Warp™

Verilog Reference Guide

Synthesis Tool for PSoC® Creator™

Document # 001-48352 Rev. *D

Cypress Semiconductor
198 Champion Court
San Jose, CA 95134-1709
Phone (USA): 800.858.1810
Phone (Intnl): 408.943.2600
http://www.cypress.com
# Contents

## 1. Introduction

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Conventions</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>References</td>
<td>5</td>
</tr>
<tr>
<td>1.3</td>
<td>Revision History</td>
<td>6</td>
</tr>
</tbody>
</table>

## 2. Verilog Language Constructs

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Identifiers</td>
<td>7</td>
</tr>
<tr>
<td>2.1.1</td>
<td>Reserved Words</td>
<td>8</td>
</tr>
<tr>
<td>2.1.2</td>
<td>Comments</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>Constants</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Data types</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1</td>
<td>Nets</td>
<td>9</td>
</tr>
<tr>
<td>2.3.2</td>
<td>Registers</td>
<td>10</td>
</tr>
<tr>
<td>2.3.3</td>
<td>Parameters</td>
<td>11</td>
</tr>
<tr>
<td>2.3.3.1</td>
<td>Support for proper parameters</td>
<td>11</td>
</tr>
<tr>
<td>2.3.3.2</td>
<td>Support for localparams</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3.3</td>
<td>Module Parameters</td>
<td>12</td>
</tr>
<tr>
<td>2.3.3.4</td>
<td>Support for named parameter passing</td>
<td>13</td>
</tr>
<tr>
<td>2.4</td>
<td>Operators</td>
<td>13</td>
</tr>
<tr>
<td>2.4.1</td>
<td>Arithmetic operators</td>
<td>13</td>
</tr>
<tr>
<td>2.4.2</td>
<td>Shift operators</td>
<td>14</td>
</tr>
<tr>
<td>2.4.3</td>
<td>Relational operators</td>
<td>14</td>
</tr>
<tr>
<td>2.4.4</td>
<td>Equality operators</td>
<td>14</td>
</tr>
<tr>
<td>2.4.5</td>
<td>Bit-wise operators</td>
<td>15</td>
</tr>
<tr>
<td>2.4.6</td>
<td>Reduction operators</td>
<td>15</td>
</tr>
<tr>
<td>2.4.7</td>
<td>Logical operators</td>
<td>15</td>
</tr>
<tr>
<td>2.4.8</td>
<td>Conditional operators</td>
<td>16</td>
</tr>
<tr>
<td>2.4.9</td>
<td>Concatenation</td>
<td>16</td>
</tr>
<tr>
<td>2.5</td>
<td>Operands</td>
<td>17</td>
</tr>
<tr>
<td>2.6</td>
<td>Modules</td>
<td>17</td>
</tr>
<tr>
<td>2.6.1</td>
<td>Module Syntax</td>
<td>17</td>
</tr>
<tr>
<td>2.6.2</td>
<td>Top Level Module</td>
<td>18</td>
</tr>
<tr>
<td>2.6.3</td>
<td>Module Connection</td>
<td>19</td>
</tr>
<tr>
<td>2.7</td>
<td>Primitive gates</td>
<td>20</td>
</tr>
<tr>
<td>2.8</td>
<td>Continuous assignments</td>
<td>21</td>
</tr>
<tr>
<td>2.9</td>
<td>Behavioral Modeling</td>
<td>22</td>
</tr>
<tr>
<td>2.9.1</td>
<td>Procedural assignment</td>
<td>22</td>
</tr>
<tr>
<td>2.9.2</td>
<td>Block statements</td>
<td>23</td>
</tr>
<tr>
<td>2.9.3</td>
<td>If...else statements</td>
<td>23</td>
</tr>
<tr>
<td>2.9.4</td>
<td>Case statements</td>
<td>24</td>
</tr>
<tr>
<td>2.9.5</td>
<td>Loopying statements</td>
<td>26</td>
</tr>
<tr>
<td>2.9.6</td>
<td>Generate Statements</td>
<td>27</td>
</tr>
<tr>
<td>2.9.6.1</td>
<td>generate/endgenerate</td>
<td>27</td>
</tr>
</tbody>
</table>
1. Introduction

The Warp synthesis tool is a Verilog compiler used by PSoC Creator for designing with PSoC devices. Warp accepts Verilog text input and then synthesizes and optimizes the design for the target hardware. Warp then outputs a file for programming the device. Warp operates in the background. Most users will not interact with the program directly.

This guide discusses the fundamental elements of Verilog HDL implemented in Warp. The first chapter covers the Verilog language constructs supported in Warp. The second chapter covers register and tri-state synthesis implemented in Warp. The last chapter includes some Verilog design examples. The appendix contains a list of Verilog reserved words.

1.1 Conventions

The following table lists the conventions used throughout this guide:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Courier New</td>
<td>Displays file locations and source code: C:\ _cd\icc, user entered text</td>
</tr>
<tr>
<td>Italic</td>
<td>Displays file names and reference documentation: sourcefile.hex</td>
</tr>
<tr>
<td>[bracketed, bold]</td>
<td>Displays keyboard commands in procedures: [Enter] or [Ctrl] [C]</td>
</tr>
<tr>
<td>File &gt; New Project</td>
<td>Represents menu paths: File &gt; New Project &gt; Clone</td>
</tr>
<tr>
<td>Bold</td>
<td>Displays commands, menu paths and selections, and icon names in procedures: Click the Debugger icon, and then click Next.</td>
</tr>
<tr>
<td>Text in gray boxes</td>
<td>Displays cautions or functionality unique to PSoC Creator or the PSoC device.</td>
</tr>
</tbody>
</table>

1.2 References

This guide is one of a set of documents pertaining to PSoC Creator and PSoC3/5. Refer to the following other documents as needed:
- PSoC Creator Help
- PSoC Creator Component Author Guide
- PSoC Creator Customization API Reference Guide
- PSoC Device Technical Reference Manuals (TRMs)
1.3 Revision History

<table>
<thead>
<tr>
<th>Revision</th>
<th>Date</th>
<th>By</th>
<th>Description of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>12/4/07</td>
<td>CKF</td>
<td>New document.</td>
</tr>
<tr>
<td>*A</td>
<td>5/5/09</td>
<td>CKF</td>
<td>Changes to example designs and product name.</td>
</tr>
<tr>
<td>*B</td>
<td>9/22/10</td>
<td>CKF</td>
<td>Updates to Introductions and explanations for design examples.</td>
</tr>
<tr>
<td>*C</td>
<td>5/24/13</td>
<td>CKF</td>
<td>Update to section 2.11.</td>
</tr>
<tr>
<td>*D</td>
<td>7/22/14</td>
<td>CKF</td>
<td>Added note that multiple non-blocking assignments are not supported to section 2.9.1.</td>
</tr>
</tbody>
</table>

**Distribution:** None  
**Posting:** None
2. Verilog Language Constructs

This chapter includes the following sections:

- Identifiers
- Constants
- Data types
- Operators
- Operands
- Modules
- Primitive gates
- Continuous assignments
- Behavioral Modeling
- Timing controls
- Structured procedures
- Compiler directives

2.1 Identifiers

An identifier in Verilog is composed of a sequence of letters, digits, dollar signs($), and underscore characters (_). Identifiers are case sensitive. In Warp, the first character in an identifier must be a letter. If an identifier starts with an underscore, Warp currently renames such identifiers by adding the prefix 'warp'.

An escaped identifier starts with the back-slash character (\) and ends with a white space (space, tab or new line). The leading back-slash and the terminating space are not treated as part of the identifier. Warp does not support the escaped identifiers.

The following are legal identifiers in Verilog:

```
clock1
dataA
_reset
```

The following are not legal identifiers in Verilog:

```
3reset // identifier cannot start with a digit
module // a keyword cannot be used as an identifier
```

An identifier that starts with a $ is a system task or system function. Warp ignores all system tasks and system function identifiers.
2.1.1 Reserved Words

The Verilog language has a set of reserved words, called keywords, that cannot be used as identifiers. Refer to Verilog Reserved Words on page 43 for a list of keywords.

