9

HDL Compiler Directives

The Synopsys Verilog HDL Compiler translates a Verilog description to the internal format Design Compiler uses. Specific aspects of this process can be controlled by special comments in the Verilog source code called HDL Compiler directives. Because these directives are a special case of regular comments, they are ignored by the Verilog HDL Simulator and do not affect simulation.

This chapter describes HDL Compiler directives and their effect on translation, in the following sections:

- Verilog Preprocessor Directives
- Notation for HDL Compiler Directives
- translate_off and translate_on Directives
- parallel_case Directive
- full_case Directive

- state_vector Directive
- enum Directive
- template Directive
- Embedding Constraints and Attributes
- Component Implication

Verilog Preprocessor Directives

Verilog preprocessing provides the following features:

- `-define` option to the `analyze` command
- dc_shell variables
- The `‘ifdef`, `‘else`, and `‘endif` directives
- The DC Macro
- Extended capabilities for the `‘define` directive

Define Option to the analyze Command

An option to the `analyze` command, `-define` (or `-d`, abbreviated), allows macro definition on the command line.

You can use only one `-define` per `analyze` command. But the argument can be a list of macros, as shown in Example 9-1.

You do not need to use curly brackets to enclose one macro, as shown in Example 9-2.

Example 9-1 analyze Command With List of Defines

```
analyze -f verilog -d { RIPPLE, SIMPLE } mydesign.v
```

Example 9-2 analyze Command With One Define

```
analyze -f verilog -define ONLY_ONE mydesign.v
```

Note:

The `read` command does not accept the `-d` option.

The input to the `analyze` command continues to be a Verilog file. The output of the `analyze` command continues to be a .syn file.

dc_shell Variables

These variables perform the following functions:

`hdlin_preserve_vpp_files`

By default, this variable is false. When it is false, intermediate preprocessor files are deleted after use.

Preprocessor files are preserved (not deleted) when this flag is set to true.

`hdlin_vpp_temporary_directory`

Indicates where the intermediate preprocessor files are created. The default is to use the user's WORK directory.

`hdlin_enable_vpp`

When set to true (the default), `hdlin_enable_vpp` allows interpretation of the `‘ifdef`, `‘else`, and `‘endif` directives.

It also activates `‘define` extensions, which allow macros with arguments.

‘ifdef, ‘else, and ‘endif Directives

The `‘ifdef`, `‘else`, and `‘endif` directives allow the conditional inclusion of code.

The macros that are arguments to the `‘ifdef` directives can also be defined in the Verilog source file by use of the `‘define` directive. In that case, there is no change in the invocation of the HDL Compiler to read in Verilog files. Example 9-3 shows a design that uses the directives.

Example 9-3 Design Using Preprocessor Directives and ‘define

```
‘ifdef SELECT_XOR_DESIGN

module selective_design(a,b,c);
input a, b;
output c;
  assign c = a ^ b;
endmodule

‘else

module selective_design(a,b,c);
input a, b;
output c;
  assign c = a | b;
endmodule

‘endif
```

DC Macro

The special macro DC is always defined, as in the following example:

Example 9-4 DC Macro

```
‘ifdef DC
    ...
    ...      /* Synthesis-only information */
    ...
‘else
    ...
    ...      /* Simulation-only information */
    ...
‘endif
```

The Verilog preprocessor directives are not affected by `translate_off` and `translate_on` (described in “translate_off and translate_on Directives” on page 9-6); that is, the preprocessor reads whatever is between `translate_off` and `translate_on`.

To suspend translation of the source code for synthesis, use the `‘ifdef, ‘else`, `‘endif` construct, not `translate_off` and `translate_on`.

‘define Verilog Preprocessor Directive

With the dc_shell variable `hdlin_enable_vpp` set to true, the `‘define` directive can specify macros that take arguments. For example,

```
‘define BYTE_TO_BITS(arg)((arg) << 3)
```

The `‘define` directive can do more than simple text substitution. It can also take arguments and substitute their values in its replacement text.

