragma_expression
  {, pragma_expression } ]
```

- Verilog-2005 structure for implementation-specific directives
- Verilog defines just a few:
  - `` `pragma reset ``
  - `` `pragma resetall ``
  - `` `pragma protect ``

```
module smux
#(parameter integer W=1)
 (output [W-1:0] y, input s,
  input  [W-1:0] a, b);
`pragma protect begin
 assign y = s ? b : a ;
`pragma protect end
endmodule
```

Area to protect

cadence

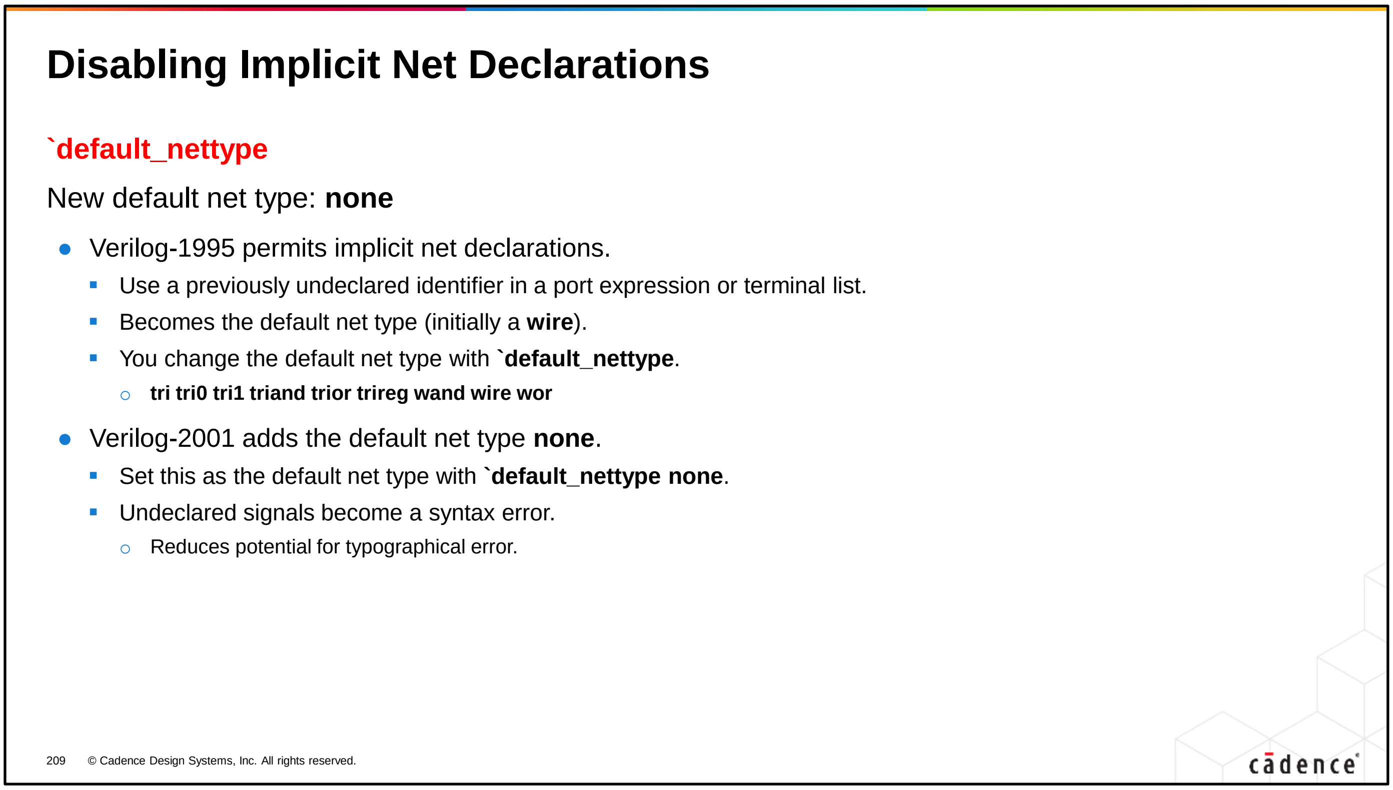
The `pragma directive provides a means for implementations to extend the set of compiler directives. A pragma name follows the directive and is followed by an optional list of pragma expressions. A pragma expression is a pragma keyword or a pragma value or an assignment of a pragma value to a pragma keyword.

The Verilog 2005 update defines only a few pragmas:

- The reset pragma resets the pragmas provided as pragma expressions;
- The resetall pragma resets all pragmas; and
- The protect pragma encrypts subsequent source code.

