The Ten Commandments of RTL Coding

Eric Ryherd

Cypress Semiconductor
15 Trafalgar Square
Nashua, NH USA

er@cypress.com

ABSTRACT

Most RTL is destined to be permanently cast into costly and unforgiving Silicon. Coding RTL is therefore very different than writing software which can be updated in the field on the next release. This paper presents the real world, high-level concepts on how to code RTL that is reusable, maintainable and relatively bug-free so that silicon is functional on the first pass. The following "Commandments" are general rules to live by when coding RTL. Inexperienced logic designers should strictly adhere to the rules presented in this paper. Experienced RTL coders should review these rules and contact the author with updates.

Table of Contents

Table of Figures

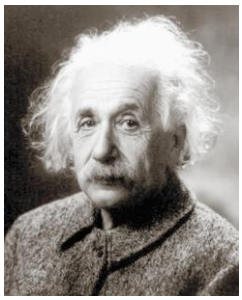
1.0 Introduction

Writing Verilog or VHDL (RTL) code seems very similar to writing C software. In fact, Verilog was modeled closely after C to make it more familiar. However, RTL code has one big difference compared to C, RTL code is usually destined to be cast into permanent and unforgiving silicon[1]. Silicon cannot be "upgraded" in the field like Software can. The RTL code must be perfect the first time around or delays in the launch of a new product will result.

Moore's law is continuing without pause and transistor count continues to double every 2 years. In order to fill these acres of silicon with meaningful and bug-free transistors, RTL code must be written to be highly reusable, easily maintainable, well documented and easy to implement. The Commandments listed in this paper provide ten high-level rules to live by that will increase the odds of high quality RTL code.

2.0 The Commandments

The "Commandments" below can always be bent depending on the situation. However, each time a commandment is not followed increases the risk that the design will have a bug or is so difficult to reuse that it is actually easier to recode it than try to fix it. A great deal of effort will be expended in coding, debugging, synthesizing, creating constraints and documenting RTL code. Ideally your hard work will be rewarded with years of reuse of the same RTL. Only a little additional effort early in the design phase is required to insure the commandments are followed.



2.1 Keep it Simple (KISS)

"Everything should be made as simple as possible, but not simpler" - Albert Einstein

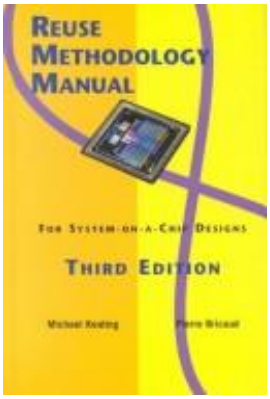
The simplest solution is almost always the best. Simple designs are easier to understand, test and support over time. They are also more likely to be reused. Never add a "feature" because it's "easy". Every feature has to be tested, verified, simulated, scan tested, fault graded, tested on a tester when silicon is built, supported in future revisions, made backwards compatible in future revisions and so on and on - forever. Every feature must be PROVEN to have value to the ***customer***, not just that it is easy to implement.

[1] FPGAs are "soft" silicon which allow hardware to be updated in the field. However, FPGAs are cost and/or power prohibitive for many applications. While the commandments can be loosened for RTL destined for FPGAs, the commandments should still be followed. The RTL code may be cast into silicon when it is reused on the next generation product, but only if it has followed these commandments.

Spend a few minutes, or even a few days **thinking** about how you are going to implement the RTL to match the requirements. Don't jump in and immediately start coding in a way that exactly matches the specification. Is there a way to perform the same function in a simpler way? Can functions be combined to reuse the same hardware multiple times? Can error conditions be simplified into a broad category so they can all be processed in the same way? If a slight simplification of the specification will significantly simplify the design, push back on the spec writers to see if they will accept the simplification and be sure to explain the ramifications of the more complex design. Don't hesitate to recode your RTL from scratch if you find that the solution is getting too complex. It'll only get worse over time and you'll never have time to recode it later.

