Cycle Model Studio

Version 9.2

RTL Style Guide

Non-Confidential

Copyright © 2017 ARM Limited. All rights reserved.
ARM DUI 1088B (ID042717)

Cycle Model Studio
RTL Style Guide

Copyright © 2017 ARM Limited. All rights reserved.

Release Information

The following changes have been made to this document.

Change History

Date	Issue	Confidentiality	Change
February 2017	A	Non-Confidential	Restamp update
May 2017	B	Non-Confidential	Release with v9.2

Non-Confidential Proprietary Notice

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM Limited ("ARM"). **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated**.

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information for the purposes of determining whether implementations infringe any patents.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version shall prevail.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to ARM's customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement specifically covering this document with ARM, then the signed written agreement prevails over and supersedes the conflicting provisions of these terms.

Words and logos marked with ® or ™ are registered trademarks or trademarks of ARM Limited or its affiliates in the EU and/or elsewhere. All rights reserved. Other brands and names mentioned in this document may be the trademarks of their respective owners. You must follow the ARM trademark usage guidelines http://www.arm.com/about/trademarks/guidelines/index.php.

Copyright © ARM Limited or its affiliates. All rights reserved.
ARM Limited. Company 02557590 registered in England.
110 Fulbourn Road, Cambridge, England CB1 9NJ.

In this document, where the term ARM is used to refer to the company it means "ARM or any of its subsidiaries as appropriate".

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.

Product Status

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

Web Address

http://www.arm.com

Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

Contents

RTL Style Guide

Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

RTL Style Guide

The RTL coding styles you use to create designs can directly impact the efficiency of the Cycle Model that is generated by the Cycle Model Studio tool compiler. This document describes the recommended coding styles that yield the best performance from a Cycle Model. Where applicable, this document also describes methods for finding and correcting RTL constructs that are not optimal.

For a complete list of Verilog and SystemVerilog constructs that are currently supported, see the *Cycle Model Compiler Verilog and SystemVerilog Language Support Guide* (ARM 100972).

In some cases, using Cycle Model compiler options and directives can also improve the efficiency of a Cycle Model. Some of these options are also discussed in this document.

1.1 Cycles

Although Cycle Model Studio handles asynchronous cycles in a design, they execute much more slowly than non-cyclic logic. This is because the logic in a cycle must be executed multiple times to resolve the final values for the nodes in the cycle. Cycle Model Studio identifies each cycle and writes them to a file called `libdesign.cycles`. You can use this file to trace the nodes in the cycle and determine if there is a good way to break the cyclic path. This can be done by adding a flop or by fixing the logic so that it does not cycle. To make debugging easier, use the `-g` option with Cycle Model Studio; this exposes more intermediate nodes of the cycle.

Note that sometimes the Cycle Model Studio tool detects false cycles through bits in a vector where no real cycle exists. The Cycle Model Studio tool does this conservatively to guarantee correctness. For example, the following code is a false cycle:

```
input [3:0] in;
reg [3:0] veca, vecb;

always @(veca)
   vecb[3:0] = {veca[2:0], in[3]};
```


Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

```
always @(vecb)
   veca[3:0] = {in[2:0], vecb[0]};
```

There is no real cycle here because if the vectors are broken into bits there would be no cycle. The Cycle Model Studio tool pessimistically detects a loop only because each vector is dependent on the other in a combinational loop (no posedge or negedge clock blocks between them). After determining that there is no real cycle, it is very easy to remove it. To break the loop, the following code could be used:

```
input [3:0] in;
reg vecb_0;
reg [3:1] vecb;
reg [3:0] veca;

always @(veca or in)
   begin
      vecb[3:1] = veca[2:0];
      vecb_0 = in[3];
end

always @(in or vecb_0)
   veca[3:0] = {in[2:0], vecb_0};
```

There are also cases were the Cycle Model Studio tool detects false cycles through bits of a vector when the vector crosses a module boundary. Often, these cycles are eliminated with the proper use of Cycle Model compiler flattening options.

1.2 Latches

Although the Cycle Model Studio tool handles latches, they do not execute as fast as edge-based logic. There are two reasons for this: 1. the flow-through nature of latches reduces the optimizations that the Cycle Model Studio tool can perform on the surrounding logic, and 2. latches are much more likely to cause asynchronous cycles in the logic. Refer to "Cycles" on page 1-1, which describes why asynchronous cycles cause non-optimal behavior.