A implementation ignores pragmas it does not recognize.

```
pragma ::=
  `pragma pragma_name [ pragma_expression { , pragma_expression } ]
pragma_name ::= simple_identifier
pragma_expression ::=
  pragma_keyword
| pragma_keyword = pragma_value
| pragma_value
pragma_keyword ::= simple_identifier
pragma_value ::=
( pragma_expression { , pragma_expression } )
| number
| string
| identifier
```



Disabling Implicit Net Declarations

`default_nettype

New default net type: **none**

- Verilog-1995 permits implicit net declarations.
  - Use a previously undeclared identifier in a port expression or terminal list.
  - Becomes the default net type (initially a **wire**).
  - You change the default net type with **`default_nettype**.
    - **tri tri0 tri1 triand trior trireg wand wire wor**
- Verilog-2001 adds the default net type **none**.
  - Set this as the default net type with **`default_nettype none**.
  - Undeclared signals become a syntax error.
    - Reduces potential for typographical error.

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

You implicitly declare a net when you use a previously undeclared identifier in a port expression or terminal list or as the lvalue of a continuous assignment. That net by default becomes a wire. You set the value of the `default_nettype directive to change that default net type. The Verilog 2001 standard adds the none type. When you set the default net type to none, you require explicit declaration of all nets, thus reducing the potential for typographical error.

Note: All Verilog compiler directives start with the grave accent (sometimes called a “back-tick”). You should be aware that some fonts cannot render this character properly and display it as an acute accent (a “forward tick”).

Reference: Compiler Directives

`` `celldefine ``
 `` `endcelldefine ``
	Tags a library cell

`` `default_nettype ``	Sets the net type for implicit net declarations

`` `define ``
 `` `undef ``
	Defines and undefines a text macro

`` `ifdef ``
 `` `else ``
 `` `endif ``
	Conditionally compiles code depending upon text macro existence

`` `include ``	Includes a source file
`` `resetall ``	Resets directives to their initial state
`` `timescale ``	Sets the time units and time precision

`` `unconnected_drive ``
 `` `nounconnected_drive ``
	Pulls up/pulls down unconnected module inputs

`` `ifndef ``
 `` `elsif ``
	Verilog-2001: More conditional compilation

`` `line ``	Verilog-2001: Overrides reported source file and line upon error

`` `begin_keywords ``
 `` `end_keywords ``
	Verilog-2005: Reserves keywords for use as identifiers

`` `pragma ``	Verilog-2005: Changes how compiler interprets subsequent source

Use the `celldefine and `endcelldefine directives to tag modules as technology cells. PLI applications such as delay calculators and power calculators query this tag to determine the design hierarchy leaf cells that have a physical implementation.

