Cycle Model Studio

Version 9.3

Cycle Model Compiler Verilog and SystemVerilog Language Support Guide

ARM

ARM 100972_0903_00_en

Cycle Model Studio

Cycle Model Compiler Verilog and SystemVerilog Language Support Guide

Release Information

Document History

Issue	Date	Confidentiality	Change
0902-00	25 May 2017	Non-Confidential	First release
0903-00	31 August 2017	Non-Confidential	Release with 9.3

Non-Confidential Proprietary Notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20349

Confidentiality Status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Unrestricted Access is an ARM internal classification.

Product Status

The information in this document is Final, that is for a developed product.

Web Address

http://www.arm.com

Contents

Cycle Model Studio Cycle Model Compiler Verilog and SystemVerilog Language Support Guide

Preface

This preface introduces the *Cycle Model Studio Cycle Model Compiler Verilog and SystemVerilog Language Support Guide*.

It contains the following:

About this book

This document describes the Cycle Model Compiler support for the Verilog and SystemVerilog languages.

Using this book

This book is organized into the following chapters:

Chapter 1 Introduction

This section provides basic information about using the Cycle Model Compiler.

Chapter 2 Verilog 95, Verilog 2001, and SystemVerilog Support

This section covers the supported subset of the language constructs provided by the Cycle Model Compiler software for Verilog 95, Verilog 2001, and SystemVerilog (2012) design files.

Glossary

The ARM® Glossary is a list of terms used in ARM documentation, together with definitions for those terms. The ARM Glossary does not contain terms that are industry standard unless the ARM meaning differs from the generally accepted meaning.

See the *ARM® Glossary* for more information.

Typographic conventions

italic
: Introduces special terminology, denotes cross-references, and citations.

bold
: Highlights interface elements, such as menu names. Denotes signal names. Also used for terms in descriptive lists, where appropriate.

`monospace`
: Denotes text that you can enter at the keyboard, such as commands, file and program names, and source code.

`mono`space
: Denotes a permitted abbreviation for a command or option. You can enter the underlined text instead of the full command or option name.

`monospace italic`
: Denotes arguments to monospace text where the argument is to be replaced by a specific value.

`monospace bold`
: Denotes language keywords when used outside example code.

`<and>`
: Encloses replaceable terms for assembler syntax where they appear in code or code fragments. For example:

```
MRC p15, 0, <Rd>, <CRn>, <CRm>, <Opcode_2>
```

SMALL CAPITALS
: Used in body text for a few terms that have specific technical meanings, that are defined in the *ARM® Glossary*. For example, IMPLEMENTATION DEFINED, IMPLEMENTATION SPECIFIC, UNKNOWN, and UNPREDICTABLE.

Feedback

Feedback on this product

If you have any comments or suggestions about this product, contact your supplier and give:

- The product name.
- The product revision or version.
- An explanation with as much information as you can provide. Include symptoms and diagnostic procedures if appropriate.

Feedback on content

If you have comments on content then send an e-mail to *errata@arm.com*. Give:

- The title *Cycle Model Studio Cycle Model Compiler Verilog and SystemVerilog Language Support Guide*.
- The number ARM 100972_0903_00_en.
- If applicable, the page number(s) to which your comments refer.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Note

ARM tests the PDF only in Adobe Acrobat and Acrobat Reader, and cannot guarantee the quality of the represented document when used with any other PDF reader.

Other information

- *ARM Developer*.
- *ARM Information Center*.
- *ARM Technical Support Knowledge Articles*.
- *Support and Maintenance*.
- *ARM Glossary*.

Chapter 1
Introduction

This section provides basic information about using the Cycle Model Compiler.

It contains the following sections:

1.1 Compilation Specifications

This section provides information about Cycle Model Compiler compilation modes.

Specifying compilation mode

By default, the Cycle Model Compiler processes design files using the Verilog 95 language definition (IEEE 1634-1995).

To enable Verilog-2001 compilation mode, use the `-2001` option. This includes partial support for Verilog-2005 (IEEE Std 1364-2005) language features. All files encountered during the compilation are treated as Verilog 2001. Note that you may also use `-2000` or `-v2k` to enable this compilation mode; these three options are equivalent.