The latch count of a design is listed in the `libdesign.costs` file that the Cycle Model Studio tool generates. The latches are also listed in the Cycle Model compiler output if the `-verboseLatches` option is used.

If latches can be removed or remodeled as edge-based components, the Cycle Model Studio tool generates a more optimal Cycle Model. For example, take the following master-slave latch logic:

```
always @(in or clka)
   if (clka)
      tmp <= in;
always @(tmp or clkb)
   if (clkb)
      out <= tmp;
```

If `clka` and `clkb` are complements of each other, then the above logic could easily be remodeled as a flip-flop.

Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

```
always @(posedge clkb)
   out <= in;
```

1.2.1 Latch-Based Designs (LSSD)

Many latch-based designs model all of the storage elements in the design with two latches arranged in master/slave fashion. These designs can be converted to more efficient flop-based designs under certain situations. A level sensitive scan design (LSSD) example is provided here for illustrative purposes, but the techniques described can be used with other latch-based methodologies.

In the LSSD case, data is clocked into the storage device using two phase-inverted non-overlapping clocks typically called `L1CLK` and `L2CLK`. There are two additional clocks (typically called `ACLK` and `BCLK`) that enter into any storage element and are used to scan data in and out of the devices. LSSD storage elements are normally instantiated directly into the design as either single bits or ranks of bits. LSSD latch pairs can be remodeled more efficiently for use with Cycle Models under the appropriate conditions.

1. Determine how the LSSD devices are initialized. LSSD devices are typically initialized by either scanning data into the elements, or by holding `L1CLK` and `L2CLK` high for several clock cycles while holding the data inputs low. On many devices the storage elements are simply used for datapath and are not even initialized. A good understanding of initialization makes remodeling much easier.
2. Examine the design to make sure that the output of the `L1` latch is used only as the input of the `L2` latch. If it is used any other way, you are not able to optimize away the `L1` latch. Proving how `L1OUT` is used is typically simple because most LSSD storage elements are wrapped in a false hierarchy level, which hides it from the instantiating module, and simply passed to `L2IN`.
3. If `L1OUT` is used only as `L2IN`, remove the contents of the LSSD device and replace it with a simple flop (or rank of flops). Leave `L1CLK` open and use `L2CLK` to clock the flop. Refer to scan in step 4 below.

   If `L1OUT` is used outside of the storage element, remove the contents of the LSSD device and replace it with two flops–one for `L1` and the other for `L2`.
4. If scan is not used as part of initialization, make sure to disconnect the scan pins. If scan is used, mux the scan clock with the `L2CLK` based on the value of the scan enable pin. Note that scan initialization is rare.

Following is an example of LSSD remodeling. The original (`L1` latch goes only to `L2` latch):

```
module msff(l1_in,l1_si,l2_si,c1,c2,aclk,bclk,l2_out);
input  l1_in,l1_si,l2_si,c1,c2,aclk,bclk;
output l2_out;
reg    L1; // master_latch
reg    L2; // slave_latch
event  trig;

   always @(c1 or aclk or l1_in or l1_si or trig)
      if (c1)        L1 <= l1_in;
      else if (aclk) L1 <= l1_si;
         else        L1 <= L1;
   always @(c2 or bclk or L1 or l2_si or trig)
      if (c2)        L2 <= L1;
```


Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

```
        else if (bclk) L2 <= l2_si;
            else       L2 <= L2;
    assign l2_out = L2;
endmodule
```

Remodeled:

```
module msff(l1_in,l1_si,l2_si,c1,c2,aclk,bclk,l2_out);
input  l1_in,l1_si,l2_si,c1,c2,aclk,bclk;
output l2_out;
reg    L2; // slave_latch

    always @(posedge c2)
        L2 <= l1_in;
    assign l2_out = L2;
endmodule
```