Notation for HDL Compiler Directives

The special comments that make up HDL Compiler directives begin, like all other Verilog comments, with the characters // or /*. The // characters begin a comment that fits on one line (most HDL Compiler directives do). If you use the /* characters to begin a multiline comment, you must end the comment with */. You do not need to use the /* characters at the beginning of each line but only at the beginning of the first line. If the word following these characters is *synopsys* (all lowercase) or an alternative defined in Design Compiler with the `hdlin_pragma_keyword` variable, HDL Compiler treats the remaining comment text as a compiler directive.

Note:

You cannot use `// synopsys` in a regular comment. Also, the compiler displays a syntax error if Verilog code is in a `// synopsys` directive.

translate_off and translate_on Directives

When the `// synopsys translate_off` and `// synopsys translate_on` directives are present, HDL Compiler suspends translation of the source code and restarts translation at a later point. Use these directives when your Verilog source code contains commands specific to simulation that HDL Compiler does not accept.

Note:

The Verilog preprocessor directives are not affected by `translate_off` and `translate_on`, and the preprocessor reads whatever is between them (see “DC Macro” on page 9-4).

You turn translation off by using

```
// synopsys translate_off
```

or

```
/* synopsys translate_off */
```

You turn translation back on by using

```
// synopsys translate_on
```

or

```
/* synopsys translate_on */
```

At the beginning of each Verilog file, translation is enabled. After that, you can use the `translate_off` and `translate_on` directives anywhere in the text. These directives must be used in pairs. Each `translate_off` must appear before its corresponding `translate_on`. Example 9-5 shows a simulation driver protected by a `translate_off` directive.

Example 9-5 // synopsys translate_on and // synopsys translate_off Directives

```
module trivial (a, b, f);
input a,b;
output f;
    assign f = a & b;

    // synopsys translate_off
    initial $monitor (a, b, f);
    // synopsys translate_on
endmodule

/* synopsys translate_off */
module driver;
    reg [1:0] value_in;
    integer i;

    trivial triv1(value_in[1], value_in[0]);

    initial begin
        for (i = 0; i < 4; i = i + 1)
            #10 value_in = i;
    end
endmodule
/* synopsys translate_on */
```

parallel_case Directive

The `// synopsys parallel_case` directive affects the way logic is generated for the `case` statement. As presented in “Full Case and Parallel Case” on page 5-19, a `case` statement generates the logic for a priority encoder. Under certain circumstances, you might not want to build a priority encoder to handle a `case` statement. You can use the `parallel_case` directive to force HDL Compiler to generate multiplexer logic instead.

The syntax for the parallel_case directive is

```
// synopsys parallel_case
```

or

```
/* synopsys parallel_case */
```

In Example 9-6, the states of a state machine are encoded as one hot signal. If the case statement were implemented as a priority encoder, the generated logic would be unnecessarily complex.

Example 9-6 // synopsys parallel_case Directives

```
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
    state3 = 4'b0100, state4 = 4'b1000;

case (1)//synopsys parallel_case

    current_state[0] : next_state = state2;
    current_state[1] : next_state = state3;
    current_state[2] : next_state = state4;
    current_state[3] : next_state = state1;

endcase
```

Use the parallel_case directive immediately after the case expression, as shown. This directive makes all case-item evaluations in parallel. All case items that evaluate to true are executed, not just the first, which could give you unexpected results.

In general, use parallel_case when you know that only one case item is executed. If only one case item is executed, the logic generated from a parallel_case directive performs the same function as the

circuit when it is simulated. If two case items are executed and you have used the `parallel_case` directive, the generated logic is not the same as the simulated description.

full_case Directive

The `// synopsys full_case` directive asserts that all possible clauses of a `case` statement have been covered and that no default clause is necessary. This directive has two uses: It avoids the need for default logic, and it can avoid latch inference from a `case` statement by asserting that all necessary conditions are covered by the given branches of the `case` statement. As shown in “Full Case and Parallel Case” on page 5-19, a latch can be inferred whenever a variable is not assigned a value under all conditions.