To enable SystemVerilog compilation mode, use the `-sverilog` option. All Verilog files encountered during compilation are treated as SystemVerilog source files.

Full and partial compilation

The Cycle Model Compiler does not support partial compilation using compilation units as described in Section 3.12.1 of the Language Standard; full compilation is supported. This means that you must include a specification of all Verilog files when you issue the `cbuild` command.

1.2 Compiler response to unsupported constructs

This section describes Cycle Model Compiler behavior in response to unsupported or ignored constructs.

If the Cycle Model Compiler encounters a construct that is unsupported, it:

- issues a warning and continues, or
- issues an alert or error and exits.

In cases where errors are reported, the offending constructs must be removed through remodeling. In cases where an alert is reported, the construct must be fixed or the alert demoted. For information about demoting alert severity, see the compiler directives described in the *Cycle Model Compiler User Manual* (Arm DUI0957).

If the Cycle Model Compiler encounters a construct that is ignored, it may or may not issue a message and will continue compiling.

Chapter 2
Verilog 95, Verilog 2001, and SystemVerilog Support

This section covers the supported subset of the language constructs provided by the Cycle Model Compiler software for Verilog 95, Verilog 2001, and SystemVerilog (2012) design files.

It contains the following sections:

2.1 General Constructs

This section describes the Cycle Model Compiler Verilog construct support.

Supported Constructs

- modules (macromodule) and instances.
- `Endmodule : <modulename>` construct.
- ANSI and non-ANSI style port declarations.
- identifiers (including escaped identifiers).
- memories (2 or more dimensional reg arrays). Maximum bit size of the array is 2^{32} bits. See also memory index expressions in the Limited Support section.
- parameters, localparams, and parameterized instances.
- bit selects and variable indices.
- strings: called "`string`" in Verilog 2001, and called "`string literal`" in SystemVerilog.
- Verilog compiler directives: `` `define ``, `` `default_nettype ``, `` `ifdef ``, `` `ifndef ``, `` `else ``, `` `endif ``, `` `undef ``, `` `include ``, `` `resetall ``, `` `timescale ``.By default, `` `define ARM_CM `` is defined for all Verilog and SystemVerilog design files.

  For conditional code blocks: Conditional code blocks must open (`` `ifdef ``, `` `ifndef ``) and close (`` `endif ``) in the same file. For example, placing an `` `ifdef `` in one file and its corresponding `` `endif `` in an `` `included `` file is illegal. `` `else `` directives must also be placed in the same file as their associated `` `ifdef `` or `` `ifndef ``.

  Similarly, when used in a `` `protected `` section, conditional code blocks must open and close within that section. When used in a file with one or more `` `protected `` sections, paired `` `ifdef `` and `` `endif `` directives must be placed outside of `` `protected `` sections. For example, placing an `` `ifdef `` in a file and its corresponding `` `endif `` inside of a `` `protect/`endprotect `` is not supported.
- Unsized Constants. These constants are sized according to the rules in the Language Standard.
- `ifnone` Conditions. As mandated by the Language Standard, only simple module paths may be described with an `ifnone` condition.
- `cast` operator (`'`) (For example, `casting_type ' (expression)`).
- Integer declared in `begin`/`end` block.
- `Integer`/`genvar` declared as `localParam`.
- Use of `inc_or_dec` operator (`++` or `--`).
- Use of combined assignment operators such as `+=` , `|=`, `&=`.
- `Timeunit` and `timeprecision`.
- User-defined types defined with the `typedef` syntax.
- Use of packages to define `typedef`s, `enum`s, and functions.
- `Output` and `inout` ports for functions (independent of port data type).
- Ignoring the value returned by a function.
- Continuous assigns to reg, and blocking and non-blocking assigns to logic.
- `static` and `automatic` qualifiers to distinguish between variables within functions, tasks, or procedural blocks.