The case of the unwanted UART status bits:
One of my best engineers was assigned the task of designing a 16-bit UART. The UART had a simple spec with a FIFO and few registers including a 16-bit status register. The status register only had 5 or 6 bits defined in it, the other bits were reserved. The bright engineer decided to put all sorts of "cool" status information in all of the reserved bits. Sounds like a great idea, but wait, the verification team didn't have any tests for any of these bits, the documentation didn't match the implementation and customer service was getting a lot of calls on what all these other bits do and why they weren't all zero. Since all of the bits were now full, there was no possibility of adding a bit when a customer *asked* for a feature to be added. While the reserved bits were "cool", no one wanted them. They wasted gates, made it hard for customer service, and made work for everyone and left no room for future expansion. He should have suggested one or 2 of his ideas when he first read the spec. Every one on the team could have weighed in on the value of each bit and we probably would have kept a couple. Instead, his hard work was removed.

2.2 Follow the Reuse and Methodology Manual Guidelines



The Reuse Methodology Manual (RMM) (Michael Keating & Pierre Bricaud ISBN:1-4020-7141-8) is **THE** guide on how to write RTL. The RMM is a "Best Practices" that has been put together by dozens of RTL veterans. All RTL engineers should at least read Chapter 5 - RTL Coding Guidelines. The RMM is the basis for most of the rules in Verilog Lint checkers.

A few key points from the RMM:

- Document a naming convention for modules and signals and STICK to it
- Indent code with spaces (not tabs)
- Port lists should be alphabetical
- Maintain signal names across hierarchy
- Signal names should be meaningful
- Always use active high logic (active low logic just confuses everyone - the one exception is reset_n)
- Busses should always be N:0 (never 0:N)

- Setup a timing budget early in the design process
- Register outputs
- Design fully synchronous circuits - they are easier to synthesize, analyze timing, place & route and test
- Use a language sensitive editor that understands Verilog
  - vim (or gvim) knows verilog syntax based on the .v extension and will highlight the syntax automatically.
  - emacs has many Verilog modes and can even automatically thread the hierarchy, fill in sensitivity lists, build generic syntax structures (like SWITCH/CASE or even State Machines) and much more.

All of these guidelines will make your job easier in the long run. With a consistent naming convention across every RTL file in a project (ideally across the entire company), anyone can reuse and maintain the code that you have worked so hard on. If your code does not follow the conventions, then later on you may even decide it's easier to rewrite the code from scratch than trying to figure out what the code is trying to do. Rewriting code wastes every ones time. It doesn't take any extra effort to follow these guidelines.

2.3 Comment the Intent of the Code

Anyone reading your RTL can be assumed to know the syntax of the language so simply stating WHAT operation is taking place isn't doing anyone any favors. Commenting WHY the operation is being performed and what the assumptions, inputs, outputs and side effects is they key. These comments will help you understand your own code in six months when you have to debug a problem. It will also help anyone trying to reuse or maintain your code in the future.

Bad Comments:

```
mulin <= left + right; // Add Left and Right

if (datrdy & ~st32ubw) begin // if datrdy and not st32ubw
```

Good Comments:

```
mulin <= left + right; // add the sources of the alu in preparation for
multiplication

if (datrdy & ~st32ubw) begin // if the ALU has data ready and not in a
stall condition
```

Always include a short description at the top of the file of what this code does at a high level. Typically this only needs to be two or three sentences that in very general terms describe what is being accomplished by the RTL in this file. Don't get too detailed or else the comments may no longer match the actual function of the RTL as things change. A few general sentences will always remain up-to-date and the details are in the code itself. Be sure to include your name in

the comment header- hey you worked hard on this RTL - take credit for that hard work and sign your name to it!

All RTL files should contain your company's copyright notice. This is for legal reasons - removal of the copyright notice by someone who has stolen IP signifies that the person knew the code was copyrighted and deliberately stole it. If the notice isn't there in the first place, someone might copy it and claim that they didn't know it was proprietary. This can be the difference between a multi-million dollar legal settlement versus a slap on the wrist.

2.4 Code with Hardware in Mind

There is a very big difference between writing software and coding RTL. Coding RTL might look and feel like you are writing software, but remember that all of this code will have to eventually be synthesized into silicon. The closer your code is to the actual silicon implementation, the more predictable your results (timing, area, power) will be. There are plenty of fancy features in Verilog and VHDL that are perfectly fine to use in a testbench or behavioral code, but should NEVER be used in synthesizable logic. The basic concept here is to code the RTL to be similar to what you are expecting in hardware. If you need a mux - code a MUX, if you need an adder - code an adder, don't make it into a complex state machine with many inputs/outputs that does everything (KISS helps the synthesizer too!).