2.1.2 Comments

The Verilog language has both line comments and block comments. A line comment begins with two consecutive forward slashes (//) and extends to the end of the line. A block comment starts with /* and ends with */. Comments can appear anywhere within a Verilog description. Block comments cannot be nested.

// this is a line comment

/*
   this is a block comment */

2.2 Constants

In Verilog, constant numbers can be specified as integer constants. The integer constants can be specified in two forms: a simple decimal number specified as a sequence of digits (0-9); a sized constant which is represented as a based number. A sized constant is composed of three tokens: an optional size, a single quote followed by a base format character (‘d for decimal, ‘b for binary, ‘o for octal and ‘h for hexa decimal) and a sequence of digits representing the value.

A decimal base number is composed of a sequence of 0 through 9 digits.

A binary base number is composed of a sequence of x, z, 0 and 1.

An octal base number is composed of a sequence of x, z, and 0 through 7 digits.

A hexadecimal base number is composed of a sequence of x, z, 0 through 9 digits and a through f characters.

The base format character (d, b, o, h) is not case sensitive.

The alphabetic digits in the base number (x, z, a through f) are not case sensitive.

An x represents the unknown value and z represents the high-impedance value.

Simple decimal numbers without the size and the base format are treated as signed integers and decimal numbers with the base format are treated as unsigned integers.

A + or - operator preceding the size constant is the sign for the constant number.

A + or - between the base format and the number is illegal.

The underscore character can be used in the constant to improve the readability of long numbers. This character is ignored by Warp.

Default size of an un-sized constant is 32 bits.

The following are legal Verilog constants:

10    // is a decimal number
-1204 // is a signed decimal number
2’b1  // is a sized binary number stored as two bits (01)
’h a7fx // is an un-sized hexadecimal number
9’o17 // is a sized octal number stored as 000001111
Verilog Language Constructs

‘b0101_1110 /* is a binary number equal to 01011110. The underscore character is ignored. */

The following are illegal Verilog constants:
2’b-1 /* - sign between the base format character b and the base number 1 */

String constants are treated as unsigned integer constants represented by a sequence of 8-bit ASCII values, with each 8-bit ASCII value representing one character.

Real constants are not supported.

2.3 Data types

Data types in Verilog belong to one of three classes: net, reg and parameter.

2.3.1 Nets

The net data type is used to represent a physical connection between different hardware blocks. A net can be driven by a continuous assignment statement or an output of a gate or module. A net data type will not store its value.

A net can be one of the following types:
- wire
- tri
- supply0
- supply1

A wire/tri net type is used to connect different hardware elements. A tri net type is identical to the wire net type both in the syntax and functionality. Two names are provided in order to distinguish the purpose of the net in the design and hence to enhance the readability. A wire net type is used for nets that are driven by a single gate or a continuous assignment. Nets driven by multiple drivers are declared as tri net type.

supply0 and supply1 nets are used to model the power supplies. supply0 represents logic 0 (ground) and supply1 represents logic1 (power).

The following Verilog net types are not supported by Warp.
- tri0
- tri1
- wand
- triand
- wor
- trior
- trireg

Examples:
wire a ;       // a is a wire type net
tri t ;        // t is a tri-state type net
supply0 gnd ;  // gnd is connected to logic 0 (ground)
supply1 vcc ;  // vcc is connected to logic 1 (power)

A net can be a scalar or a vector. Scalar nets represent individual signals and vector nets represent bus signals. By default, all nets are treated as scalars. Vector nets are declared by specifying the
range of bits after the net type. The left-hand value in the range specifies the most-significant-bit and the right-hand value in the range specifies the least-significant-bit.

Example:
```verbatim
wire [7:0] dataA ; /* dataA is an 8 bit vector.
    bit0 is LSB and bit 7 is MSB */
wire [0:7] dataB ; /* dataB is an 8 bit vector.
    bit0 is MSB and bit 7 is LSB */
```

In Verilog, the strength of a net is defined using a combination of two strengths: drive strength (weak1, weak0, highz0, highz1, pullo, pull1, pullup and pulldown) and charge strength (small, medium, large). Warp ignores the strengths associated with any net.

### 2.3.2 Registers

Register data types are used as variables. A register data type stores its value until another assignment changes the register. The register data type is declared using the keyword `reg`. Registers can be assigned only in `always` blocks, functions and tasks.

`integer` is a register data type used for manipulating quantities that are not regarded as hardware registers. In Warp, integers are treated as 32 bit signed quantities and `reg` datatypes are treated as unsigned quantities by default unless specified to be signed quantities.

Examples:
```verbatim
reg a, b ;   // a, b are two register variables
integer i ;  // i is an integer variable
```

Register variables also can be declared as scalar or vector. Vector registers are declared by specifying the range of bits after the `reg` or `integer` keyword. The left-hand value in the range specifies the most-significant-bit and the right-hand value in the range specifies the least-significant-bit.

Examples:
```verbatim
reg[7:0] dataA; /* dataA is an 8 bit vector.
    bit0 is LSB and bit 7 is MSB */
reg [0:7] dataB; /* dataB is an 8 bit vector.
    bit0 is MSB and bit 7 is LSB */
integer a;       // a is an integer (always 32 bit quantity)
```

Register declaration does not imply a flip-flop or a latch. Warp does not support multiple drivers for register and integer variables.

The `time`, `real` and `realtime` declarations are not supported in Warp.

Ranges and arrays for integers are not supported by Warp.

Arrays of register data types (memories) are also not supported in Warp.

Example:
```verbatim
// The following declarations are illegal in Warp
time t;
real f;
realtime rt;
integer [3:0] x; // integer array
reg [7:0] mem [0:63];
```
2.3.3 Parameters

Unlike nets and registers, parameters are constants. The value of a parameter cannot be modified during run time. Parameters are used to write parameterized models. Parameters are declared using the parameter keyword as follows:

```
parameter parameter_assignment {,parameter_assignments}
```

```
parameter_assignment ::= parameter_identifier = constant_expression
```

Examples:

```
parameter lsb = 0, msb = 3 ;  // lsb and msb are parameters
reg [msb:lsb] x ;             // x is a vector with range 3:0
parameter tPD = 7 ;           /* parameter tPD is used to represent
propagation delay */
```

**Note** If the given parameter is not intended to be modified by an upper level module, using the `define` compiler directive (see `define on page 32`) is recommended.

2.3.3.1 Support for proper parameters

Parameters are treated as arbitrary length bit-strings. The length of a parameter is unconstrained by default but can be constrained by the user, for example:

```
parameter     unconst_param = 12;  /* Unconstrained - size is
// determined by its usage. */
```

```
parameter [3:0] const_param = 12;  // Constrained - uses only 4 bits.
```

A parameter can become a signed quantity depending on the parameter override. In most cases, there is very little difference between signed and unsigned, but it can have an impact in magnitude comparison operators (and *, /). However, the overrides only have an impact if the parameter is declared plainly. You can specify a parameter to be signed or to be of an integer type, in which case it is signed regardless of the override.

Example of a plain parameter:

```
parameter P = 23 ;
```

The true type and size of the parameter is determined by the actual value of the parameter at runtime (elaboration time). Warp does not automatically handle the size or the signed/unsigned nature of the parameter. If the parameter does not have a size constraint or a type (signed/unsigned/integer/etc.) designation, Warp will use the defaults. If you want something else, you must be specific and assign it a size and type (as shown the following examples).

Examples of a fully-specified parameter:

```
parameter [3:0] P = 3;    // unsigned 4-bit quantity
parameter signed P = 3;   /* signed quantity whose size
is determined later */
```

```
parameter signed [3:0] P = 3;  // signed 4-bit quantity
parameter integer P = 3;     // signed 32-bit quantity
```

Since the type/size of these params are defined by their declaration, the expressions that such params get involved in will be handled correctly. When an override is done, it is properly type-converted in all cases. Cypress recommends this kind of declaration for all parameters that are expected to be a fixed size/type.
Use this kind of a parameter when you are using params for arbitrary strings and use them only in
unsigned context (for instance in equal/notequal comparisons, addition, subtraction, etc.).