Limited Support

- Enumeration declaration with `typedef` syntax, and usage of variables and values declared with this type. The built-in functions `.first()`, `.last()`, and `.size()` are supported. The built-in functions `.next()`, `.prev()`, and `.name()` are not supported.
- Port specifications in module declarations are generally supported; however the following cases are not supported: Concatenation expressions in the module declaration port list are not supported:

```
module foo ( {a,b}, .d{e,f} );
```

  A bit or part select that is not for the full identifier is not supported:

```
module foo ( in1[3:1] ) ; // full width not selected
                input [3:0] in1;
```

Multiple occurrences of the same identifier in a module declaration is not supported, except when all bits are specified and listed in declaration order:

```
module foo (b[2], b[1], b[0]) // supported
                input [2:0] b;
module foo ( a, a); // not supported
                input a;
```

- Multiple driven nets. The Cycle Model Compiler selects a driver and does not perform conflict resolution; the exception is tristates, which are handled correctly.
- Specify blocks The Cycle Model Compiler does not ignore `specify` blocks, however it does ignore most of the contents of `specify` blocks. Only the following two optional and implicit connections are recognized: 1) between the net of the `reference_event` and the `delayed_reference` net, and 2) between the net of the `data_event` and the `delayed_data` net.

  If the `$setuphold` includes a specification for a `delayed_reference` net and it is the same width as the net of the `reference_event`, then a continuous assignment is created: `assign delayed_reference = reference_event_net;`.

  If the `$setuphold` includes a specification for a `delayed_data` net and it is the same width as the net of the `data_event`, then a continuous assignment is created: `assign delayed_data = data_event_net;`.

  The partial support provided for `$setuphold` does not include the timing check that is specified by the `$setuphold`.
- Memory index expressions. The Cycle Model Compiler does not support memory index expressions that are wider than 32 bits. If a memory index expression wider than 32 bits is found, the Cycle Model Compiler prints a warning and truncates the expression to the least significant 32 bits. The Cycle Model Compiler implements the equivalent of the following transformation: Original Verilog:

```
...
  reg [7:0] mem [1023:0];
  reg [63:0] index;
...
  always @(...) begin
  mem[index] = value;
```

  The Cycle Model Compiler transformation:

```
...
  reg [7:0] mem [1023:0];
  reg [63:0] index;
  reg [31:0] short_index;
...
  always @(...) begin
  short_index = index[31:0];
  mem[short_index] = value;
  end
...
```

  In addition, the Cycle Model Compiler prints an error if it finds that it must truncate an index expression and the memory has been declared with a range that includes negative values.
- Support for the exponent operator (`a ** b`) is limited as follows: At least one of the bases or exponents must be a constant.

  For non-constant bases the exponent must be a constant power of 2.

  For non-constant exponents the base must be a constant power of 2.
- Hierarchical references to variables.
- Variables in SystemVerilog functions default to `static` or `automatic` as defined in the Language Standard. Static variables can be initialized provided that initialization does not depend on an `automatic` or `port`.
- `Inout` ports with an associated memory type are not supported. `Inout` ports with structure or union type are supported, provided that they do not contain nested memories.

- The function `$clog2()` is supported in the case where the argument is a constant. The following alert is emitted if the argument is non-constant: `Alert 3271: Non-constant argument for $clog2 is unsupported.`
- The `priority`, `unique`, and `unique0` keywords are ignored, but do not cause errors. The related Violation Checks are not performed and Violation Reports are not created. A warning is emitted that states that these keywords are ignored. The associated case or if-then-else statements are executed as specified in the Language Standard.
- Wildcard equality binary operators (`==?` and `!=?`) are supported only when the right-hand operand is a constant. For example:

```
a ==? 3'b1x0; // supported
3'b00x ==? c; // not supported
```

- `case inside` is fully supported except when ‘x’, ‘z’ or ‘?’ values appear in the case select expression. If ‘x’, ‘z’ or ‘?’ values are specified in the case select expression, the following alert is printed:

```
Alert 3273: 'x','z','?' values are unsupported for case statement select expression.
```

The following table shows examples of supported and unsupported `case inside` expressions. In this table, `a`, `b`, and `c` are variables:

Table 2-1 Supported and unsupported case inside expressions

Supported	Unsupported

`Case (a) inside`
`4’b10x0:`
`4’b1xz1:`
`4’b??00:`
	`Case (4’b1x10) inside`
	`4’b10x0:`
	`4’b1xz1:`
	`4’b??00:`

`Case (4’b1010) inside`
`a:`
`b:`
`4’b1010:`
	`Case (4’b1?zx) inside`
	`a:`
	`b:`
	`4’b1010:`

- If you use the `always_comb`, `always_ff`, or `always_latch` construct, be aware of the following limitations: Section 9.2.2.2 of the Language Standard specifies that variables written on the left-hand side of assignments must not be written to by other processes. The Cycle Model Compiler does not perform this check or issue a warning if this language requirement is not met.