Use the `default_nettype directive to set the default net type for implicitly declared nets. This is by default the wire type but you can set it to any net type. The Verilog 2001 standard adds the argument none which prevents implicitly declared nets.

Use the `define directive to define a text replacement macro and the `undef directive to undefine it.

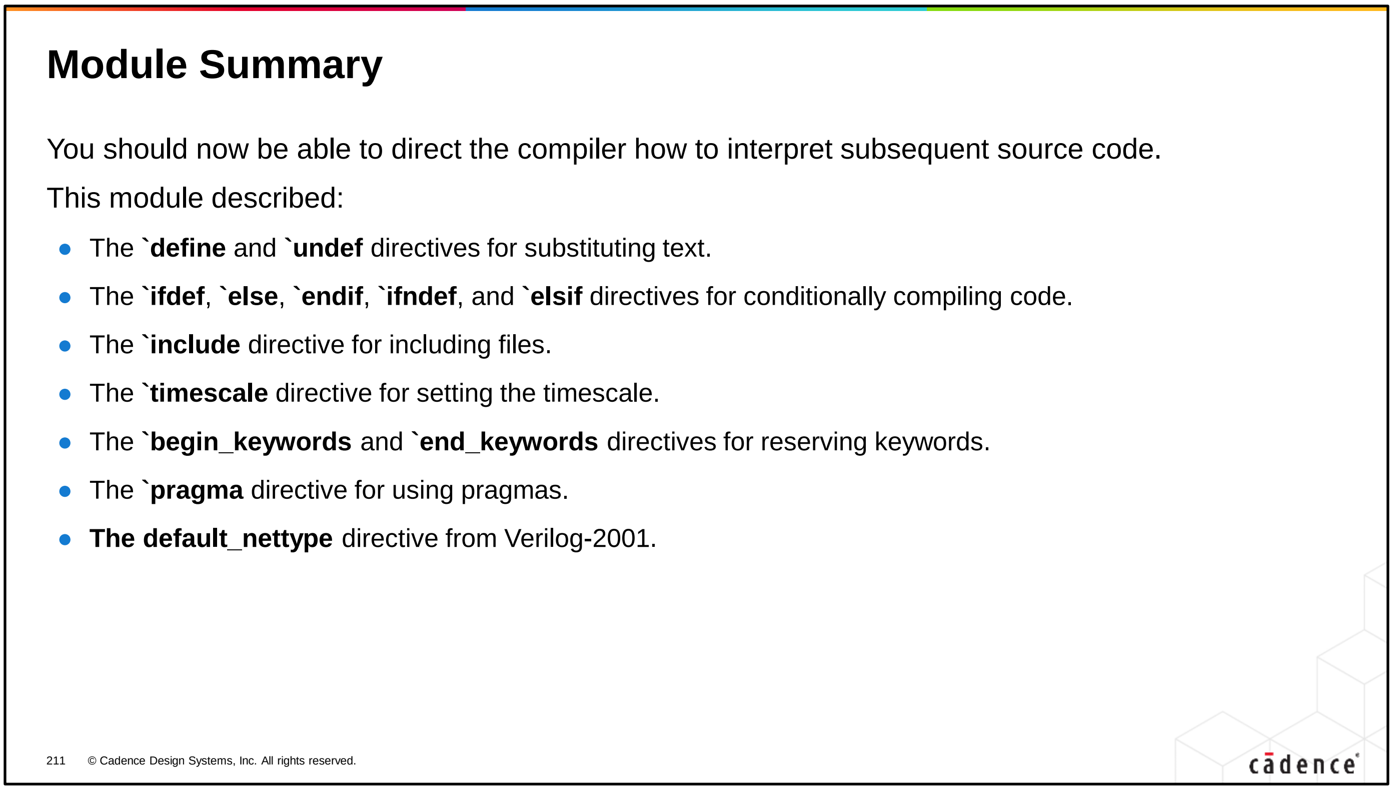
Use the `ifdef, `else and `endif directives to conditionally compile source text depending upon the existence of a text macro. The Verilog 2001 standard adds the `ifndef and `elsif directives.

Use the `include directive to insert the contents of a source file at the point of the directive. The argument to the directive is a quoted file name. No standard way exists to generate the file name.

Use the `resetall directive to reset all compiler directives back to their initial state. This is very useful and recommended for placing at the beginning of every source file to ensure that only the directives set by that file are in effect during parsing of the file. This directive affects the `celldefine, `default_nettype, `timescale and `unconnected_drive directives. It does not undefine text macros, so your file can still be dependent upon text macros defined in some other file.

Use the `timescale directive to specify the time unit and time precision for subsequent module declarations. Any expression used in a context to mean simulation time will assume the time unit of the current module and be rounded to the time precision of the current module. System tasks that display simulation time will scale the display to the time unit of the current module.

Use the `pragma directive to change how the compiler interprets subsequent source. A pragma consists of a name and an optional keyword and an optional value. The standard defines only the reset and resetall keywords. Tool vendors define additional pragmas.



Module Summary

You should now be able to direct the compiler how to interpret subsequent source code.

This module described:

- The **`define** and **`undef** directives for substituting text.
- The **`ifdef**, **`else**, **`endif**, **`ifndef**, and **`elsif** directives for conditionally compiling code.
- The **`include** directive for including files.
- The **`timescale** directive for setting the timescale.
- The **`begin_keywords** and **`end_keywords** directives for reserving keywords.
- The **`pragma** directive for using pragmas.
- **The default_nettype** directive from Verilog-2001.

Your objective is to appropriately and effectively utilize compiler directives. To do that, you need to know what directives are available and what they do and how to use them.

Module Review

1. True or False: The **`undefall** compiler directive undefines all defined text macros.
2. True or False: The standard pre-defines the VERILOG text macro that you can use to permit only the Verilog simulator to compile the code.