The signed/unsigned nature of the parameter is determined by its initial value (also applies to
localparams).

2.3.3.2 Support for localparams

You can use this to create local-parameters that cannot be overwritten by defparams.

Example

```vLanguages
parameter x = 4 ;
localparam x2 = (x / 2);
```

Warp allows parameters to appear on the right-hand-side of another parameter definition

```vLanguages
parameter x = 4 ;
parameter x2 = (x / 2);
```

This will work as expected. If you override just ‘x’ then x2 will appropriately be calculated using the ‘x’
new value. If you override both ‘x’ and ‘x2’, both will be overridden according to spec. and x2 may no
longer have the relationship intended. This is where localparam’s should be used instead.

2.3.3.3 Module Parameters

The parameters declared inside an instantiated module can be modified during instantiation
(parameter value assignment) or by using defparam construct. In Warp, the defparam can be only
used to modify the parameters of immediate instances.

Example:

```vLanguages
module dreg(clk, d, q) ;  // define a parameterizable reg.
parameter range = 4 ;
input [range-1:0] d ;
output [range-1:0] q ;
...
endmodule
dreg #(8) inst_1(...) ;   // instantiates 8 bit dreg
dreg #(16) inst_2(...) ;  // instantiates 16 bit reg
```

Parameter values in a module can also be re-defined by using defparam construct. At any level of
the design, Warp allows the re-definition of parameters of the modules instantiated at that level only.
More than one levels of hierarchical path names are not currently supported.

Example:

```vLanguages
defparam inst_1.range = 16 ;
```

In the absence of an explicit declaration of a net or a register, statements for gate and module
instantiations shall assume an implicit net declaration. These implicitly declared nets shall be treated
as a scalar of type wire.

Module declaration including parameters and ports

```vLanguages
module Nbit_adder
    #(parameter SIZE = 3, JUNK = SIZE+3) (output co, output [SIZE-1:0] sum, input [SIZE-1:0] a, b, input ci);
...
endmodule
```
This is more concise than traditional module declaration. Note the two unexpected rules that Verilog enforces (and consequently, so does Warp). If you use the new style of declaring a module:

- You cannot redefine or define a port or its type, which is preferred.
- It will still allow you to define a new parameter which can be overridden using a `defparam`. Try to avoid declaring additional parameters in the body of the module. The body should have `localparams` only.

### 2.3.3.4 Support for named parameter passing

The following is better than using `defparams`:
```
Nbit_adder #( .SIZE(5) ) ul (.co(co1), .sum(sum), .a(a), .b(b), .ci(ci));
```

### 2.4 Operators

The following table lists the Verilog operators that Warp supports.

<table>
<thead>
<tr>
<th>Arithmetic operators</th>
<th>*, +, -, /, %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shift operators</td>
<td>&lt;&lt;, &gt;&gt;</td>
</tr>
<tr>
<td>Relational operators</td>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
</tr>
<tr>
<td>Equality operators</td>
<td>==, !=</td>
</tr>
<tr>
<td>Bit-wise operators</td>
<td>¬, &amp;,</td>
</tr>
<tr>
<td>Reduction operators</td>
<td>&amp;,</td>
</tr>
<tr>
<td>Logical operators</td>
<td>!, &amp;&amp;,</td>
</tr>
<tr>
<td>Conditional operator</td>
<td>?:</td>
</tr>
<tr>
<td>Event or</td>
<td>or</td>
</tr>
<tr>
<td>Concatenation</td>
<td>{}, {{}}</td>
</tr>
</tbody>
</table>

### 2.4.1 Arithmetic operators

In Verilog, the `+` and `-` operators perform addition and subtraction, respectively. The `*` operator performs multiplication. Division and modulus operators (`/`, `%`) are supported when both operands are constants, or when the second operand is a power of 2.

Use of the division and modulus operators may result in an error if the operand constant criteria described above is not met.

Examples:
```
reg [3:0] sum, diff, mult, div, mod;
reg [1:0] a, b;
parameter size1 = 16, size2 = 4;
sum = a + b;
diff = a - b;
mult = a * b;
div = size1/size2;
mod = size1%size2;
```
2.4.2 Shift operators

The shift operators are binary operators. The left shift operator \(\ll\) shifts the bits in the left operand by the number of bit positions specified by the right operand. The right shift operator \(\gg\) shifts the bits in the left operand by the number of bit positions specified by the right operand. Both shift operators fill the vacated bits with zeros. These shift operators perform logical shift.

Examples:

```verilog
wire [7:0] a, b;
parameter shift = 4;
assign a = (b << shift); // a is assigned to b[3],..,b[0],0,0,0,0
assign a = b >> shift;  // a is assigned to 0,0,0,0,b[7],..,b[4]
```

2.4.3 Relational operators

Relational operators perform comparison between two operands and return 1'b0 if the specified relation is false or 1'b1 if the specified relation is true.

When the operands used in a relational expression are not of the same size, the smaller operand will be zero filled on the most-significant bit side to extend to the size of the larger operand.

Examples:

```verilog
reg a, b, c;
a < b;  /* evaluates to 1 if the value of a is less than the value of b, 0 otherwise. */
a <= b; /* evaluates to 1 if a is less than or equal to b, 0 otherwise. */
a >= b; /* evaluates to 1 if a is greater than or equal to b, 0 otherwise. */
a > b;  /* evaluates to 1 if a is greater than b, 0 otherwise. */
```

2.4.4 Equality operators

The equality operators \(==\) and \(!=\) compare each bit of the left operand with corresponding bit of the right operand. The equal operator \(==\) evaluates to true if the operands have the same value. The not equal operator \(!=\) evaluates to true if the operands have different values. Zero filling is done if the operands are of different size.

Case equal operators \(===\) and \(!==\) are not supported by Warp.

Examples:

```verilog
if( a == b)
begin
... /* this block is executed if a and b have the same value */
end
else // (a != b)
begin
... /* this block is executed if a and b have different values */
end
```
2.4.5 Bit-wise operators

Bit-wise operators perform bit-wise manipulations on the operands. The result of a bit-wise operation is obtained by performing the operation on each bit of the left operand with the corresponding bit of the right operand. When the operands are of unequal bit length, the shorter operand is zero-filled in the most significant bit positions.

Examples:
```verilog
reg [1:0] a, b, c;
a = 2'b1; b = 2'b3;
c = a & b;  // bit-wise and. c gets a value "01"
c = a | b;  // bit-wise or. c gets a value "11"
c = ~a ;   // bit-wise negation. c gets a value "10"
c = a ^ b ; // bit-wise xor. c gets a value "10"
c = a ^~ b ;  // bit-wise xnor. c gets a value "01"
```

2.4.6 Reduction operators

Reduction operators are unary operators and perform a bit-wise operation on each bit of an operand and give a 1-bit result.

Examples:
```verilog
reg [2:0] a;
reg c;
a = 2'b1;
c = &a;   // c gets a value 0
```

2.4.7 Logical operators

Logical operators are used to perform a true/false test on an expression. The logical operators return a true (1'b1) or false (1'b0).

Logical not (!) is used to test if a variable is a true or false.
Logical and (&&) returns true if both of its operands evaluate to true.
Logical or (||) returns true if one or both of its operands evaluate to true.

Examples:
```verilog
reg A, B;
reg C, D;
C = A && B;  // C is assigned 1'b1 if both A and B are true
D = A || B;  /* D is assigned 1'b1 if either A or B is true.
              Otherwise D = 1'b0 */
D = !A;       // D gets a value 0 if A is 1 and D gets 1 if A is 0
```
2.4.8 Conditional operators

Conditional operator "?:" is used to write expressions which get different values based on a condition. The usage of the conditional operator is as follows:

\[
\text{conditional_expression ::= expression1 ? expression2 : expression3}
\]

If expression1 evaluates to true (1'b1), then expression2 will be used as the result of the conditional_expression. If expression1 evaluates to false (1'b0), then expression3 will be used as the result of the conditional_expression.

Example:

```verilog
wire a, b, c, d;
assign d = a ? b : c;
```

2.4.9 Concatenation

Concatenation operator provides a means to combine together bits of two or more expressions. Concatenation is achieved by enclosing the list of expressions within the concatenation operator `{}`.