  The Cycle Model Compiler does not check or warn you if the logic within the `always_comb` does not represent combinational logic. Similarly, checks are not performed and warnings are not issued if the logic within `always_ff` does not represent flip-flop logic, or if the logic within `always_latch` does not represent latch logic.

  Auto-trigger of the body of the block may not be performed at time `0`.

  Implicit sensitivity list of `always_comb` blocks may not include inputs to functions called from within the `always_comb` construct.
- Using the conditional operator (`?:`) with aggregate expressions and integral types is supported. However, using the conditional operator with nonintegral types like `Real`is not supported.
- The following name spaces, defined in Section 3.13 of the Language Standard, are supported:
  `definitions`

  `package`

  `compilation-unit scope` (see *1.1 Compilation Specifications* on page 1-10 for information about supported compilation units.)

  `text macro`

  `module`, with the exception of the checkers because checkers are not supported.

`block`

`port`

The `attribute` name space is not supported.

- Using `disable` to disable blocks, tasks, loops, and `non_local` blocks is partially supported. See *2.4 Behavioral Constructs* on page 2-19 for more information.)
- Task argument passing and function argument passing by value, default argument values, binding by name, or optional argument list is supported. Task argument passing and function argument passing by reference is not supported. Default argument groups are not supported.

Unsupported

- `$cast dynamic` casting function.
- `realtime` data type.
- Libraries, library map files (Language Standard, 1364-2001 sec 13.1.1, or 1800-2012 section 33.3).
- Recursive tasks and functions.
- `Final` blocks.
- Streaming operators {<<{}} and {>>{}}.
- Binary operators with arguments of type `real` or `shortreal`.
- `string` type.
- Automatic conversion of `shortreal` type to `integer` type.
- Calling a nonvoid function that requires no arguments without parentheses `()` is not supported. For example: `i = foo() + 42` is supported

  `i = foo + 42` is not supported
- Conditional operators `&&&` and `matches`.
- Selection statements used in `if`, `case`, and pattern matching operators (`&&&` and `matches`).
- The keywords `unique`, `unique0` and `priority` are ignored. The violation checks and reports that they enable are not generated. The behavior of the case or if-then-else statements that include these keywords are handled as defined in the Language Standard.
- `Jump` statements.
- Port declarations to `ref` port types (also known as `ref` port direction); the Cycle Model Compiler handles these as if they are hierarchical references.
- Compilation by unit (by unit scope or using `$unit`).
- Top-level instances of `$root` and reference to objects using `$root.name`.
- Nested modules (modules declared within modules).
- The ability to select configuration libraries.
- Format specifications related to assignment patterns and net strength (`$display` specifications such as `%P`, `%0P`, and `%V`).
- Extensions for handling packed data (`$readmemb` and `$readmemh`, `$writememb` and `$writememh`), including file format considerations.
- Time literals are not supported (Language Standard 2012, sec 5.8).
- Bounded queues.
- The SystemVerilog attribute syntax (for example, `(*full_case*)` is ignored. See the IEEE Language Standard 2012, section 5.12, for the syntax.
- Event control using `@`.
- Task and function argument passing by reference `ref`.
- Associative Arrays (Language Standard 1800-2012, section 7.8).
- Queues (Language Standard 1800-2012, section 7.10).
- Array Querying Functions (Language Standard 1800-2012, section 7.11).
- Classes (Language Standard 1800-2012, section 8).
- Bit select or part select starting from 65536 of a vector wider than 64K might cause a simulation mismatch.

Ignored

- `delays`#. For example, in `a = #5 b;` the `#5` is ignored.

2.2 Net Types

This section describes the Verilog Net Types supported by the Cycle Model Compiler.

Supported

- tri
- trireg
- tri1, tri0
- wire

Limited support

- wor, wand, trior, triand. These are treated as wire; the Cycle Model Compiler issues an alert and selects only one driver

2.3 Synthesizable Subset

In general, the Cycle Model Compiler supports the Synthesizable Subset of the Verilog language. This section provides details about this support.