The syntax for the `full_case` directive is

```
// synopsys full_case
```

or

```
/* synopsys full_case */
```

If the `case` statement contains a default clause, HDL Compiler assumes that all conditions are covered. If there is no default clause and you do not want latches to be created, use the `full_case` directive to indicate that all necessary conditions are described in the `case` statement.

Example 9-7 shows two uses of `full_case`. The `parallel_case` and `full_case` directives can be combined in one comment.

Example 9-7 // synopsys full_case Directives

```
reg [1:0] in, out;
reg [3:0] current_state, next_state;
parameter state1 = 4'b0001, state2 = 4'b0010,
          state3 = 4'b0100, state4 = 4'b1000;

case (in) // synopsys full_case
    0: out = 2;
    1: out = 3;
    2: out = 0;
endcase

case (1)  // synopsys parallel_case full_case
    current_state[0] : next_state = state2;
    current_state[1] : next_state = state3;
    current_state[2] : next_state = state4;
    current_state[3] : next_state = state1;
endcase
```

In the first `case` statement, the condition `in == 3` is not covered. You can either use a default clause to cover all other conditions or use the `full_case` directive (as in Example 9-7) to indicate that other branch conditions do not occur. If you cover all possible conditions explicitly, HDL Compiler recognizes the `case` statement as full-case, so the `full_case` directive is not necessary.

The second `case` statement in Example 9-7 does not cover all 16 possible branch conditions. For example, `current_state == 4'b0101` is not covered. The `parallel_case` directive is used in this example because only one of the four case items can evaluate to true and be executed.

Although you can use the `full_case` directive to avoid creating latches, using this directive does not guarantee that latches will not be built. You still must assign a value to each variable used in the

case statement in all branches of the case statement. Example 9-8 illustrates a situation in which the full_case directive prevents a latch from being inferred for variable b but not for variable a.

Example 9-8 Latches and // synopsys full_case

```
reg a, b;
reg [1:0] c;
case (c)   // synopsys full_case
    0: begin a = 1; b = 0; end
    1: begin a = 0; b = 0; end
    2: begin a = 1; b = 1; end
    3: b = 1;  // a is not assigned here
endcase
```

In general, use full_case when you know that all possible branches of the case statement have been enumerated, or at least all branches that can occur. If all branches that can occur are enumerated, the logic generated from the case statement performs the same function as the simulated circuit. If a case condition is not fully enumerated, the generated logic and the simulation are not the same.

Note:

You do not need the full_case directive if you have a default branch or you enumerate all possible branches in a case statement, because HDL Compiler assumes that the case statement is full_case.

state_vector Directive

The `// synopsys state_vector` directive labels a variable in a Verilog description as the state vector of an equivalent finite state machine.

The syntax for the `state_vector` directive is

```
// synopsys state_vector vector_name
```

or

```
/* synopsys state_vector vector_name */
```

The *vector_name* variable is the name chosen as a state vector. This declaration allows Synopsys Design Compiler to extract the labeled state vector from the Verilog description. Used with the `enum` directive, described in the next section, the `state_vector` directive allows you to define the state vector of a finite state machine (and its encodings) from a Verilog description. Example 9-9 shows one way to use the `state_vector` directive.

Caution!

Do not define two `state_vector` directives in one module. Although Design Compiler does not issue an error message, it recognizes only the first `state_vector` directive and ignores the second.

Example 9-9 // synopsys state_vector Example

```
reg [1:0] state, next_state;
// synopsys state_vector state

always @ (state or in) begin
     case (state) // synopsys full_case
          0: begin
               out = 3;
               next_state = 1;
               end
          1: begin
               out = 2;
               next_state = 2;
               end
          2: begin
               out = 1;
               next_state = 3;
               end
          3: begin
               out = 0
               if (in)
               next_state = 0;
               else
                    next_state = 3;
          endcase
     end

     always @ (posedge clock)
          state = next_state;
```

Note:

The `state_vector` directive works only with inferred flip-flops. You can also define the state vector and its encodings if you read in a state machine with instantiated flip-flops in HDL format and use embedded dc_shell scripts.

enum Directive

The `// synopsys enum` directive is designed for use with the Verilog parameter definition statement to specify state machine encodings. When a variable is marked as a `state_vector` (see “state_vector Directive” on page 9-13) and it is declared as an `enum`, the Synopsys HDL Compiler uses the enum values and names for the states of an extracted state machine.