1.3 Scalar Logic

Although the Cycle Model Studio tool takes every opportunity to vectorize logic, sometimes it cannot recombine vectors due to the complexities of the logic or hierarchy. For instance, a chip design may have vectored busses as ports to the top-level module, but breaks them into individual bits in the pad cells and then recombines them back into vectors at the core level of the chip. In such a case, the generated Cycle Model is inefficient because a larger number of operations are required to transfer data. Remodeling the pads so that they do not scalarize the vectors results in an optimal Cycle Model. For example, if the pad logic is as follows:

```
inpad a0 (.out(a[0]), .in(apad[0]));
inpad a1 (.out(a[1]), .in(apad[1]));
inpad a2 (.out(a[2]), .in(apad[2]));
```

a more efficient remodeling is:

```
assign a[2:0] = apad[2:0];
```

Note that memories often break up vectors into scalars as well. It is common for the top-level module of a memory to have vectored ports that are broken into scalar bits at lower levels, and then recombined into vectors again at even lower levels. Removing the levels that break apart the vectors, or remodeling those levels to keep the vectors intact results in a more efficient Cycle Model. Remodeling would look very similar to the pad example above.

1.4 Memories

The Cycle Model Studio tool supports most kinds of memory models, but some memories are modelled in a more complex manner than is required for pure 2-state functionality. Timing checks are sometimes added, as well as X checking. This extra modelling adds very little to the functional value of the model, and if removed/remodelled can have a positive impact on the resulting Cycle Model's performance. Often a very complex timing-accurate memory with X checking can be replaced with a much simpler 2-state functional model.

Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

1.5 Gate-Level Constructs

The Cycle Model Studio tool is most effective at optimizing high-level RTL constructs. Although the Cycle Model Studio tool supports most gate-level constructs, these constructs do not execute as fast as their RTL counterparts and removing them improves the performance of the Cycle Model.

Frequently, test logic is implemented at the gate-level and because many environments do not care about the test logic, it can be safely removed. One easy method of doing this is by using the `tieNet` directive on input test clocks and control pins. The `tieNet` directive sets the test pins to a constant inactive state. The Cycle Model Studio tool then has a better opportunity to optimize the associated test logic away.

Remodelling gate-level constructs into higher-level constructs also improves performance. For example, this pad cell model:

```
buf (inbuf, in);
buf (enn, en);
not (enp, en);
pmos (pad, inbuf, enp);
nmos (pad, inbuf, enn);
not (outbuf, pad);
not (out, outbuf);
```

could easily be remodelled more efficiently as follows:

```
assign pad = en ? in : 1'bz;
assign out = pad;
```

DesignWare components are a good candidate for this type of substitution. A DesignWare component of an adder may be implemented with very low-level or gate-level logic. Replacing the original DesignWare adder module with a higher-level RTL module (`out <= a + b`) improves performance of the Cycle Model.

1.6 Low-level Modeling

As with the gate-level constructs, modeling RTL at a very low level can have a negative impact on the Cycle Model's performance. Examples of this type of modeling could contain many continuous assign statements to implement higher-level functions. For instance:

```
assign sel0 = (sel == 2'b00);
assign sel1 = (sel == 2'b01);
assign sel2 = (sel == 2'b10);
assign sel3 = (sel == 2'b11);
assign out = sel3 ? in3 : (sel2 ? in2 : (sel1 ? in1 : (sel0 ? in0 : 8'b0)));
```

This could easily be remodeled as:

```
always @(sel or in0 or in1 or in2 or in3)
   case (sel)
      2'b00 : out = in0;
      2'b01 : out = in1;
      2'b10 : out = in2;
```


Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

```
        2'b11 : out = in3;
    endcase
```

Other examples include bit-slicing or vector slicing logic instead of operating at the full vector widths. See "Gate-Level Constructs" on page 1-5 for a discussion about DesignWare components.

1.7 Hierarchical References

Using hierarchical references inside Verilog code affects the performance of a Cycle Model. The same is true if using the Cycle Model API to reference signals in the design (*i.e.*, making signals depositable or observable). This is because optimizations have to be turned off at the signals that are hierarchically referenced. Reducing the number of hierarchical references improves Cycle Model performance. If these references are required, try to specify only state points (flop outputs) as being depositable, observable, or hierarchically referenced from Verilog. The advantage here is that state points are much less sensitive to performance degradation due to hierarchical referencing or outside referencing via the API.

Most of the time, hierarchical references are used by the testbench logic to examine signals inside the design. When using Cycle Model tools, it is better to obtain the values of signals using the Cycle Model API, which provides full access to any signal in the design. All that may be required is an `observeSignal` compiler directive on the accessed signals.