Supported

The following aspects of the Synthesizable Subset are supported:

- `always` constructs that can be mapped into flops with 1 clock and asynchronous sets and resets; limited to one edge per signal.
- `always` constructs that can be mapped into latches with 1 clock and asynchronous sets and resets.
- `always` constructs that can be mapped to purely combinational logic.
- blocks (begin-end and named).
- blocking and non-blocking assignments.
- conditional statements.
- `full_case` and `parallel_case` in comments.
- `translate_off`/`translate_on`.
- tasks and functions.
- `genvars`.
- `generate` blocks that contain any of the following: declarations of variables, UDPs, gate primitives, continuous assignments, `initial` blocks, `always` blocks, functions, and tasks.
- `generate` statements: `generate-loop` (`generate-for`), `generate-conditional` (`generate-if`), and `generate-case`. `Generate` blocks that contain module instantiations are also supported.

Limited Support

The following aspects of the Synthesizable Subset have limited support:

- `initial` blocks with statements that can be evaluated to constants, or expressions that evaluate to constants, are supported. `initial` blocks with statements that cannot be evaluated to a constant are not supported.

Unsupported

The following aspects of the Synthesizable Subset are unsupported:

- procedural continuous assignments.
- implicit state machines in `always` or `initial` blocks.
- UDFs.

2.4 Behavioral Constructs

This section describes the Cycle Model Compiler support for Verilog behavioral constructs.

Fully Supported Constructs

- `for` statements
- `repeat` statements
- sensitivity lists
- `while` statements

Supported with Limitations

- `disable`. The target of the `disable` statement must be within the execution scope of the `disable` statement and must not be a hierarchical reference.

Consider the following where only the first `disable` statement is supported because it is within the execution of the target block.

```
always @(posedge clock)
begin
begin : block_1
if (a == 0)
disable block_1; // supported
else
task1();
end
disable block_1;      // not supported
end

always @(posedge clock)
begin
begin : block_2
if (a == 0)
disable block_1; // not supported
end
disable block_1;      // not supported
end
```

In addition, `disable` statements are only supported when the target is not a hierarchical reference. For example:

```
always @(...)
begin
if (in1 | in2)
disable task1a.b1;  // not supported
end
```

Unsupported

- events
- `force` and `release`
- `fork-join` blocks

2.5 Gate-level Constructs

This section lists the Verilog gate-level constructs supported by the Cycle Model Compiler.

Supported

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

2.6 Hierarchical References

This section describes the Cycle Model Compiler Verilog support for hierarchical references.

Limited Support

The Cycle Model Compiler supports hierarchical references only to nets, tasks, and functions. Hierarchical references to anything other than nets, tasks, and functions are not currently supported.

Unsupported

A hierarchical reference to a net declared under a task or function is not supported.

2.7 Switch-level Constructs

This sections describes the Verilog support for switch-level constructs.

Supported

Supported switch-level constructs are:

- `cmos`
- `nmos`
- `pmos`

Limited support

The following switch-level constructs have limited support:

- pullup sources are supported with the restriction: If a pullup source is connected to one or more bits of a vector, then a pullup source must be connected to all other bits of that vector.
- pulldown sources are supported with the restriction: If a pulldown source is connected to one or more bits of a vector then a pulldown source must be connected to all bits of that vector.
- strength ordering is supported, but limited to strong and pull strengths; strength propagation is not supported
- `rcmos` (converted to `cmos`).
- `rnmos` (converted to `nmos`).
- `rpmos` (converted to `pmos`).

Unsupported

Unsupported switch-level constructs are:

- `tran` (alias), `rtran`
- `tranif1`, `tranif0`
- `rtranif1`, `rtranif0`

2.8 User-defined Primitives

The Cycle Model Compiler supports most commonly-modeled User-defined Primitives (UDPs), thereby decreasing the time required to compile a design and move it into a test environment.

Supported

Latch models such as the following are supported:

Note

UDP descriptions generally do not yield the best performance from generated objects. Arm encourages replacing UDPs with RTL models whenever possible.

```
table
//D       G        : Q :        Qnext
1       1        : ? :        1
0       1        : ? :        0
?       0        : ? :        -
endtable
```

Limited Support

The following have limited support:

- Notifiers - UDPs with notifiers are handled; the notifier itself is ignored.
- Special optimization of separate Q, Qbar - Often Q and Qbar of a single flop are modeled with separate UDPs. The Cycle Model Compiler optimizes the result to a single state element, but it may not always do so. In such cases, performance may be improved by remodeling the UDP pair, or adding UDP pair optimization to recognize this common situation.