The syntax of the `enum` directive is

```
// synopsys enum enum_name
```

or

```
/* synopsys enum enum_name */
```

Example 9-10 shows the declaration of an enumeration of type colors that is 3 bits wide and has the enumeration literals red, green, blue, and cyan with the values shown.

Example 9-10 Enumeration of Type Colors

```
parameter [2:0] // synopsys enum colors
red = 3’b000, green = 3’b001, blue = 3’b010, cyan = 3’b011;
```

The enumeration must include a size (bit-width) specification. Example 9-11 shows an invalid `enum` declaration.

Example 9-11 Invalid enum Declaration

```
parameter /* synopsys enum colors */
red = 3’b000, green = 1;
// [2:0] required
```

Example 9-12 shows a register, a wire, and an input port with the declared type of colors. In each of the following declarations, the array bounds must match those of the enumeration declaration. If you use different bounds, synthesis might not agree with simulation behavior.

Example 9-12 More enum Type Declarations

```
reg   [2:0]  /* synopsys enum colors */ counter;
wire  [2:0]  /* synopsys enum colors */ peri_bus;
input [2:0]  /* synopsys enum colors */ input_port;
```

Even though you declare a variable to be of type `enum`, it can still be assigned a bit value that is not one of the enumeration values in the definition. Example 9-13 relates to Example 9-12 and shows an invalid encoding for colors.

Example 9-13 Invalid Bit Value Encoding for Colors

```
counter = 3'b111;
```

Because `111` is not in the definition for colors, it is not a valid encoding. HDL Compiler accepts this encoding, because it is valid Verilog code, but Design Compiler recognizes this assignment as an invalid encoding and ignores it.

You can use enumeration literals just like constants, as shown in Example 9-14.

Example 9-14 Enumeration Literals Used as Constants

```
if (input_port == blue)
    counter = red;
```

You can also use enumeration with the `state_vector` directive. Example 9-15 shows how the `state_vector` variable is tagged by use of enumeration.

Example 9-15 Finite State Machine With // synopsys enum and // synopsys state_vector

```
// This finite-state machine (Mealy type) reads 1 bit
// per cycle and detects 3 or more consecutive 1s.

module enum2_V(signal, clock, detect);
input signal, clock;
output detect;
reg detect;

// Declare the symbolic names for states
parameter [1:0]//synopsys enum state_info
    NO_ONES = 2'h0,
    ONE_ONE = 2'h1,
    TWO_ONES = 2'h2,
    AT_LEAST_THREE_ONES = 2'h3;

// Declare current state and next state variables.
reg [1:0] /* synopsys enum state_info */   cs;
reg [1:0] /* synopsys enum state_info */   ns;

// synopsys state_vector cs

always @ (cs or signal)

    begin
        detect = 0;// default values
        if (signal == 0)
            ns = NO_ONES;
        else
            case (cs)              // synopsys full_case
                NO_ONES: ns = ONE_ONE;
                ONE_ONE: ns = TWO_ONES;
                TWO_ONES,
                AT_LEAST_THREE_ONES:
                    begin
                        ns = AT_LEAST_THREE_ONES;
                        detect = 1;
                    end
            endcase
    end
always @ (posedge clock) begin
    cs = ns;
end
endmodule
```

Enumerated types are designed to be used as whole entities. This design allows Design Compiler to rebind the encodings of an enumerated type more easily. You cannot select a bit or a part from a variable that has been given an enumerated type. If you do, the overall behavior of your design changes when Design Compiler changes the original encoding. Example 9-16 shows an unsupported bit-select.

Example 9-16 Unsupported Bit-Select From Enumerated Type

```
parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;

assign high_bit = state[2];// not supported
```

Because you cannot access individual bits of an enumerated type, you cannot use component instantiation to hook up single-bit flip-flops or three-states. Example 9-17 shows an example of this type of unsupported bit-select.