Synthesizable RTL should always be synchronous. Think about how much logic a signal will have to travel thru before it reaches a flip-flop. If you're adding or multiplying vectors together, realize that the carry paths could take a long time to propagate. Don't let these long carry paths be an input to a complex state machine without registering the signal first. Wherever possible, register the outputs of a module. This is the most basic form of timing budgeting. Try not to pass an input directly to an output without a register stage somewhere. This tends to make for very long paths when modules are strung together. Silicon is relatively cheap - add DFFs where you can to make the timing easier. Often the design will actually be smaller because the timing is easier to achieve.

Never assume that the synthesizer will optimize your design. Synthesizers follow the simple rule of Garbage-In generates Garbage-Out. The closer RTL is to the actual silicon version, the more time the synthesizer will spend working on meeting timing and not have to try to figure out what you intended. I'm not saying you need to lay down each and every gate yourself in the RTL - that would be a complete waste of time. Use the power of the synthesizer to take well structured code and optimize the area and timing. Giving the synthesizer RTL code that it has to make assumptions about the intent of the code can result in dead logic, redundant logic, poor timing, poor area and worst of all, the gate level simulations may not match the RTL ones. Logic Equivalency may also fail if the code is poor.

Realize that each ELSE in an if-then-else is actually the AND of the NOT of all of the previous ELSE statements. The gate level result can become a very deep priority encoder that is in the

critical timing path. Use a SWITCH/CASE statement if there are more than 2 or 3 levels of ELSE statements.

State machines should be kept to 32 states or less. There are always exceptions to this rule but generally state machines with more than 32 states are so complex it may be better to break the state machine up into several smaller state machines. Testing every possible branch path thru a complex state machine will take a great deal of time. Try to simplify branch paths wherever possible. For example, try to have all error conditions branch to a common state and then process the error condition in the same way. Proper indenting the CASE statement so that the outputs line up in blocks of code make it much easier to read, review and debug.

Don't code at too low-level where you are implementing the language operators. It's much more efficient to use the +,- or * operators and let the synthesizer choose the implementation that matches your timing constraints. The number of bugs per line of code has widely been accepted to be a constant. So the fewer lines of code, the fewer bugs there should be. Using operators like +,- and * will significantly reduce the number of lines of code and result in fewer bugs and a shorter development time.

2.5 Cross Clock Boundaries Carefully

Have you ever heard of the word "metastability"? Digital engineers think the world is a binary place. Unfortunately the world is very much analog, especially when crossing clock boundaries. A DFF will go metastable if the D input changes within the setup/hold time requirements of the DFF. This means that the Q output of the DFF will NOT be at a valid logic level for some amount of time. The probability that the Q output remains at an invalid logic level is exponential with time. Note that the probability of the DFF going metastable isn't exponential, it's that the amount time that it IS metastable is exponential. Read that sentence again. If you violate the setup/hold of a DFF (which will ALWAYS happen if the signal is asynchronous) then the DFF WILL go metastable. BAD THINGS WILL HAPPEN while the DFF is metastable unless you design for it up front.

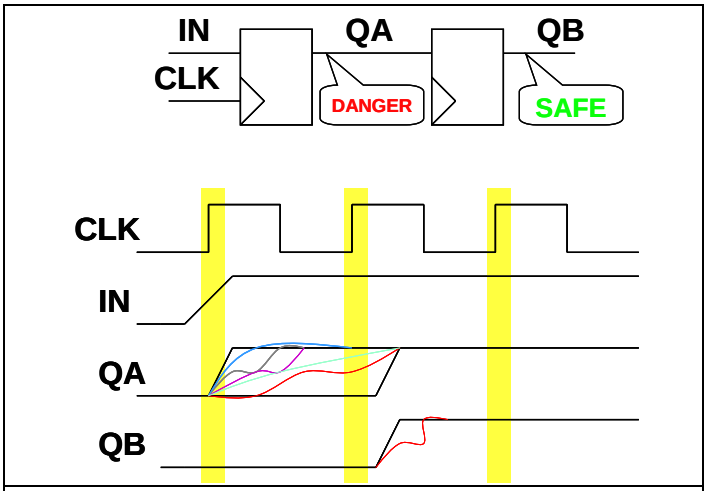


Figure 1 Metastability Example
When IN changes in the "danger" zones of the DFF (when the CLK goes high), QA will be at an invalid state for an exponential amount of time. QA could go immediately to a 0 or 1, or may follow on of the other tracks and extend all the way to the 2nd clock edge and cause even the 2nd DFF to go metastable.