Unsupported

The following are unsupported:

- Latch models such as the following:

```
table
// D    G        : Q :        Qnext
(01)    1        : ? :        1
(10)    1        : ? :        0
1        *        : ? :        1
0        *        : ? :        0
?        0        : ? :        -
endtable
```

- Level behavior or combinational logic modeled with edges.
- Look-up-table implementation of UDPs.

2.9 System Tasks

This section describes the Cycle Model Compiler support for Verilog system tasks.

Fully Supported Constructs

- `$dumpvar` variants
- `$fsdbDumpvar` variants
- `$stop`
- `$time`

Supported with Limitations

Note

The system tasks `$fclose`, `$fflush`, `$sformat`, `$display`, `$fdisplay`, `$fwrite`, and `$fopen`, must be enabled with the `-enableOutputSysTasks` command line option. Otherwise, the Cycle Model Compiler issues a warning and ignores them. See the information about `-enableOutputSysTasks` in the *Cycle Model Compiler Guide* (Arm DUI0957) for more information.

- `$fclose`
- `$fflush`
- `$sformat`
- `$display`. `$display` is supported. `$display{b,h,o}` is not supported.
- `$fdisplay`. `$fdisplay` is supported. `$fdisplay{b,h,o}` is not supported.
- `$fopen`. Filenames must be constants at design compile time. The following examples show uses of filenames with `$fopen`.

```
$fopen("file1.dat");  // supported; filename is a constant
reg [72:1] filename1;
...
initial
begin
filename1 = "file2.dau";
filename1[1] = 1'b0; // change file extension from
//    .dau to .dat
end
$fopen(filename1);  // supported; filename is a constant at
//        Cycle Model Compiler runtime
------------------------------------------------------------
reg [72:1] filename2;
...
initial
begin
filename2 = "file2.dau";
if (in1) filename2[1] = 1'b0; // conditionally change
//    extension from .dau to .dat
end
$fopen(filename2); // not supported; filename is not
// a constant at Cycle Model Compiler runtime
```

- `$readmemb` and `$readmemh`. Filenames specified as strings (such as `data.dat`) are supported. Filenames specified with variables are not supported.
- `$fwrite`. `$fwrite` is supported. `$fwrite{b,h,o}` is not supported.
- `$write`. `$write` is supported. `$write{b,h,o}` is not supported.

Unsupported

Use of the following is not supported:

- `$bitstoreal`
- `$exit`
- `$feof`
- `$fgetc`
- `$fgets`
- `$fread`
- `$fmonitor{b,h,o}`

- $fscanf
- $fseek
- $fstrobe{b,h,o}
- $finish
- $ftell
- $itor
- $monitor
- $monitor {b,h,o,on,off}
- $random
- $realtime
- $realtobits
- $recordon
- $rewind
- $rtoi
- $sformatf
- $stime
- $sscanf
- $strobe{b,h,o}
- $swrite{b,h,o}
- $timeformat
- $ungetc
- $writemem{b,h}

2.10 Format specifications

This section describes the Cycle Model Compiler support related to Verilog format specifications.

Supported

- The following format specifications for real numbers are supported: `%e`, `%f`, and `%g`.
- The following escape sequences used for format specifications are supported, as defined in the Verilog standard (IEEE Std 1364-2005): `%h`, `%d`, `%o`, `%b`, `%c`, `%m`, `%s`, `%t`, `%u`, and `%z`.

Note

The `%u` and `%z` format specifiers are supported only for the `$fwrite` system output function.

Note

The current implementation produces only zeros and ones, not x or z values, for `%h`, `%o`, `%b`, `%v`, and `%z`.

Unsupported

- `%l` and `%v` format specifiers. As described in *2.1 General Constructs* on page 2-13, libraries and library map files are not supported.

2.11 Z State Propagation

This section describes the Cycle Model Compiler support for Z state propagation.

