CODE QUALITY ESSENTIALS FOR HIGH RELIABILITY FPGAS

Abstract

A recent study showed 68% of FPGA projects are behind schedule and 83% have a significant bug escape into production. This paper will provide tips and tricks to improve reliability on of your FPGA designs.

The “I can just fix it” FPGA design attitude is a major reason a recent study showed that 68% of FPGA projects are behind schedule and 83% of have a significant bug that escapes into production. On top of that, code developed this way has a habit of sticking around to become what’s known as “legacy code,” the code that no one wants to touch because it’s so poorly written that no one on the team has any chance of understanding what it’s really doing.

When designing FPGAs, code quality is essential to staying on schedule, avoiding design iterations and, worse, bugs found in production. This is especially true when it comes to high reliability applications such as automotive, space and medical devices where bugs can be extremely expensive or impossible to fix. But just what makes RTL (VHDL or SystemVerilog) quality code? The answer is, well, there isn’t just one thing that makes for quality code. It’s a combination of considerations from readability to architecture choices.

In this white paper, we will investigate and do a deep dive into specific aspects of “quality code.” This includes readable and maintainable RTL code, Finite State Machine (FSM) architectures and coding guidelines and finally the challenges around multiple Clock Domains.

Readable and maintainable code

Readable and maintainable code takes some common discipline, starting with the basics such as naming conventions. It's not so much whether the organization prefers big endian or little endian, spaces or tabs, prefixes or suffixes, underscores or hyphens or camel case, the important thing is to have a standard and stick to it. By standardizing on naming conventions for architectures, packages, signals, clocks, resets, etc., code becomes clearer and as a side benefit, it can reduce the code complexity. Going through code by hand to uphold these standards is incredibly tedious, but the task can be easily automated with a static analysis tool.

A simple and common rookie mistake is to hard coded numbers, especially in shared packages. While the original coder may understand exactly the specific hard coded number, 10 years down the line, when it’s time to update the device or functionality, the use of a hard coded number will add to confusion and delays.

Simulation vs synthesis mismatches

While correct simulation of code is important, it doesn’t necessarily mean that the code is free from issues. For example, variables that are used before they’re assigned might be unknown or might retain a previous value, which means possible mismatches between the simulation and actual functionality after synthesis. This simple mistake can mean the design works in simulation but then later fails in the lab or, worse, in the field.

Another common example of code quality issues is missing assignments in IF and Else blocks and case statements that will cause most synthesis tools to create latches in the design along with the registers. Implied latches can cause issues with timing, and different synthesis tools will do this in different ways, so change vendors, change implementations.

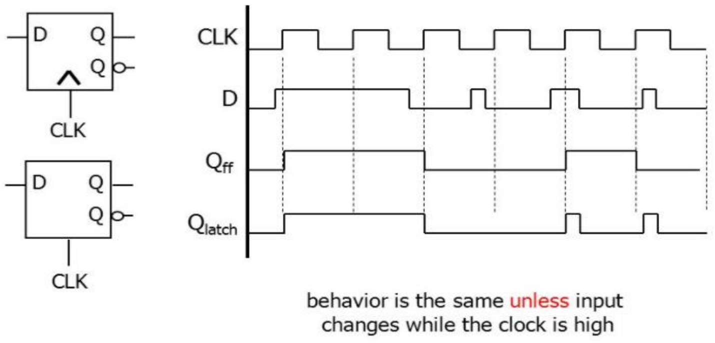

D
Q
Q
CLK
D
Q
Q
CLK
CLK
D
Q_ff
Q_latch
behavior is the same unless input changes while the clock is high

Figure 1 Latch vs Flip Flop

The goal of any tool is to succeed, and a synthesis tool wants to synthesize successfully, so it may give your code the benefit of the doubt and assume the code is right until it's proven wrong. Many tools will even accept common poor practices and "fix" them. Again, different tools, different "fixes."

Reset issues

Another potential code quality issue is synchronous de-assertion of asynchronous resets. Some pay little attention to how their reset will work in the real world. They just throw in a global reset just so the simulation looks "tidy" at Time Zero, but this isn't enough. That reset must work in the real world. Keep in mind, a global reset is just that, global, so it can have a large fanout that causes significant delay and skew, so it should be buffered properly. And because this reset is asynchronous, it can happen at any time, and it forces the flip-flops to a known state immediately.