Example:

```verilog
reg [7:0] a;
reg [3:0] b;
reg x1, x2, x3, x4;
a = {b, x1, x2, x3, x4};
```

The assignment above produces the same result as the following assignments.

```verilog
a[7] = b[3];
a[6] = b[2];
a[5] = b[1];
a[4] = b[0];
a[3] = x1;
a[2] = x2;
a[1] = x3;
a[0] = x4;
```

Un-sized constants are not allowed within the concatenation operator.

Concatenation can be repeated by using a repetition multiplier. In Warp, the repetition multiplier needs to be a constant.

Example:

```verilog
reg [7:0] a;
reg x1, x2, x3, x4;
reg b;
a = {2{x1, x2, x3, x4}};
```

The above assignment is equivalent to

```verilog
a = {x1, x2, x3, x4, x1, x2, x3, x4};
a = {b{x1, x2}}; // illegal in Warp, b is not constant
```


2.5 Operands

An expression can contain operands of the following types.

- **Constants**: Constant operands must conform to the specification in Section 2.2.
- **Reference to a net or register**: The name of a net or register, when used as an operand, implies all the bits of the net or register. Bit-select and part-select can be used to reference a part of a vector net or register variable.
- **Bit-selects**: Bit-select extracts one bit of a net or register vector. The bit selected should be within the range of the vector.
- **Part-selects**: A part-select is used to access contiguous bits in a vector net or register. The part-select expressions shall be constant-valued.
- **Strings**: String operands are converted to bits, as described in Section 2.2.
- **Function call**: A function call can be used as an operand.
- **Concatenation**: An operand can also be concatenation of the above mentioned operands.

_Warp_ ignores the delay expressions (minimum, typical, maximum).

Examples:

```verilog
wire a, b;
reg [7:0] c, d;
wire [3:0] x;
a + b;            // scalar operands
a + c[1];         // scalar net and bit select (bit 1 of c) operands
c[3:0] + x;       // part-select (c[3]..c[0]) and vector (x) operands
c - d;            // vector operands
a + "01" + 4'b1;  // scalar net, string and constant operands
{a, c};           // concatenation of scalar net and vector reg
```

2.6 Modules

A module in Verilog encapsulates the description of a design. The description of a design can be either, or a combination of the following:

- **behavioral descriptions** – provide a means to define the **behavior** of a circuit in abstract **high level** algorithms, or in terms of **low level** boolean equations.
- **structural descriptions** – define the **structure** of the circuit in terms of components and resemble a net-list that describes a schematic equivalent of the design. Structural descriptions contain hierarchy in which components are defined at different levels.

2.6.1 Module Syntax

A Verilog design consists of one or more modules connected with each other by means of ports. Ports provide a means of connecting various hardware elements. Each port has an associated name and mode (input, output and inout). Module definitions cannot be nested. A module is defined using the following syntax:

```verilog
module <name> (interface_list) ;{ module_item }
endmodule
interface_list ::= port_reference
| {port_reference {, port_reference}}
port_reference ::= port_identifier
| port_identifier [ constant_expression ]
| port_identifier [ msb_constant_expression : lsb_constant_expression ]
```
Verilog Language Constructs

module_item ::= module_item_declaration
    | continuous_assignment
    | gate_instantiation
    | module_instantiation
    | always_statement
module_item_declaration ::= parameter_declaration
    | input_declaration
    | output_declaration
    | inout_declaration
    | net_declaration
    | reg_declaration
    | integer_declaration
    | task_declaration
    | function_declaration

Warp treats the keywords macromodule and module as synonyms.

Example:
// a module definition for a d flip-flop
module my_dff (clk, d, q);
    input clk, d;
    output q;
    wire clk, d;
    reg q;
    always @(posedge clk)
        begin
            q = d;
        end
endmodule

2.6.2 Top Level Module

In Verilog, hierarchical designs are specified by instantiating one or more modules in a top level module. A top level module is a module that is not instantiated by any other module.

The syntax of the module instantiation statement is as follows:
<module_name> [parameter_value_assignment] <instance_name>
module_instance {, module_instance} ;
module_instance ::= instance_identifier ([list_of_module_connections])
list_of_module_connections ::= ordered_port_connection {, ordered_port_connection} | named_port_connection {, named_port_connection}

Example:
/* a 4-bit shift register defined by instantiating 4 d-ffs */
module shift_reg(clk, d, q) ;
    input clk, d;
    output q;
    wire q0, q1, q2 ;
/* module instantiation statements. my_dff is the type of the module instantiated. inst_3, ... inst_0 are the instance names */
my_dff inst_3 (clk, d, q0) ;
my_dff inst_2 (clk, q0, q1) ;
my_dff inst_1 (clk, q1, q2) ;
my_dff inst_0 (clk, q2, q) ;
endmodule

One or more instantiations of the same module can also be specified in a single module instantiation statement. The four instantiation statements in the above example can be combined into one instantiation statement as follows:

my_dff inst_3(clk, d, q0),
inst_2(clk, q0, q1),
inst_1(clk, q1, q2),
inst_0(clk, q2, q) ;

**Note** The range specification in module instantiations (array of instances) is not supported in Warp.

### 2.6.3 Module Connection

A module connection describes the connection between the signals listed in the module instantiation statement and the ports in the module definition. This connection can be specified in two ways: *ordered port association* and *named port association*.

In the case of *ordered port association*, the signals in the instantiation statement should be in the same order as the ports listed in the module definition.

In the case of *named port association*, the port names of instantiated modules are also included in the connection list.

**Example:**

```verilog
my_dff inst_3(clk, d, q0) ;                // ordered connection list.
my_dff inst_3(.d(d), .q(q0), .clk(clk)) ;  /* named association: q0 is
connected to the port q of
my_dff module. */
```

The port expression in the module connection list can be one of the following:

- A simple identifier
- A bit-select of a vector declared within the module
- A part-select of a vector declared within the module
- A concatenation of any of the above
2.7 **Primitive gates**

The Verilog language provides gate level and switch level modeling capability by means of a set of primitive gates. *Warp* supports the following primitive gates:

- `and`
- `nand`
- `or`
- `nor`
- `xor`
- `xnor`
- `buf`
- `not`
- `bufif0`
- `bufif1`
- `notif0`
- `notif1`

The primitive gates `and`, `nand`, `or`, `nor`, `xor`, `xnor` have one output and one or more inputs. The first terminal in the terminal list is the output and all other terminals are inputs.

Examples:

```verilog
and i1 (f, a, b, c);      // 3-input (a, b, c) and gate
and i2 (f, a, b, c, d);  // 4-input (a, b, c, d) and gate
xor i3 (f, a, b);        // 2-input (a, b) xor gate
```

The primitive gates `buf` and `not` have one input and multiple outputs. The last terminal in the terminal list is the input and all other terminals are outputs.

Examples:

```verilog
buf i1 (f1, f2, a);      // 2 output (f1, f2) buf gate
not i2 (x, y, a);        // 2 output (x, y) not gate
```

The primitive gates `bufif0`, `bufif1`, `notif0` and `notif1` model three state drivers. These gates have three terminals. The first terminal is output, the second terminal is data input and the last terminal is control input.

Examples:

```verilog
bufif0 i1 (o1, i1, c1);  /* tri-state buffer with active low enable (c1) */
bufif1 i2 (o2, i2, c2);  /* tri-state buffer with active high enable (c2) */
notif0 i3 (o3, i3, c3);  /* tri-state buffer with inverted output and active low enable(c3) */
notif1 i4 (o4, i4, c4);  /* tri-state buffer with inverted output and active high enable(c4) */
```

*Warp* ignores drive strength and delays specified in gate instantiation statements.
### 2.8 Continuous assignments

Values are assigned to nets by means of a continuous assignment statement. Continuous assignments can be specified either in the net declaration statement or by using an `assign` construct using the following syntax.

```
assign net_assignment {, net_assignment}
net_assignment ::= net_lvalue = expression
net_lvalue ::= net_identifier
| net_identifier [expression]
| net_identifier [msb_const_expression : lsb_const_expression]
| net_concatenation
```

Examples:

```
wire a;
reg b, c;
assign a = b;    // continuous assignment using assign construct
wire d = b + c;  // continuous assignment in the net declaration
```

Continuous assignments drive values onto nets, in a manner similar to the way gates drive nets.