The biggest problem with metastability is that it is nearly impossible to simulate. Logic simulators don't have a "metastable" state. To get the exact picosecond when the D changes relative to the CLK pins in order for the DFF to go metastable may take centuries of simulation time. Thus, you must design clock boundary crossings to be correct by construction.

The classic solution is to double flop the signal. Double flopping generally gets the probability of a metastable signal passing thru the 2nd DFF down to the point where it basically won't happen in our lifetime. The probability depends heavily on the clock frequency, the toggle frequency of the asynchronous signal and the silicon technology. Thus, synchronizing a pushbutton signal pressed by a human finger with a 10Mhz clock and decent CMOS technology will result in a likelihood of a metastable state to be perhaps a century or more (an acceptable risk). However, a phase locked 10GHz clock/data recovery circuit on a poor CMOS technology may go metastable every few minutes.

Ideally the signal should come directly from a DFF in the other clock domain and go directly into a DFF in the new one. This will ensure there are no glitches or other signal integrity problems with the incoming signal (things are already bad enough). The Q output of the 1st DFF goes exclusively into the D input of the 2nd DFF which is clocked on the same edge as the 1st one. It is often tempting to clock the 1st stage on the negative edge of the clock so that the signal can pass thru the synchronizers in 1 clock instead of 2. If the clock is relatively slow, say less than 10MHz, then it's probably OK. But if the clock is much faster, then you're eating into the probability that the 1st DFF is still metastable when the 2nd one clocks and significantly increasing the odds that it will be metastable. Once the asynchronous signal has been double flopped, you can then safely use the Q output of the 2nd DFF. DO NOT USE the Q output of the 1st stage. I've often seen circuits where a rising edge detector is built using the output of the 1st stage with the assumption that it is fully synchronous. IT IS NOT. I've seen this circuit in action many times and pulses will just disappear and you'll spend many nights and weekends scratching your head as to why your circuit doesn't work. Metastability can make any circuit malfunction in ways you cannot possibly imagine - or simulate.

"I can safely cross clock boundaries because I grey coded the signals" WRONG!!! If you need to pass the entire contents of a register, counter, state machine state, etc, you must send a valid flag to the new clock domain to indicate that the value is now safe and can be clocked into the new clock domain. Just grey coding will NOT result in a hazard free circuit. Yes, only 1 signal will transition at a time in a grey coded circuit, but that does not insure that the signals can cross a clock boundary safely. It might be possible in a hand-laid-out circuit where you can carefully match all the delays. But in an RTL circuit with lots of automatic tools synthesizing your design, place and routing it and the fact that more than likely you'll false-path the clock domain crossing, then grey coding will NOT result in a working circuit.

http://www.fpga-faq.com[2] has an interesting article on Metastability and has links to other interesting articles.

2.6 Use Hierarchy Wisely

I've seen RTL code with hierarchy levels that included nothing more than one DFF or even 1 NAND gate! On the flip-side, I've seen RTL code with over 5,000 lines of code and multiple state machines all in one file. Somewhere in between is Nirvana. Think about the logical

[2] http://www.fpga-faq.com/FAQ_Pages/0017_Tell_me_about_metastables.htm

partitioning of your design and also consider the physical implementation as well. Hierarchy often follows clock domains or power domains. Try to minimize the interconnect between blocks. If the signal list is more than 2 pages, then perhaps the block should be folded into the next higher layer of hierarchy. Every signal that goes up and down hierarchy levels has to be typed into multiple files so the fewer the signals that traverse the hierarchy the less work there is in maintaining it. Think about the signals that are passed between blocks. It is often better to place the registers that control a block in that block and wire the bus to read/write the registers rather than thread all of the signals from the registers to/from the block. Often this will also help out in the physical implementation as well.

Look for places where you can design the same logic once, perhaps with a few parameters, and reuse it multiple times. Coding and debugging the code one time instead of multiple times will save time in the long run. Recall that bugs/line of code is a constant so if you can code a module once, debug it once, and reuse it multiple times you will reduce the number of bugs and accelerate your schedule.

Do not rename signals when passing down thru hierarchy. Renaming signals makes it very hard to follow the logic. The exception to this rule is if there is a module that is generic and is likely to be reused multiple times. In this case, the names should be generic like "addr", "datain", "dataout", "write_enb" and so on.