1.8 System Tasks

The inclusion of Verilog system tasks in a design can negatively affect Cycle Model performance.

Output system tasks that send data to the screen or to a file, such as $display, $write, $fdisplay, $fwrite, *etc.*, slow down performance because they create observable points in the RTL that cannot be optimized by the Cycle Model Studio tool. They can also generate a significant amount of file I/O. Reducing or eliminating the output system tasks in a design enables the Cycle Model Studio tool to create a more efficient Cycle Model.

The inclusion of the simulation control system tasks, $stop and $finish, can also negatively affect Cycle Model performance. Because these system tasks imply a strict execution order, relative to the updating of variables, no optimizations of RTL variables near these tasks can be performed. If the exact execution order between $stop and $finish and the surrounding assignments is not required, Cycle Model performance can be improved by isolating the control system tasks from the data assignments in separate always blocks. For example:

```
always @(sel or in1 or in2)
    case (sel)
        1'b0: begin
            a = in1;
            b = a;
        end
        1'b1: begin
            a = in2;
            $stop;
            b = a;
```


Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

```
        end
    endcase
```

This could be remodeled more efficiently as (losing the ordering between $stop and the a and b assignments):

```
always @(sel or in1 or in2)
    case (sel)
        1’b0: begin
            a = in1;
            b = a;
        end
        1’b1: begin
            a = in2;
            // $stop;
            b = a;
        end
    endcase

always @(sel)
    if (sel == 1’b1)
        $stop;
```

1.9 Clock Simplification

Cycle Model Studio handles any type of clocking (gated, asynchronous domains, *etc.*) efficiently. Even so, you can improve performance by simplifying the logic required to generate clocks.

An easy improvement is to remove test signals and test clocks from the clock tree if the test functionality is not required. (Usually the test logic is not exercised in the applications for Cycle Models, making this simplification a good candidate for performance improvement). The `tieNet` directive can be used to tie test signals and clocks to constants. The test logic can also be removed from the clock tree. For example:

```
assign clk = test_en ? test_clk : clk_in;
```

can be remodeled as:

```
assign clk = clk_in;
```

Another option is to use the Cycle Model compiler `tieNet` directive to set the test signals to constants. The tied nets are propagated to all associated parts of the design. For example:

```
tieNet 1'b0 top.test_en
tieNet 1'b0 top.test_clk
```

Another easy method to simplify the clocking is to use the API to drive the clocks as far downstream in the clock tree as possible. This removes much of the gating logic, and allows the API to efficiently drive the clocks. For example:

```
assign int_clk = test_en ? test_clk : clk_in;
always @(posedge int_clk)
    half_clk = ~half_clk;
```


Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential

In this example, if the API was used to drive `half_clk` directly (and remove the `assign` statement and `always` block), it would be more efficient than having the API drive `clk_in` (and keeping the `assign` statement and `always` block).

1.10 Test Logic

In general, removing test logic makes Cycle Models run faster. Most of the time, the test logic is not used in the same applications that Cycle Models are. This includes jtag functionality, BIST, and scan.

An easy way to remove test logic is to use the tieNet directive to tie test pins (clocks and control) to their de-asserted state. Another way is to actually remove this logic from the library cells that use them. Library cells such as pad, latches, flops, and memories often contain test logic. Remodeling these libraries to remove the test logic improves Cycle Model performance.

1.11 Remodeling Example

The following example shows the benefits of using vectors instead of scalars, and using a `casex` statement instead of a case. The `casex` is more efficient because the one-hot code requires fewer bits to compare.

Original:

```
reg [0:N] x, a, b;
x_0 = (a [0] ^ b[0]);
x_1 = ~x_0 & (a[1] ^ b[1]);
x_2 = ~x_0 & ~x_1 & (a[2] ^ b[2]);
...
x_N = ~x_1 ... x_N-1 & (a[N] ^ b[N]);
x = {x_0,x_1,x_2,x_3,...x_N}
case (x)
   100..:
   010..:
   001..:
   ...
   00..1:
endcase
```

Remodeled:

```
reg [0:N] x, a, b;
x = a ^ b;
casex (x)
   1??..:
   01??.:
   ...
   00..1:
endcase
```

Note that the value `'x'` takes is not changed between the original and revised model, however the output from the case statement is consistent.

Copyright © 2017 ARM Limited. All rights reserved.
Non-Confidential