Example 9-17 Unsupported Bit-Select (With Component Instantiation) From Enumerated Type

```
DFF ff0 ( next_state[0], clk, state[0] );
DFF ff1 ( next_state[1], clk, state[1] );
DFF ff2 ( next_state[2], clk, state[2] );
```

To create flip-flops and three-states for `enum` values, you must imply them with the posedge construct or the literal `z`, as shown in Example 9-18.

Example 9-18 Using Inference With Enumerated Types

```
parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;

parameter [1:0] /* synopsys enum outputs */
    DONE = 2'd0, PROCESSING = 2'd1, IDLE = 2'd2;
reg [1:0] /* synopsys enum outputs */ out, triout;

always @ (posedge clk) state = next_state;
assign triout = trienable ? out : 'bz;
```

If you use the constructs shown in Example 9-18, you can change the enumeration encodings by changing the `parameter` and `reg` declarations, as shown in Example 9-19. You can also allow HDL Compiler to change the encodings.

Example 9-19 Changing the Enumeration Encoding

```
parameter [3:0] /* synopsys enum states */
    s0 = 4'd0, s1 = 4'd10, s2 = 4'd15, s3 = 4'd5,
    s4 = 4'd2, s5 = 4'd4, s6 = 4'd6, s7 = 4'd8;
reg [3:0] /* synopsys enum states */ state, next_state;

parameter [1:0] /* synopsys enum outputs */
    DONE = 2'd3, PROCESSING = 2'd1, IDLE = 2'd0;
reg [1:0] /* synopsys enum outputs */ out, triout;

always @ (posedge clk) state = next_state;
assign triout = trienable ? out : 'bz;
```

If you must select individual bits of an enumerated type, you can declare a temporary variable of the same size as the enumerated type. Assign the enumerated type to the variable, then select individual bits of the temporary variable. Example 9-20 shows how this is done.

Example 9-20 Supported Bit-Select From Enumerated Type

```
parameter [2:0] /* synopsys enum states */
    s0 = 3'd0, s1 = 3'd1, s2 = 3'd2, s3 = 3'd3,
    s4 = 3'd4, s5 = 3'd5, s6 = 3'd6, s7 = 3'd7;
reg [2:0] /* synopsys enum states */ state, next_state;
wire [2:0] temporary;

assign temporary = state;
assign high_bit = temporary[2]; //supported
```

Note:

Selecting individual bits from an enumerated type is not recommended.

If you declare a port as a `reg` and as an enumerated type, you must declare the enumeration when you declare the port. Example 9-21 shows the declaration of the enumeration.

Example 9-21 Enumerated Type Declaration for a Port

```
module good_example (a,b);

parameter [1:0] /* synopsys enum colors */
    green = 2'b00, white = 2'b11;
input a;
output [1:0] /* synopsys enum colors */ b;
reg [1:0] b;
.
.
endmodule
```

Example 9-22 shows the wrong way to declare a port as an enumerated type, because the `enumerated type` declaration appears with the `reg` declaration instead of with the `output port` declaration. This code does not export enumeration information to Design Compiler.

Example 9-22 Incorrect Enumerated Type Declaration for a Port

```
module bad_example (a,b);

parameter [1:0] /* synopsys enum colors */
    green = 2'b00, white = 2'b11;
input a;
output [1:0] b;
reg [1:0] /* synopsys enum colors */ b;
.
.
endmodule
```

template Directive

The `// synopsys template` directive overrides the setting of the `hdlin_auto_save_templates` variable. If you use this directive and your design contains parameters, the design is archived as a template. Example 9-23 shows how to use the directive.

Example 9-23 // synopsys template Directive in a Design With a Parameter

```
module template (a, b, c);
input a, b, c;
// synopsys template
parameter width = 8;
.
.
.
endmodule
```

See “Module Instantiations” on page 3-16 for more information.