This is not a problem; however, the de-assertion issue comes not when the reset pulse begins, but when it ends relative to the active clock edge. The minimum time between the inactive edge of your reset and the active edge of your clock is called recovery time. Violating recovery time is no different than violating setup or hold time. The easiest way to avoid this issue is to design the reset as shown below. The active edge can happen at any time, but the inactive edge is synchronous with the clock.

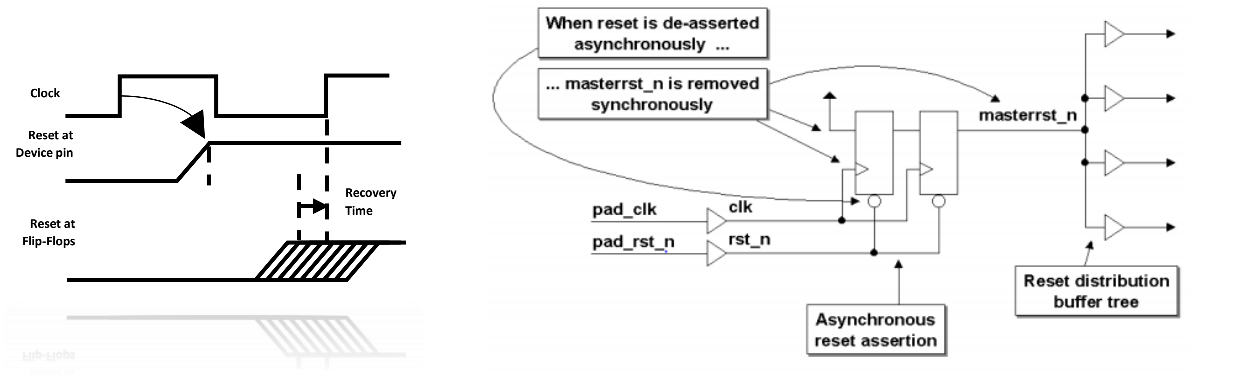
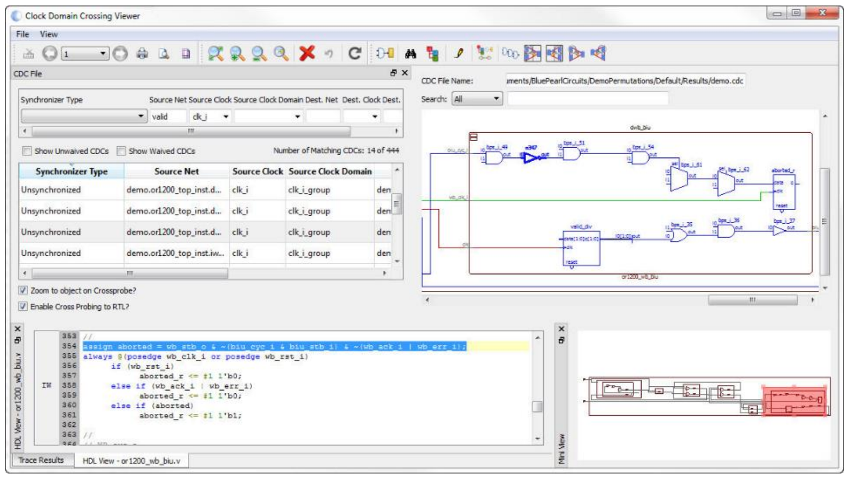

Clock
Reset at Device pin
Recovery Time
Reset at Flip-Flops
When reset is de-asserted asynchronously ...
... masterrst_n is removed synchronously
masterrst_n
pad_clk
clk
pad_rst_n
rst_n
Asynchronous reset assertion
Reset distribution buffer tree

Figure 2 Asynchronous Reset Circuit

Finding and addressing code quality issues, such as naming conventions, reset issues, excessive area, low frequency, and meantime between failures up front, as you code can significantly reduce the number of iterations through synthesis and place and route, improving productivity, reducing development costs, and improving the reliability of a design.

The Visual Verification Suite provides RTL Analysis to identify coding style and structural issues up front. The RTL Analysis points out naming conventions as well as structural issues such as long paths and if-then-else analysis as you code, rather than late in the design cycle.