Try to pass only bits of a bus into a module that are actually used. Often a group of control signals are concatenated into a bus to make it easier to thread thru the hierarchy. It's often easy to simply pass this entire bus down into a submodule that may only use 1 or 2 bits of the bus. But... all of those extra bits will cause lint and DFT violations. It is better to break the bus up into the pieces and only pass in what is actually going to be used. Don’t concatenate signals with different timing into a bus as timing parameters are usually applied to the entire bus.

The RMM requires that there is only one verilog module per file and that the filename exactly match the module name. This convention makes it much easier to find and follow a complex designs hierarchy where files are often widely dispersed across numerous directories. Prefixing module names with a few characters that help identify the block as belonging to a particular IP also helps locate problems quickly.

2.7 Parameterize Where it Makes Sense

The RMM recommends never using a constant and always using a parameter instead. That's a bit of overkill. However, there is a happy medium. Most "constants" should be parameterized. Specifically bit widths of something that may want to change from 8 bits to 12 bits or more in the future. Use the verilog "localparam" instead of "parameter" for constants that are only used in the current file.

Verification of parameters is very difficult. Theoretically all possible combinations of parameter values would have to be verified - a 2**n problem. This is clearly impractical. The better solution is to DOCUMENT what values of a parameter have been tested. In the comment line that describes the parameter (you do have a comment for every parameter don't you???) simply

list the valid values for the parameter and the values that have been tested. That way if someone tries a new value for the parameter they'll know if they are using a known working configuration or if they are blazing new trails and may have to do a little debugging/fixing.

Verilog-2001 added the FOR-GENERATE construct which had already proven to be one of the most powerful constructs in VHDL. FOR-GENERATE enables the instancing of a parameterizable number of objects. For-Generate is a very powerful and easy to use feature of the language. Be sure your Verilog reference book includes details on using FOR-GENERATE.

2.8 Warnings are NOT OK

DC_SHELL, LEDA, LEC and other tools that will "compile" the RTL will produce Errors, Warnings and Informational messages. Obviously errors have to be fixed no matter what, but warnings are often ignored. Always review warnings and where ever possible, eliminate them. The more warnings, the more chaff that has to be threshed and the more likely a true problem will slip thru. Always review the logs of all compilation tools. Review informational messages every now and then but these can generally be ignored.

Lint tools should always be used on RTL, especially Verilog. Most lint tools such as LEDA are very easy to use and with just a few clicks can immediately point out obvious errors or warnings in the RTL code. VHDL is a strongly typed language and often catches numerous "typos" at compile time. Verilog however has much looser rules and often a bug will slip thru and take hours or even days to find a simple typo. Lint tools will highlight these errors or warnings quickly. Often engineers complain that the lint tools are too picky and result in far too many warnings that are not actually problems. The CAD department should work with designers to come up with a set of rules that quickly highlight potential problems without spewing too much chaff.

Lint in 11 steps:
The number one reason engineers give for why they didn't run lint on their code is because it's "too hard" or "I don't know how to run it".

1) type "leda"
2) Click on New Project
3) Clock on OK
4) Click Next
5) Click Next
6) Click Next
7) Click Add in the Files window
8) Click on your filename(s)
9) Click OK
10) Click Next
11) Click Finish

The GUI is very intuitive and gives you a list of errors in RED that you need to fix. Lint literally takes a few seconds to run. There are no excuses!

2.9 Always Synthesize and Review Gate Level Implementation

Synopsys has a great tool for visualizing what your RTL will look like at the gate level. It's called "Design Vision". Design Vision is basically a GUI front end to DC_SHELL. To invoke Design Vision, just type design_vision from the same directory as your RTL code. Then click on FILE->READ and select the Verilog files you want to see graphically. Then click on SCHEMATIC->New Schematic Design View or click on the little AND gate icon. A window will

```
//sample RTL code for Design_vision
reg [3:0] dataout;
always @(posedge clk or posedge
reset) begin
    if (reset) begin
        dataout <= 32'b0;
    end else begin
       if (index%control) begin
           dataout <= datain;
       end
    end
end
```

pop up with a schematic representation of your design similar to the one shown here.