Supported

The Cycle Model Compiler has limited support for Z state propagation. The Z propagation is supported in simple assignment statements only. For example, in the following sample the Z state is propagated to `dout`.

```
module top(clk, rst, dout, re, din);

    input        rst, re, clk;
    input [3:0]  din;
    output [3:0] dout;

    reg [3:0]    dtemp;

    always @(posedge clk)
        if (re)
        dtemp <= din;
        else
        dtemp <=  'bz;

    assign dout = dtemp;

endmodule
```

Z propagation is implemented using aliasing, therefore any pullup or pulldown on one of the nets is applied to both nets. This can cause a simulation mismatch between the Cycle Model and other event-driven simulators.

Unsupported

The following cases are not supported:

- Any directives applied to the nets used in the assignment stop the Z propagation from occurring because aliasing does not occur.
- The Cycle Model Compiler does not support cases in which both of the nets in the assign are formal module ports, as in the following example:

```
module top(b1, b2, en, d);
    output b1;
    output b2;
    input  en,d;
    assign b1 = b2;
    assign b2 = en ? d : 'bz;
endmodule
```

Usage notes

The following warnings can be reported when either the net is undriven (weakly driven) or one of the nets in the chain is undriven (weakly driven). An example for each type of warning is shown in the following examples:

- `Warning 4020: Net is undriven`
- `Warning 4063: Net is weakly driven.`

Undriven example:

```
module top(b1);
    output b1;
    wire w1, w2;
    assign b1 = w1;
    assign w1 = w2;
endmodule

d.v:2 top.b1: Warning 4020: Net is undriven.
```

This warning reports that `b1` is undriven because the chain of nets `w2->w1->b1` is undriven.

Weakly Driven example:

```
module foo(i1, o1, o2, o3);
    input i1;
    output o1;
    output o2;
    output o3;
    tri1 w2;
    tri0 w3;
    assign o2 = w2;
    assign o3 = w3;
    assign o1 = i1;
endmodule

tristate_30.v:4 foo.o2: Warning 4063: Net is weakly driven.
tristate_30.v:5 foo.o3: Warning 4063: Net is weakly driven.
```

These warnings report that `o2` and `o3` are weakly driven because the chain of nets `w2->o2` and `w3->o3` are weakly driven.

2.12 Arrays

This section describes the Cycle Model Compiler support for arrays.

Supported

- Assignment of multi-dimensional arrays in blocking/non-blocking assignments.
- Full and slices of multi-dimensional arrays in port connections.
- System task arguments for multi-dimensional unpacked arrays.
- Assignment patterns for arrays and structures, including the use of `default:`.
- Use of unpacked structure and array data objects and unpacked structure and array constructors as aggregate expressions (Language Standard Section 11.2.2).

Limited Support

See *2.13 Unions* on page 2-30 for additional information about using arrays with unions.

- Arrays of regs declared within a named block of a `generate_for` loop must have hierarchical names.
- Assignment of packed arrays to unpacked is supported, and assignment of unpacked arrays to packed is supported. Assigning unpacked arrays to packed arrays is supported with casting; however, assigning packed arrays to unpacked arrays with casting is not supported. For example:

```
                        // Assigning unpacked array to packed array is supported with
casting
                        PackedArray_t mem_packed_C;
                        UnpackedArray_t mem_unpacked_C;
                         always @(posedge clock)
                          begin
                          for (int i = 0; i < 16; i=i+1) begin
                            mem_unpacked_C[i] = in1 + i;
                         end

                        mem_packed_C = PackedArray_t'(mem_unpacked_C);
                        out3 = mem_unpacked_C[address];
                        end

                        // Assigning packed array to unpacked array with casting is not
supported
                        PackedArray_t mem_packed_D;
                        UnpackedArray_t mem_unpacked_D;
                         always @(posedge clock)
                          begin
                          for (int i = 0; i < 16; i=i+1) begin
                               mem_packed_D[i] = in1 + i;
                        end

                        mem_unpacked_D = UnpackedArray_t'(mem_packed_D);
                        out4 = mem_unpacked_B[address];
                        end
```

Unsupported

- Array querying functions
- Unpacked array concatenations, as described in Section 10.10 of the Language Standard (1800-2012).
- Left hand side assignment patterns are skipped when datatype is explicit.