Clock Domain Crossing Viewer
File View
CDC File
CDC File Name:
Search: All
Synchronizer Type
Source Net Source Clock Source Clock Domain Dest. Net Dest. Clock Dest.
valid
clk_i
Show Unwaived CDCs
Show Waived CDCs
Number of Matching CDCs: 14 of 444
Zoom to object on Crossprobe?
Enable Cross Probing to RTL?
Trace Results
HDL View - or1200_wb_biu.v
Mini View

Synchronizer Type	Source Net	Source Clock	Source Clock Domain	
Unsynchronized	demo.or1200_top_inst.d...	clk_i	clk_i_group	den
Unsynchronized	demo.or1200_top_inst.d...	clk_i	clk_i_group	den
Unsynchronized	demo.or1200_top_inst.d...	clk_i	clk_i_group	den
Unsynchronized	demo.or1200_top_inst.iw...	clk_i	clk_i_group	den

```
//
assign aborted = wb_stb_o & ~(biu_cyc_i & biu_stb_i) & ~(wb_ack_i | wb_err_i);
always @(posedge wb_clk_i or posedge wb_rst_i)
        if (wb_rst_i)
                aborted_r <= #1 1'b0;
        else if (wb_ack_i | wb_err_i)
                aborted_r <= #1 1'b0;
        else if (aborted)
                aborted_r <= #1 1'b1;

//
```

Figure 3 Visual Verification Suite

So, what is it that makes the suite such a powerful debugging environment? It has a straightforward, understandable graphical user interface that runs on Windows or Linux, and quickly generates reports that show aspects of your design, like the highest fanout nets, or the longest if-then-else chains, and an easy-to-filter report window showing the specific issues it has found. The suite also includes numerous checks to catch violations of easily created company specific naming conventions.

FSM Architectures and Coding Guidelines

There are different methods or protections that can be used on state machine designs. If a single-event upset (SEU, a radiation- induced change in the bit in one flip-flop) occurs in a state register, the FSM that contains the register could go into an erroneous state or could "hang," by which is meant that the machine could remain in undefined states indefinitely or requiring a reset of the FSM as the only way to recover.

Obviously, in many applications, this is not acceptable or is completely impractical. Typically, the designer wants the state machine to be able to either continue operating (tolerance) or detect a failure and fail safe (detection).

To ensure reliability of the state machine, the coding scheme for bits in the state register must satisfy the following criteria:

1. All possible states must be defined
2. A SEU brings the state machine to a known state
3. There is no possibility of a hang state
4. No false state is entered
5. A SEU exerts no effect on the state machine

Tolerance

When it comes to tolerance there are two mechanisms we can use:

- **Triple Modular Redundancy (TMR)** — With TMR three instantiations of the state machine are created, and the output and current state are voted upon clock cycle by clock cycle. TMR is a good approach, but it does require three implementations and ensuring physical separation of the three implementations to ensure only one FSM is corrupted. For more on TMR we recommend that you view Xilinx Isolation Design Flow. This documented flow can be especially useful for ensuring physical separation in the chip.

- **Hamming Three Encoded** — With a Hamming three encoded state machine, each state is encoded with a hamming distance of three between them. This prevents a SEU from changing between valid states, as the SEU can only flip a single bit. In a Hamming three, each state has several adjacent states which also cover the possible states to which a SEU could change the Hamming three state. This adjacent state behaves the same as the original state, hence allowing the state machine to tolerate the SEU and keep operating. It does, however, mean the number of states declared is large. For a 16 state FSM encoded sequentially, seven bits are needed to encode the 16 states separated by a Hamming distance of three. This means there are N * (M+1) states required, where N is the number of states and M is the register size.

Both the TMR and Hamming three structures require considerable effort from the design engineer unless the structure can be implemented automatically by the synthesis tool.

Detection

When it comes to detection, the structures used are considerably simpler:

- Hamming Two (sequential + parity) — This method encodes the state with a Hamming distance of two. Should a SEU occur, the error can be found using a simple XOR network and the state machine can be recovered to a safe state to recommence operation.

- Default Case / When Others —This method uses the Verilog *default* or VHDL *when others* to implement a recover state. This does require that the synthesis tool does not ignore the *default* case or *when others*, and that the user does not define them as null or do not care.

Hamming Two is the best compromise in terms of size, speed, and fault-tolerance and is preferred over both binary and one-hot state machine encoding for high reliability designs. Hamming Three encoding is the best fault-tolerant to single faults and, therefore, preferred when ultimate reliability is required in a critical application, however it is slower and larger than other implementations.