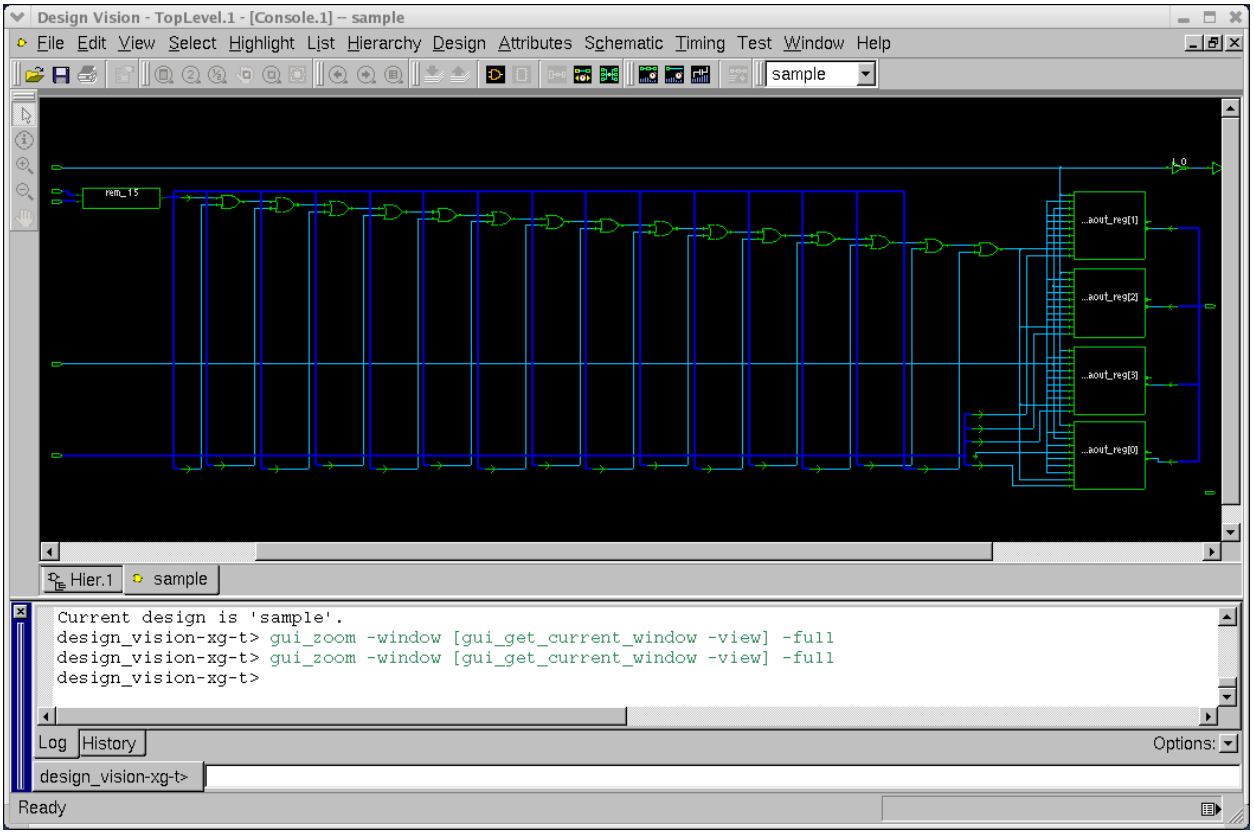


Figure 2 Design Vision High Level Schematic

This first view is a high-level version with the Design Ware objects shown as large blocks. This level is great for getting an idea of what the structure of your code looks like graphically. The more interesting view however is to synthesize and flatten the design. To do that, just click on DESIGN->COMPILE DESIGN. You'll need at least a technology library for this and the easiest way to have a default technology is to place a file (called .synopsys_dc.setup) in your home directory or in the current working directory.