*Warp* ignores the charge strength, drive strength and delay specified in the continuous assignment statements.

The left-hand side of a continuous assignment statement can be one of the following:

- a scalar or vector net
- constant bit-select of a vector net
- constant part-select of a vector net
- concatenation of any of the above

Examples:

```
wire [3:0] dataA;
wire dataB;
wire [2:0] dataC;
wire [4:0] dataD, dataE;
wire [7:0] dataF, dataG;
assign dataE = dataA - dataB;
assign dataD[2:0] = dataC;
assign {dataF, dataG} = ~dataE + dataD;
```
2.9 Behavioral Modeling

2.9.1 Procedural assignment

Procedural assignment statement assigns value to register data type variables. Procedural assignments are used inside procedural flow blocks (always, initial, function and task). The right-hand side of a procedural assignment can be any expression that evaluates to a value. The left-hand side of a procedural assignment can be one of the following:

- a scalar or vector reg data type or a scalar integer register data type
- bit-select of a reg data type
- part-select of a reg data type
- concatenation of the above

In the Verilog language, there are two types of procedural assignment statements:

- **Blocking procedural assignment statement** – blocks the execution of a statement in a sequential block. In other words, the execution of a statement next to a blocking statement is not executed until the execution of the blocking assignment is completed. Blocking assignment is specified using the "=' operator.

- **Nonblocking procedural assignment statement** – does not block the execution of a statement. Nonblocking assignment is specified using the "<=" operator.

It is illegal in **Warp**, to assign value to a register variable using both blocking and nonblocking assignment.

The syntax of these statements is as follows:

```
blocking_assignment ::= reg_lvalue = expression
nonblocking_assignment ::= reg_lvalue <= expression
reg_lvalue ::= reg_identifier
| reg_identifier [expression]
| reg_identifier [msb_const_expression : lsb_const_expression]
| reg_concatenation
```

Examples:

```
// blocking assignment
reg a, b, c;
a = b;
c = a; /* 'c' gets the value 'b' because the above statement is blocking statement. */

// non-blocking assignment
reg a, b, c;
a <= 0;
c <= a; /* 'c' gets the previous value of 'a' (value of a prior to the statement a <= 0.

// illegal procedural assignment in Warp.
reg a, b, c, d;
a <= b + c; // non-blocking assignment to 'a'
a = d; // blocking assignment to 'a'
```

**Note** Nonblocking assignment statements within a function/task are not supported by Warp.

**Note** Warp does not support multiple non-blocking assignment statements. Any such code should be restructured to avoid multiple assignments.
2.9.2 Block statements

Block statements are used to group several statements together so that they act like a single statement. The sequential block is defined by including a set of statements between the keywords: begin and end. The statements inside a sequential block are executed sequentially. The sequential blocks can also have a name (label). A named block can include reg, integer and parameter declarations.

A sequential block statement has the following syntax:

```verilog
begin [:block_name {block_item_declaration}] {statement}
end
```

block_item_declaration ::= parameter_declaration
| reg_declaration
| integer_declaration

Examples:

```verilog
// a sequential block
begin
  a = b;
  if( (c + d) > 1)
    begin
      ...
    end
  else
    begin
      ...
    end
end

// a named sequential block
begin : reset_block
  if( reset)
    ..
  else
    ..
end
```

* Warp does not support parallel block.

2.9.3 If...else statements

The if-else statement selects one or more statements to be executed within a sequential block, based on the value of a condition. For example:

```verilog
if(expression) statement_or_null
[else statement_or_null]
```

A condition is a boolean expression; that is, an expression that resolves to a boolean value. If the condition evaluates to true, the sequence of statements in the if block, if present, are executed. If the condition evaluates to false, the sequence of statements following the else keyword, if present, are executed.

Example:

```verilog
/* q is assigned a value 0 if reset is logic 1.
   otherwise q is assigned d */
if( reset) // reset == 1
```
```verilog
q = 0;
else
    q = d;

// multiple if-else statements
if( p_state == 2'b1)
    next_state = 2'11;
else if( p_state == 2'b11)
    next_state = 2'00;
else
    next_state = p_state;
```

### 2.9.4 Case statements

The case statement is a multi-way decision statement that branches to one or more statements based on the value of an expression.

```
    case (expression) case_item {case_item} endcase
```

**Example:**

// a multiplexer implemented using a case statement

```verilog
'define selA 4'd1
'define selB 4'd2
'define selC 4'd3
'define selD 4'd4
'define selE 4'd5
reg [3:0] select;
reg out;
reg a, b, c, d, f;
case (select)
    'selA: out <= a;
    'selB: out <= b;
    'selC: out <= c;
    'selD, 'selE: out <= d;
    default: out <= f;
endcase
```

// a barrel shifter
reg [2:0] s;
reg [7:0] in, out;
always @(in or s)
begin
    case (s)
        3'b000: out = in;
        3'b001: out = {in[6], in[5], in[4], in[3], in[2], in[1], in[0], in[7]};
        3'b010: out = {in[5], in[4], in[3], in[2], in[1], in[0], in[7], in[6]};
        3'b011: out = {in[4], in[3], in[2], in[1], in[0], in[7], in[6], in[5]};
        3'b100: out = {in[3], in[2], in[1], in[0], in[7], in[6], in[5], in[4]};
        3'b101: out = {in[2], in[1], in[0], in[7], in[6], in[5], in[4], in[3]};
        3'b110: out = {in[1], in[0], in[7], in[6], in[5], in[4], in[3], in[2]};
        3'b111: out = {in[0], in[7], in[6], in[5], in[4], in[3], in[2], in[1]};
    endcase
end
```
x or z in a case expression or case-item expression results in a warning and the comparison is not done for those bits.

The don't-care conditions are handled in the case statements, using casex and casez statements. The syntax of casex and casez statements is the same as the case statement, except for the keyword. In the case statement, both x and z are treated as don't-care bits. In casez, z is treated as don't-care bit.

Warp partially supports casex and casez statements. For the casex statement, ?, x, z are allowed in a case-item expression but not allowed in a case expression. Similarly, for casez statement, ?, z are allowed in a case-item expression but not allowed in a case expression.

Example:
/* instruction decoder of an ALU that performs 3 operations (arithmetic, logical and bit-wise). The most-significant 2 bits decodes the operator class, and the least significant 2 bits decodes the operator within the class */
reg [3:0] instr ;
casez (instr)
  4'b00??: arith_operator(..) ; /* call arithmetic operator function/task */
  4'b01??: logical_operator(..) ; // logical operator
  4'b10??: bitwise_operator(..) ; // bit-wise operator
endcase

In the above example, 'z' in instr[1] and instr[0] bits are treated as don't care.

To treat both 'z' and 'x' as don't care bits in instr[1] and instr[0] (above example), use casex as follows:
reg [3:0] instr ;
casex (instr)
  4'b00??: arith_operator(..) ; /* call arithmetic operator function/task */
  4'b01??: logical_operator(..) ; /* logical operator */
  4'b10??: bitwise_operator(..) ; /* bit-wise operator */
endcase

When Warp synthesizes any of the case statements, it synthesizes a memory element for each output assigned to it in the case statement (in order to maintain any outputs at their previous values) unless one of the following conditions occurs:

- All outputs within the body of the case statement are previously assigned a default value within the always block.
- The case statement completely specifies the design's behavior following any possible result of the conditional test. The best way to ensure complete specification of design behavior is to include a default clause within the case statement.

Therefore, to use the fewest possible resources during synthesis, either assign default values to outputs in the always block or make sure all case statements include a default clause.
2.9.5 Looping statements

Loop statements repeatedly execute a statement. The number of times a statement is executed is determined by the loop condition. *Warp* supports two kinds of loop statements: *for* loop and *while* loop.