2.13 Unions

Unions are partially supported; the following limitations apply:

Limited Support

- Unpacked unions are not supported in the port list of the top-level module. This is because the SystemVerilog standard does not specify how many bits are required to represent an unpacked union. Therefore, it is impossible to know how many bits to reserve for an unpacked union port. This construct is probably unsynthesizable.
- Declaration of tagged union elements is allowed, but the tag is ignored.
- Tagged union expressions and member access requires the addition of storage to keep track of which union type was stored and is being read.

See the *Cycle Model Compiler Guide* (Arm DUI0957) for additional information about using directives with unions.

2.14 Structures

Structures are partially supported; the following limitations apply:

Limited Support

- For arrays of structures, out of bounds references using a variable index does not return the value defined in the Language Reference Manual.
- Unpacked structures are supported with the following limitations: Assignments to objects defined as structures are supported, but any initial value assignments to structure members (values defined in the structure definition) are not supported (Language Standard 1800-2012 7.2.2).

  Module inputs declared using the ANSI style declaration, and using an unpacked structure type, and specifying a default value are only partially supported. The declaration is supported but the default value is not applied. (See Language Standard 1800-2012 23.2.2.4 for instantiation rules - 23.3.2.1-23.3.2.4).

2.15 Interfaces

Interfaces are partially supported. This section describes which are features supported and which are not.

Supported interface features

- Tasks and functions in interfaces (Language Standard 1800-2012, section 25.7).
- Interface declaration/instantiation (Language Standard 1800-2012, section 25.3).
- Interface ports (Language Standard 1800-2012, section 25.4).
- Interface as a module port.
- Generic interfaces and generic interfaces with modports (Language Standard 1800-2012, section 25.3.3, section 25.10).
- Parametric interfaces (Language Standard 1800-2012, section 25.8).
- Arrays of interfaces.
- ANSI and non-ANSI style interface/modport module port declarations.
- Command-line directives on interface module ports on interface instances and interface members.
- Embedded directives on interface module ports.
- Hierarchical references to interfaces and modports or to members of an interface or modport.

Limited support

`Modports` (Language Standard 1800-2012, section 25.5) are generally supported except in following cases:

- CMS directives (such as `observeSignal` and `forceSignal`) using a hierarchical path through modports are unsupported.
- Nested interface `modport` expressions are unsupported, because nested interfaces (interfaces declared instantiated or used as a port inside an interface) are also unsupported (see the Language Standard 1800-2012, section 25.5).
- Only `modport` (including modport expression) and `constant` (parameter or localparam) declarations under `generate` statements are supported; `net` and `variable` declarations are unsupported.

Unsupported interface features

- Multiple task exports (see the Language Standard 1800-2012, section 25.7.4).
- Nested `interface` declarations (`interface` declared inside an `interface`).
- Use of `interface` in the port list of another `interface` declaration.
- Clocking blocks under interfaces (Language Standard 1800-2012, section 25.5.5).
- Interfaces with `specify` blocks (Language Standard 1800-2012, section 25.6).
- Virtual interfaces (Language Standard 1800-2012, section 25.9).
- `Interface` or `modport` usage in the portlist of the top-level module.
- `Always` block/`initial` block or continuous `assign` statements specified inside interface declarations.
- Embedded directives on `interface` instance.
- Interface member initializations are unsupported. Runtime `const` declarations using the `const` keyword are also unsupported inside an interface, because they must always have initial values.
- Nested interface uses are unsupported (`interface` instantiation or `interface port` declaration inside another interface).
- Use of the `generate` statement within the declaration of an interface.

2.16 Data Types

This section describes the support for data types by the Cycle Model Compiler:

Supported

The following data types, found in Verilog and SystemVerilog, are supported:

- `integer`

The following supported data types are found only in SystemVerilog:

- `logic`
- `bit`
- `byte`
- `shortint`
- `int`
- `longint`

Unsupported

The following data types, found in Verilog and SystemVerilog, are not supported:

- `realtime`
- `Void` when used as a function return type or member of a tagged union (Language Standard 2012, sec 6.13).
- `String` data type and associated string functions such as `len()` and `putc()`.
- Enumerated types in numerical expressions; for example, in an `Array` declaration where range is defined by an `enum` value.