Once the design has been compiled, the schematic window will automatically close as it is no longer valid. Open a new one by clicking on the AND gate icon again and this time you'll get a good idea of the logic depth of your circuit. The schematic should look something like this:

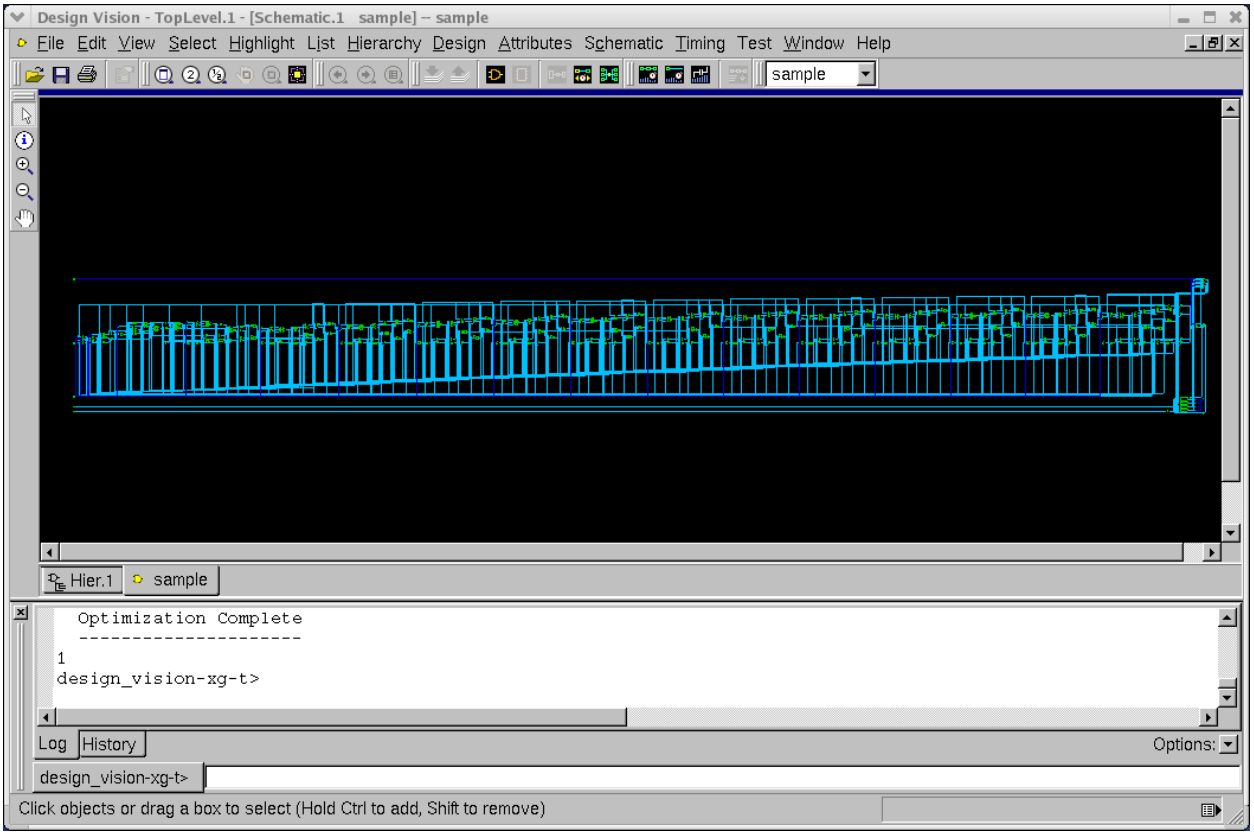


Figure 3 Design Vision Low Level Schematic

Uh-oh! How in the world did we get such a giant mess of logic with just a few lines of code? We'll never meet timing!!! This is primary value of reviewing your RTL in Design Vision. It is very hard to visualize how many gates and how many levels of logic just a few lines of RTL can turn into. In the sample code here the culprit is the modulo operator % used in the IF statement. Modulo is a division and division is very difficult in hardware. But DC_SHELL will blindly install whatever logic you specified in the RTL. The objective of this exercise is NOT to review each and every gate. What you're looking for is unusually deep levels of logic, excessive amounts of logic for what you thought was a simple function or too much logic on critical paths.

With just a little recoding to replace the modulo operator with a simple bit-select, we get the amount of logic we were expecting for this design, 4 DFFs and a little logic in front. You can explore the timing of the design by clicking on TIMING and generating various reports including a histogram of slack. You'll want to setup some timing constraints first via the ATTRIBUTES menu or by reading in a constraint file.