```
for (init_assignment; condition; step_assignment) statement
while (expression) statement
```

In *for* loop statement, the *init_assignment* is a register assignment that initializes the loop variable. The *step_assignment* is a register assignment that assigns new value to the loop variable. The loop is executed until the condition evaluates to false. In *Warp*, the loop variable should be initialized to a constant value and the step assignment should be + or -. The loop condition should be a comparison (<, <=, >, >=) to a constant. The following are different for loop statements currently supported in *Warp*.

```
for( i = <number>; 
    i <comparison_operator> <number>; 
    i = i <increment/decrement> <number> ) 
....
```

where <comparison> is one of the comparison operators: <, <=, >, >= and <increment/decrement> is + or - operator. <number> is an integer number.

For loops can be nested.

Example:

```verilog
integer i;
reg [7:0] a, b;
for ( i = 0 ; i < 8 ; i = i + 1 )
    if( i > 3 ) b[i] = ~a[i] ;
    else b[i] = a[i] ;

// for loop not supported in Warp
integer i;
reg a;
for( i = a; i <= 8; i = i || a)
    ...
```

The *while* loop is supported only inside a function or task. The *while* loop condition should be a comparison to a constant. The following is the *while* loop template supported in *Warp*.

```
while ( i <comparison> <number>)
```

where <comparison> is one of the comparison operators: <, <=, >, >= and <number> is an integer number, and ‘i’ is a local variable that has been previously assigned a constant value.
Example:

```verilog
function [7:0] b;
    input [7:0] a;
    reg [2:0] i;
begin
    i = 0 ;
    while (i <= 7)
        begin
            if(i > 3) b[i] = ~a[i] ;
            else b[i] = a[i] ;
            i = i + 1 ;
        end
end
endfunction
```

2.9.6 Generate Statements

Warp supports the following:

2.9.6.1 `generate/endgenerate`

This is optional in Verilog 2005 and is required for Verilog 2001. Warp follows the more recent Verilog 2005 standard.

2.9.6.2 `if-generate`

```verilog
generate
    if (constant_expression)
        module_item_or_null
    [else module_item_or_null]
endgenerate
```

where constant expression is an expression involving constants, parameters and localparams.

`module_item_or_null` is either a begin/end block, any item that is valid in the module-body (like an always clause, a module instance, etc.)...or a simple ';' to say that you have nothing for the condition. The else part is optional.

Example:

```verilog
generate
    if (oper == "AND") // Assume Oper here is a parameter
        assign result = a & b ;
    else // Else it is assumed to be an OR
        assign result = a | b ;
endgenerate
```
2.9.6.3 case-generate:

Similar to if-generate, except using a case statement

```verilog
generate
  case (oper) /* Assume Oper here is a parameter */
  "OR": assign result = in_a | in_b;
  "XOR": assign result = in_a ^ in_b;
  "AND": assign result = in_a & in_b;
  default: assign result = 1'bX;
endcase
endgenerate
```

2.9.6.4 for-generate:

You can have a loop generate where you can create zero or more module items. The following is an adder created using the for-generate:

```verilog
generate
  genvar i;
  for(i=0; i<SIZE; i=i+1)
    begin: addbit
      wire n1,n2,n3; //internal nets
      xor g1 ( n1, a[i], b[i]);
      xor g2 (sum[i], n1, c[i]);
      and g3 ( n2, a[i], b[i]);
      and g4 ( n3, n1, c[i]);
      or g5 (c[i+1],n2, n3);
    end
endgenerate
```

2.10 Timing controls

In Verilog, timing controls provide a means to control the time at which a statement is executed or the time at which values are assigned to a net or register data type.

Warp ignores the intra-assignment timing controls, delay based timing controls and wait timing controls. Event timing controls are partially supported (onlyposedge and negedge event timing controls are supported when used with an always @).

Example:

```verilog
// timing control supported in Warp
always @(posedge clk or negedge reset ..)
...
2.11 Structured procedures

2.11.1 Initial

_Warp_ ignores the _initial_ construct.

2.11.2 Always

An _always_ statement contains a block of statements that are executed whenever that _always_ statement becomes active. An _always_ statement that is executing is said to be active; otherwise, it is said to be suspended. Every _always_ statement in the Verilog design may be active at any time. All active _always_ statements are executed concurrently with respect to simulation time.

An _always_ statement can be activated by means of a sensitivity list (a list of signals enclosed in parentheses appearing after the _always_ keyword). In _Warp_, an always statement must have a sensitivity list. The sensitivity list is specified using the following syntax:

```verbatim
@(event_expression [or event_expression {or event_expression}])
```

Timing controls other than _posedge_ and _negedge_ are not allowed in the _event_expression_. The syntax for the _event_expression_ is:

```verbatim
identifier | _posedge_ identifier | _negedge_ identifier
```

Bit-selects and part-selects are not allowed in the _event_expression_. The sensitivity list must contain an expression consisting of either plain identifiers or _posedge_/ _negedge_ tagged identifiers but never a combination of both.

Event expressions with plain identifiers result in combinational logic, unless a latch is inferenced using the latch inferencing mechanism (refer to latch synthesis section).

When _posedge_/ _negedge_ identifiers are used in the sensitivity list, sequential logic is synthesized.

Examples:

```verbatim
/* Always block for combinational logic: */ always @(x or y) begin ... end

/* Always block which realizes sequential logic using the rising edge of a clock: */ always @(posedge clock) begin ... end

/* Always block which realizes sequential logic using the falling edge of a clock and an asynchronous preload */ always @(negedge clock or posedge load) begin ... end
```

Refer to the Verilog Synthesis on page 33 for a list of synthesis templates supported by _Warp_. 
2.11.3 Task

Tasks are sequences of declarations and statements that can be invoked repeatedly from different parts of a Verilog description. They also provide the ability to break up a large behavioral description into smaller ones for easy readability and code maintenance.

A task can return zero or more values. A task declaration has the following syntax.

```
task <task_name> ;{ task_item_declaration}
statement_or_null endtask
```

task_item_declaration ::= parameter_declaration
| reg_declaration
| integer_declaration
| input_declaration
| output_declaration
| inout_declaration

Warp ignores any timing controls present inside a task. The order of variables in the task enable statement (calling a task) should be the same as the order in which the I/Os are declared inside a task definition.

Only reg variables can receive output values from a task. Wire variables cannot.

Note Datapath operator inferencing is not supported inside tasks. When datapath operators (+, -, *) are used inside tasks, atleast one of the operands should be a constant or an input.

Example:

```
module task_example(a,b,c,d,sum);
  output sum;
  input a,b,c,d;
  reg sum;
  always @(a or b or c or d)
  begin
    t_sum(a,b,c,d,sum);
  end

  task t_sum;
  input i1,i2,i3,i4;
  output sum ;
  begin
    sum = i1+i2+i3+i4;
  end
endtask
endmodule
```

Disabling of named blocks and tasks using the disable construct are not supported by Warp. All system tasks are ignored by Warp.
2.11.4 Function

Similar to tasks, functions are also sequences of declarations and statements that can be invoked repeatedly from different parts of a Verilog design. They also provide the ability to break up a large behavioral description into smaller ones for readability and maintenance.

```
function [range_or_type] <function_name>
  function_item_declaration {function_item_declaration}
  statement endfunction
function_item_declaration ::= parameter_declaration
  | reg_declaration
  | integer_declaration
  | input_declaration
```

Unlike a task, a function only returns one value. The function declaration will implicitly declare an internal register which has the same type as the type specified in the function declaration. The return value of the function is the value of this implicit register.

A function should have at least one input type argument. It can not have an output or inout type argument.

A function declaration can have the following declarations:

```
input declaration, reg declaration, integer declaration, parameter declaration.
```

The order in which the inputs are declared should match the order in which the arguments are used in the function call.

Timing controls and nonblocking assignment statements are not allowed inside a function definition.

Datapath operator inferencing is not supported inside functions. When datapath operators (+, -, *) are used inside functions, atleast one of the operands should be a constant or an input.

The function inputs can not be assigned to any value, inside the function.

All system task functions are ignored by Warp.