3. What file does this code attempt to include?
   a. `` `define ext inc ``
   b. `` `define test(what) test.what.`ext ``
   c. `` `include "`test(alu)" ``
4. What is the IEEE-standard default timescale for a module not subject to a **`timescale** directive?

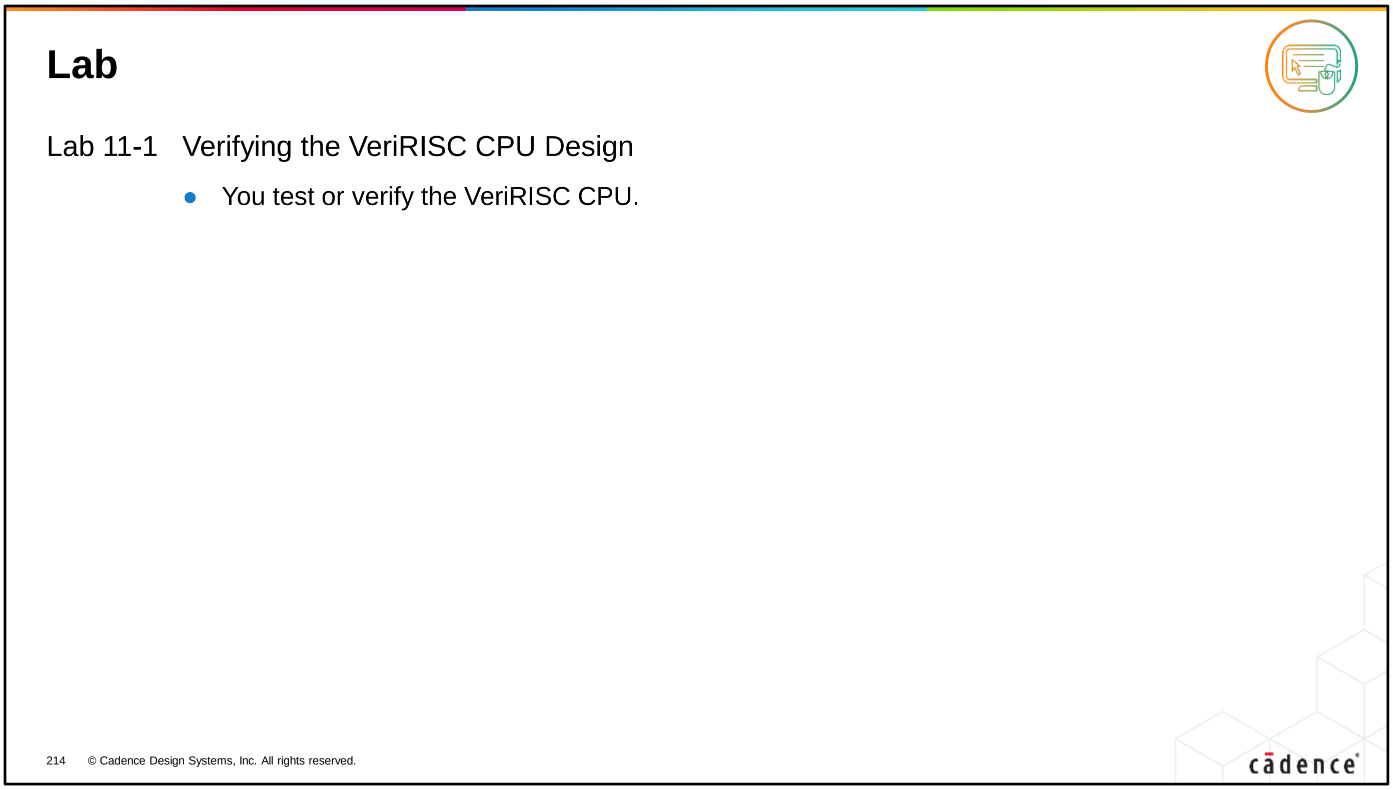
This page does not contain notes.

Module Review Solutions

1. True or False: The `undefall compiler directive undefines all defined text macros.
   - False. The standard does not describe an **`undefall** compiler directive. The **`undef** directive undefines one text macro.
2. True or False: The standard pre-defines the VERILOG text macro that you can use to permit only the Verilog simulator to compile the code.
   - False. The standard does not pre-define any text macros. Vendors may pre-define their own text macros such as VERILOG.
3. What file does this code attempt to include?
   - **``define ext inc``**
   - **``define test(what) test.what.`ext``**
   - **``include "`test(alu)"``**

   It attempts to include the file "`test(alu)". The compiler directive's argument is a string literal. No generation is permitted.
4. What is the IEEE-standard default timescale for a module not subject to a **`timescale** directive?
   - The standard does not define a default time scale. If ANY module is subject to a time scale then ALL modules must be subject to a timescale.

This page does not contain notes.



Lab

Lab 11-1 Verifying the VeriRISC CPU Design

- You test or verify the VeriRISC CPU.

Your objective is to direct the compiler.

For this lab, you use compiler directives to define a mixed-level configuration of components.