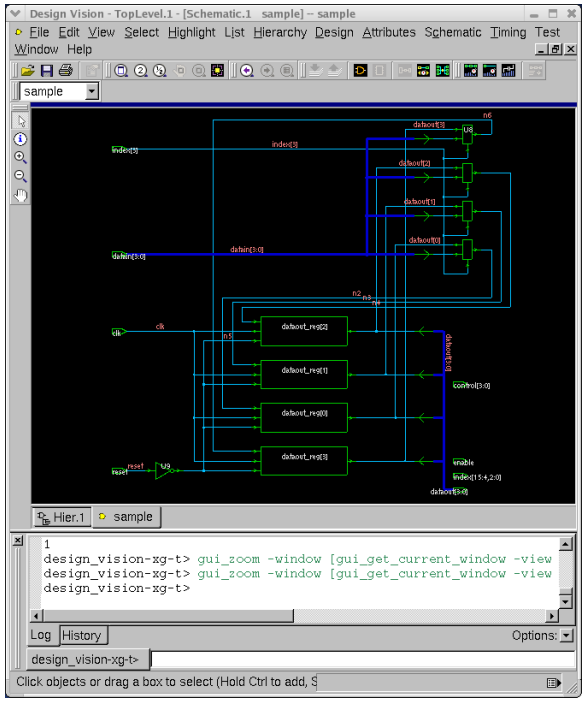


Figure 4 Final Schematic with corrected RTL

Always review the final gate level netlist. Look for SYNOPSYS_UNCONNECTED signals or Logic* signals which indicate that you have unused signals or tied off logic that could be further optimized away. Look for latches and be sure there aren't any. If there are latches in the netlist, then you've probably got some bad RTL. Lint will help find latches or poor coding styles. Also review the synthesis log file and review all errors and warnings. Check that the clock and reset signals are not buffered as they'll get buffer trees added by the Place & Route tools. Review the timing report of the maximum delay and especially look for any inputs that have an asynchronous path directly to an output. Hopefully you've registered all of your outputs.

Use the command REPORT_REFERENCE to get a list of cells used in your design. Look for any latches or unusual cells (like tristate drivers). This is always a good practice to insure your RTL code is of high-quality.

2.10 If the Code Has Not Been Tested, It Does Not Work

If I only had a nickel each time an engineer made a change and didn't bother to rerun a simulation because "It was a small change, I know it works", I'd be a gazillionare now. If you haven't run a simulation that puts the RTL code thru all combinations, it doesn't work. Always write a small testbench that verifies that the RTL works at a basic level. This small testbench doesn't need to test everything. But reading/writing a few registers and passing a little data will only take a couple of hours to write and will save days of debug at the system level. Not every block needs a module level testbench. If a set of files makes a clean boundary for testing then the RTL could be tested as a group.

Learn how to run the simulator and collect code coverage statistics. For VCS, use the -cm_pp gui option to load up the coverage data and quickly find the lines of code that are difficult to cover. Add the following options to the VCS invocation line to collect coverage data.

-cm line+path+branch+cond+fsm+tgl -cm_ignorepragmas -cm_line contassign -cm_noconst -cm_glitch 5 -cm_name $(TEST)

3.0 Conclusions and Recommendations

Coding RTL looks a lot like writing software, but it will eventually be cast into unforgiving expensive silicon where you cannot simply download a new rev if there are any bugs. The Commandments presented here provide a few golden rules to live by which will reduce the likelihood of bugs. Clean designs are more likely to be reused in future generation products. The progression of Moore's law means there are acres of silicon to be filled. Those acres of silicon need good clean RTL code.

COMMENT YOUR CODE!!!

4.0 Acknowledgements

I wish to acknowledge several engineers at Cypress who reviewed and helped formulate the RTL commandments: Timothy Houlihan, Johnie Au and Srinivasa Rao Prerepa.

I also want to thank Jonah Probell for reviewing this paper.

5.0 References

Reuse Methodology Manual (RMM) (Michael Keating & Pierre Bricaud ISBN:1-4020-7141-8)

"The Ten Commandments of Excellent Design" – Peter Chambers 1997
http://asic-world.com/code/verilog_tutorial/peter_chambers_10_commandments.pdf

Sunburst Design has a number of valuable SNUG tutorials on their web site at:
http://www.sunburst-design.com/papers/

Stuart Sutherland teaches Verilog and also has a number of applicable papers at:
http://www.sutherland-hdl.com/papers-by-sutherland.php

"The Ten Edits I Make Against Most IP (And Wish I Didn't Have To)"
Wilson Snyder SNUG 2007
http://www.veripool.org/papers/TenIPEdits_SNUGBos07_paper.pdf

Metastability articles: http://www.fpga-faq.com