Example:

```
module func_example(a,b,c,d,sum);
  output [2:0] sum;
  input a,b,c,d;
  reg [2:0] sum;
  always @(a or b or c or d)
  begin
    sum = func_sum(a,b,c,d);
  end

  function[2:0] func_sum;
  input i1,i2,i3,i4;
  begin
    func_sum = i1+i2+i3+i4;
  end
endmodule
```
2.12 Compiler directives

2.12.1 `define

The directive `define creates a macro for text substitution. Once a text macro name is defined, it can be used anywhere in a source description. There are no scope restrictions. Redefinition of a text macro is allowed and the latest definition of a particular macro read by the compiler prevails when the macro name is encountered in the source text.

Examples:

```
`define selA 4'b1
`define selB 4'b2
`define selC `selA
```

2.12.2 `undef

The directive `undef is used to undefine a previously defined text macro. An attempt to undefine a text macro that was not previously defined using a `define compiler directive results in a warning.

Examples:

```
`define selA 4'b1
`undef selA
```

2.12.3 `include

The `include compiler directive allows one to include the contents of a source file in another file during compilation. The file name in the `include directive can be a full or relative path name. A file included in the source using the `include compiler directive may contain other `include compiler directives. The `include construct cannot be used recursively.

Example:

```
`include "lpm.v" // include lpm.v file
```

2.12.4 `ifdef, `ifndef, `else, `elseif, `endif

Example:

```
`ifdef WARP // warp specific code
....
`else // not warp specific code
....
`endif
```

When `ifdef WARP compiler directive is used, Warp compiles only the code within the `ifdef WARP block.

2.12.5 Unsupported compiler directives

All the other compiler directives are ignored and Warp issues a warning when it encounters any of the unsupported compiler directives.
3. Verilog Synthesis

3.1 Tri-state Synthesis

Warp does not synthesize tri-state logic. In order to include tri-state logic in a Verilog module the cy_bufoe must be instantiated. The tri-state output of this module, y, must then be connected to an inout port on the Verilog module. That port can then be connected directly to a bidirectional pin on the device. The feedback signal of the cy_bufoe, yfb, can be used to implement a fully bidirectional interface or can be left floating to implement just a tri-state output.

```verilog
module ex_tri_state (out1, en, in1);
    inout out1;
    input en;
    input in1;
    cy_bufoe buf bidi (
        .x(in1), // (input) Value to send out
        .oe(en), // (input) Output Enable
        .y(out1), // (inout) Connect to the bidirectional pin
        .yfb()); // (output) Value on the pin brought back in
endmodule
```

3.2 Latch Synthesis

Warp synthesizes a latch whenever a variable inside an always block with asynchronous trigger, has to hold its previous value. The following code fragment should synthesize a latch.

```verilog
// example: latch synthesis with if statement
always @ (signal1 or signal2)
begin
    if( signal1 )
        begin
            out_sig = signal2 ;
        end
end
```

3.3 Register Synthesis

3.3.1 Edge-Sensitive Flip-Flop Synthesis

Warp uses the following templates to synthesize synchronous flip-flops.

The template for the positive edge sensitive flip-flop is:

```verilog
always @ (posedge clock_signal)
    synchronous_signal_assignments
```
The template for the negative edge sensitive flip flop is:

```verilog
always @ (negedge clock_signal)
  synchronous_signal_assignments
```

### 3.3.2 Asynchronous Flip-Flop Synthesis

*Warp* uses the following format to synthesize asynchronous flip-flops with reset or preset.

```verilog
always @ (edge_of clock_signal or
  edge_of preset_signal or
  edge_of reset_signal)

  if (reset_signal)
  reset_signal_assignments
  else if (preset_signal)
  preset_signal_assignments
  else
  synchronous_signal_assignments
```

Use the `posedge` construct to specify active high condition and the `negedge` construct to specify active low condition.

The variables in the sensitivity list can appear in any order.

Subsequent reset or preset conditions can appear in the `else-if` statements. The last `else` block represents the synchronous logic.

The polarity of the reset/preset signal condition used in the sensitivity list and the polarity of the reset/preset condition in the `if/else-if` statements should be the same.

**Example:**

A `posedge` `reset_signal` condition in the sensitivity list is required when the reset condition is one of the following forms:

```verilog
if (reset_signal)
if (reset_signal == constant_one_expression)
```

A `negedge` `reset_signal` condition in the sensitivity list is required when the reset condition is one of the following forms:

```verilog
if (!reset_signal)
if (~reset_signal)
if (reset_signal == constant_zero_expression)
```

*Warp* generates an error if the polarity restriction mentioned above is violated.

*Warp* allows more than two asynchronous `if/else-if` statements before the synchronous `else` statement as shown in the following example.

```verilog
// An example of two different preset signals:

module asynch_rpp(in1, clk, reset, preset, preset2, out1);
input in1, clk, reset, preset, preset2;
output out1;
reg out1;
```
always @ (posedge clk or posedge reset or posedge preset or posedge preset2)
    if (reset)
        out1 = 1'b0;
    else if (preset)
        out1 = 1'b1;
    else if (preset2)
        out1 = 1'b1;
    else
        out1 = in1;
endmodule

\section{Case Statement Synthesis}

\textit{Warp} provides the user, a capability to specify a particular case block to be implemented like a multiplexer (parallel case) rather than a priority encoder (full case). A parallel case or a full case is specified by including the following directives before a case statement.

- warp parallel_case
- warp full_case

These directives can be specified within the Verilog comment section (line comment or block comment). The directive must follow the word "warp".

Examples:

\begin{verbatim}
case (expression)  // warp parallel_case
    ...
endcase

case (expression)  // warp full_case
    ...
endcase

case (expression)  /* warp parallel_case */
    ...
endcase

case (expression)  /* warp full_case */
    ...
endcase
\end{verbatim}
4. **Design Examples**

This chapter provides Verilog design examples. To use a Verilog file in PSoC Creator, it must be included as part of a component. For details about how to create a PSoC Creator component, refer to the *PSoC Creator Component Author Guide*.

4.1 **Counter**

This example describes the behavior of a counter that increments the count by 1 on the rising edge of a clock (trigger). It also contains an asynchronous reset signal that resets the counter to zero.

```verilog
module counter (trigger, reset, count);
    parameter counter_size = 4;
    input trigger;
    input reset;
    inout [counter_size:0] count;
    reg [counter_size:0] tmp_count;

    always @(posedge reset or posedge trigger)
    begin
        if (reset == 1'b 1)
            tmp_count <= {(counter_size + 1){1'b 0}};
        else
            tmp_count <= count + 1;
    end

    assign count = tmp_count;
endmodule
```

4.2 **Vending Machine**

This example describes a soft-drink dispensing machine. The machine has two bins to dispense regular cola and diet cola. Each bin holds three cans of soft drink. (This could be any value, but three is an easy number to simulate.)

The circuit dispenses a beverage if the user presses a button for that beverage and at least one can is available. A *refill* signal appears when both bins are empty. Pressing a *reset* signal tells the circuit that the machine has been replenished and the bins are full.

The circuit is implemented as a hierarchical design. The low level component of the design is a *binctr* which controls the operation of one bin. The *top level* is the description of the entire design. The *top level* circuit instantiates two *binctrs* and other logic as appropriate to describe the larger soda machine design. The low level design is named *binctr* and the top level design is named *refill*. 
Note PSOC Creator allows only one Verilog file per component. All modules for a component must be included in the same Verilog file. Also, all module names must be unique across the entire design.