Embedding Constraints and Attributes

Constraints and attributes, usually entered at the dc_shell prompt, can be embedded in your Verilog source code. Prefix the usual constraint or attribute statement with the Verilog comment characters `//`, and delimit the embedded statements with the compiler directives `// synopsys dc_script_begin` and `// synopsys dc_script_end`. The method is shown in Example 9-24.

Example 9-24 Embedding Constraints and Attributes With // Delimiters

```
...
// synopsys dc_script_begin
// max_area 0.0
// set_drive -rise 1 port_b
// max_delay 0.0 port_z
// synopsys dc_script_end
...
```

Constraints and attributes as shown in Example 9-25 can also be delimited with the characters `/*` and `*/`. When you use these delimiters, the `// synopsys dc_script_end` comment is not required or valid, because the attributes or constraints are terminated by `*/`.

Example 9-25 Embedding Constraints and Attributes With / and */ Delimiters*

```
/* synopsys dc_script_begin
   max_area 10.0
   max_delay 0.0 port_z
*/
```

The dc_shell script interprets the statements embedded between the `// synopsys dc_script_begin` and the `// synopsys dc_script_end` directives. If you want to comment out part of your dc_shell script, use the convention for comments that dc_shell uses.

Limitations on the Scope of Constraints and Attributes

The following limitations apply to the use of constraints and attributes in your design:

- Constraints and attributes declared outside a module apply to all subsequent modules declared in the file.
- Constraints and attributes declared inside a module apply only to the enclosing module.
- Any dc_shell scripts embedded in functions apply to the whole module.
- Include in your dc_shell script only commands that set constraints and attributes. Do not use action commands such as `compile`, `gen`, and `report`.
- The constraints or attributes set in the embedded script go into effect after the `read` command is executed. Therefore, variables that affect the read process itself are not in effect before the read. Thus, if you set the variable `hdlin_no_latches = true` in the embedded script, this variable does not influence latch inference in the current read.
- dc_shell performs error checking after the `read` command finishes. Syntactic and semantic errors in dc_shell strings are reported at this time.

Component Implication

In Verilog, you cannot instantiate modules in behavioral code. To include an embedded netlist in your behavioral code, use the directives `// synopsys map_to_module` and `// synopsys return_port_name` for HDL Compiler to recognize the netlist as a function being implemented by another module. When this subprogram is invoked in the behavioral code, HDL Compiler instantiates the module (see Example 9-26 on page 9-25).

The first directive, `// synopsys map_to_module`, flags a function for implementation as a distinct component. The syntax is

```
// synopsys map_to_module modulename
```

The second directive, `// synopsys return_port_name`, identifies a return port (functions in Verilog do not have output ports). To instantiate the function as a component, the return port must have a name. The syntax is

```
// synopsys return_port_name portname
```

Note:

Remember that if you add a `map_to_module` directive to a function, the contents of the function are parsed and ignored whereas the indicated module is instantiated. Ensure that the functionality of the module instantiated in this way and the function it replaces are the same; otherwise, pre-synthesis and post-synthesis simulation do not match.

Example 9-26 illustrates the `map_to_module` and `return_port_name` directives.

Example 9-26 Component Implication

```
module mux_inst (a, b, c, d, e);
input a, b, c, d;
output e;
              function mux_func;
         // synopsys map_to_module mux_module
         // synopsys return_port_name mux_ret
         input in1, in2, cntrl;
              /*
              ** the contents of this function are ignored for
              ** synthesis, but the behavior of this function
              ** must match the behavior of mux_module for
              ** simulation purposes
              */
              begin
              if (cntrl) mux_func = in1;
              else mux_func = in2;
              end

         endfunction

assign e = a & mux_func (b, c, d);
// this function call actually instantiates component (module) mux_module

endmodule

module mux_module (in1, in2, cntrl, mux_ret);
input in1, in2, cntrl;
output mux_ret;

and and2_0 (wire1, in1, cntrl);
not not1 (not_cntrl, cntrl);
and and2_1 (wire2, in2, not_cntrl);
or or2 (mux_ret, wire1, wire2);

endmodule
```