4.2.1 Low-level Design

The following is example code for a bin controller of the soda machine:

```verilog
// Behavioral description of module binctr
module binctr (reset, get_drink, clk, give_drink, empty);
input reset;
input get_drink;
input clk;
inout give_drink;
inout empty;

parameter full = 2'b 11;
reg tmp_give_drink;
reg tmp_empty;
reg [1:0] remaining;

always @(posedge clk or posedge reset)
begin
    if (reset == 1'b 1)
    begin
        remaining <= full;
        tmp_empty <= 1'b 0;
        tmp_give_drink <= 1'b 0;
    end
    else
        begin
            if (remaining == 2'b 00)
                begin
                    tmp_empty <= 1'b 1;
                    tmp_give_drink <= 1'b 0;
                end
            else if (get_drink == 1'b 1)
                begin
                    remaining <= remaining - 1;
                    tmp_give_drink <= 1'b 1;
                end
            else if (get_drink == 1'b 0)
                begin
                    tmp_give_drink <= 1'b 0;
                end
            else
                begin
                    tmp_give_drink <= give_drink;
                    remaining <= remaining;
                    tmp_empty <= empty;
                end
        end
end
```

```
assign give_drink = tmp_give_drink;
assign empty = tmp_empty;
endmodule

A line-by-line explanation of the above design follows.

The module definition names the design and identifies the I/O ports used:
module binctr (reset, get_drink, clk, give_drink, empty);

The next 5 lines assign the direction of the ports identified by the module definition by defining them as inputs, outputs or inouts.
input reset;
input get_drink;
inout clk;
inout give_drink;
inout empty;

The parameter full is next defined as a constant of binary value 11:
parameter full = 2'b 11;

The remaining 3 lines of the definition create temporary variables as reg's for keeping track of signals used in the always procedural block:
reg tmp_give_drink;
reg tmp_empty;
reg [1:0] remaining;

The always procedural block describes the action of the design in response to the “clk” and “reset” signals. In this instance, it is triggered on the positive edge of either.

The body of the block describes the logic followed when either of these triggering signals is received. The temporary variables are used internal to the block to implement the logic needed.
always @(posedge clk or posedge reset)
begin
  if (reset == 1'b 1)
    begin
      remaining <= full;
      tmp_empty <= 1'b 0;
      tmp_give_drink <= 1'b 0;
    end
  else
    begin
      if (remaining == 2'b 00)
        begin
          tmp_empty <= 1'b 1;
          tmp_give_drink <= 1'b 0;
        end
      else if (get_drink == 1'b 1)
        begin
          remaining <= remaining - 1;
          tmp_give_drink <= 1'b 1;
        end
      else if (get_drink == 1'b 0)
begin
    tmp_give_drink <= 1'b 0;
else
    begin
        tmp_give_drink <= give_drink;
        remaining <= remaining;
        tmp_empty <= empty;
    end
end

To make the internal signals visible to the outside, the temporary variables are assigned to the ports of the module.

assign give_drink = tmp_give_drink;
assign empty = tmp_empty;

Finally, the definition of the module is terminated with:

endmodule

4.2.2 Top-Level Design

The following is example code for the refill module of the soda machine:

module refill (GIVE_cola,GIVE_diet,REFILL_BINS,RESET,CLK
    GET_diet,GET_cola);
inout   GIVE_cola;
inout   GIVE_diet;
output  REFILL_BINS;
input   RESET;
input   CLK;
input   GET_diet;
inout   GET_cola;

    binctr bin_1 (.reset(RESET),
                .get_drink(GET_cola),
                .clk(CLK),
                .give_drink(GIVE_cola),
                .empty(empty_1));

    binctr bin_2 (.reset(RESET),
                .get_drink(GET_diet),
                .clk(CLK),
                .give_drink(GIVE_diet),
                .empty(empty_2));

    assign REFILL_BINS = (empty_1 == 1'b 1 & empty_2 == 1'b 1)
        ? 1'b 1 : 1'b 0;
endmodule
The Verilog code used to implement the top level Verilog design is entirely structural in nature. The definition of the module with its ports is done with the first 9 lines of the file:

```verilog
module refill (GIVE_cola,GIVE_diet,REFILL_BINS,RESET,CLK
GET_diet,GET_cola);
    inout   GIVE_cola;
    inout   GIVE_diet;
    output  REFILL_BINS;
    input   RESET;
    input   CLK;
    input   GET_diet;
    input   GET_cola;
endmodule
```

Following those lines are instantiations of the `binctr` as `bin_1` and `bin_2`:

```verilog
binctr bin_1 (.reset(RESET),
    .get_drink(GET_cola),
    .clk(CLK),
    .give_drink(GIVE_cola),
    .empty(empty_1));

binctr bin_2 (.reset(RESET),
    .get_drink(GET_diet),
    .clk(CLK),
    .give_drink(GIVE_diet),
    .empty(empty_2));
```

Then, the constant assignment logic of the `REFILL_BINS` signal depending upon the value of `empty_1` and `empty_2` is done before the module definition is closed:

```verilog
assign REFILL_BINS = (empty_1 == 1'b 1 & empty_2 == 1'b 1)
    ? 1'b 1 : 1'b 0;
```

Finally, the definition of the module is terminated with:

```verilog
endmodule
```
### A. Verilog Reserved Words

The following is a list of reserved words that cannot be used as identifiers.

<table>
<thead>
<tr>
<th>always</th>
<th>and</th>
<th>assign</th>
<th>automatic</th>
<th>begin</th>
</tr>
</thead>
<tbody>
<tr>
<td>buf</td>
<td>buf0</td>
<td>buf1</td>
<td>case</td>
<td>casex</td>
</tr>
<tr>
<td>casez</td>
<td>cell</td>
<td>cmos</td>
<td>config</td>
<td>deassign</td>
</tr>
<tr>
<td>default</td>
<td>defparam</td>
<td>design</td>
<td>disable</td>
<td>edge</td>
</tr>
<tr>
<td>else</td>
<td>end</td>
<td>endcase</td>
<td>endconfig</td>
<td>endfunction</td>
</tr>
<tr>
<td>endgenerate</td>
<td>endmodule</td>
<td>endprimitive</td>
<td>endspecify</td>
<td>endtable</td>
</tr>
<tr>
<td>endtask</td>
<td>event</td>
<td>for</td>
<td>force</td>
<td>forever</td>
</tr>
<tr>
<td>fork</td>
<td>function</td>
<td>generate</td>
<td>genvar</td>
<td>highz0</td>
</tr>
<tr>
<td>highz1</td>
<td>if</td>
<td>ifnone</td>
<td>incdir</td>
<td>include</td>
</tr>
<tr>
<td>initial</td>
<td>inout</td>
<td>input</td>
<td>instance</td>
<td>integer</td>
</tr>
<tr>
<td>join</td>
<td>large</td>
<td>liblist</td>
<td>library</td>
<td>localparam</td>
</tr>
<tr>
<td>macromodule</td>
<td>medium</td>
<td>module</td>
<td>nand</td>
<td>negedge</td>
</tr>
<tr>
<td>mos</td>
<td>nor</td>
<td>noshown</td>
<td>cancelled</td>
<td>not</td>
</tr>
<tr>
<td>notif1</td>
<td>or</td>
<td>output</td>
<td>parameter</td>
<td>pmos</td>
</tr>
<tr>
<td>posedge</td>
<td>primitive</td>
<td>pull0</td>
<td>pull1</td>
<td>pulldown</td>
</tr>
<tr>
<td>pullup</td>
<td>pulsestyle_onevent</td>
<td>pulsestyle_ondetect</td>
<td>rcmos</td>
<td>real</td>
</tr>
<tr>
<td>realtime</td>
<td>reg</td>
<td>release</td>
<td>repeat</td>
<td>rmos</td>
</tr>
<tr>
<td>rmos</td>
<td>rtran</td>
<td>rtran0</td>
<td>rtran1</td>
<td>scalared</td>
</tr>
<tr>
<td>showcancelled</td>
<td>signed</td>
<td>small</td>
<td>specify</td>
<td>specparam</td>
</tr>
<tr>
<td>strong0</td>
<td>strong1</td>
<td>supply0</td>
<td>supply1</td>
<td>table</td>
</tr>
<tr>
<td>task</td>
<td>time</td>
<td>tran</td>
<td>trani0</td>
<td>trani1</td>
</tr>
<tr>
<td>tri</td>
<td>tri0</td>
<td>tri1</td>
<td>triand</td>
<td>trior</td>
</tr>
<tr>
<td>trireg</td>
<td>unsigned</td>
<td>use</td>
<td>vectored</td>
<td>wait</td>
</tr>
<tr>
<td>wand</td>
<td>weak0</td>
<td>weak1</td>
<td>while</td>
<td>wire</td>
</tr>
<tr>
<td>wor</td>
<td>xnor</td>
<td>xor</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>