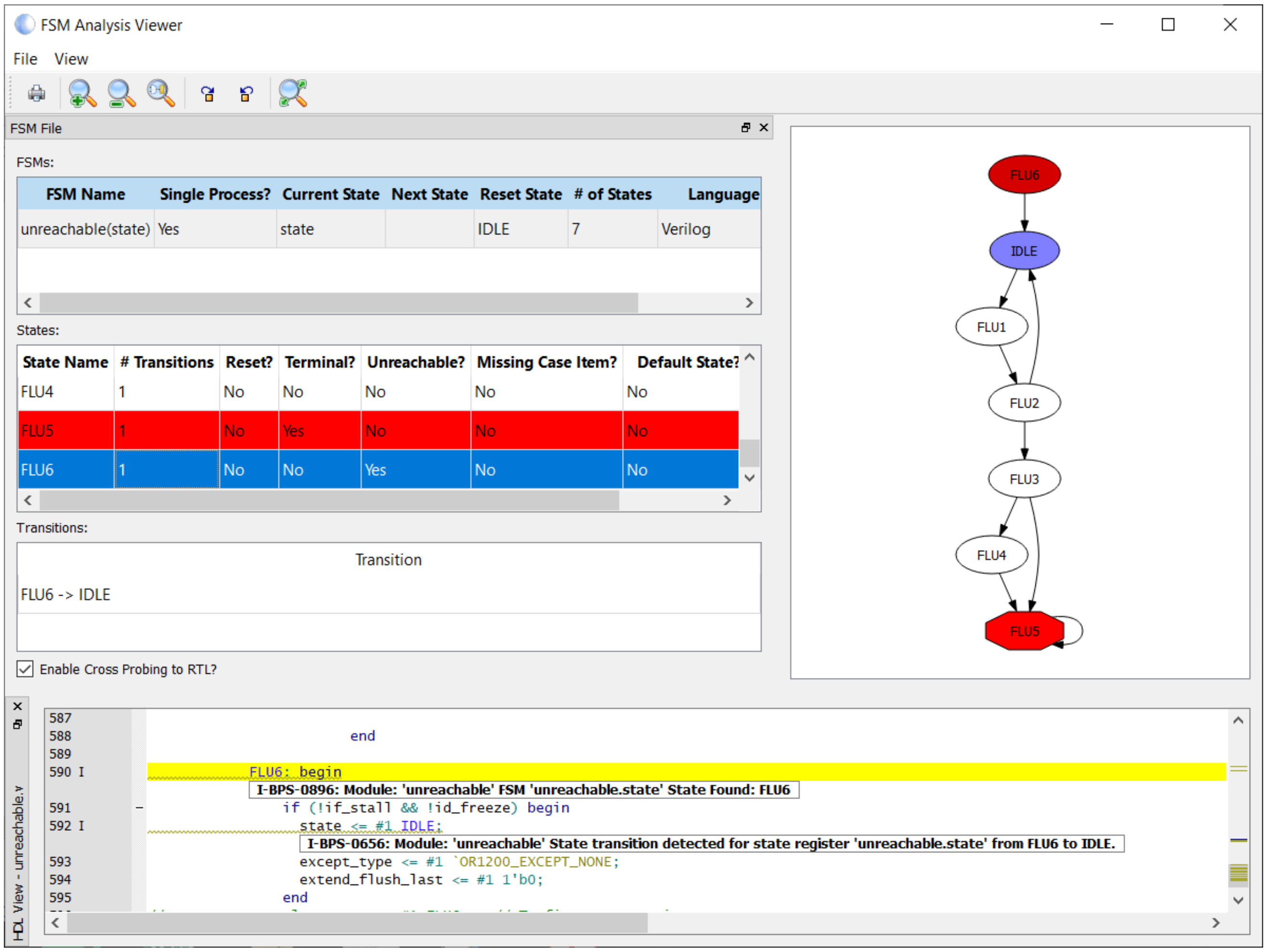
Static Verification of FSMs

For highly reliable FSMs, all possible states must be defined and there must be no possibility of a hang state. Functional verification of an FPGA's RTL requires a considerable simulation effort to achieve code coverage (branch, path, toggle, condition, expression etc.).

To achieve a high level of coverage the simulation must test several boundary and corner cases to observe the behavior of the unit under test and ensure its correctness. This can lead to long simulation runs and significant delays between iterations when issues are detected. Of course, issues found in simulation can range from functional performance such as insufficient throughput to state machine deadlocks due to incorrect implementation during the coding process.

This is where static analysis tools such as the Visual Verification Suite can be wonderfully complementary to functional simulation and can help save considerable time and effort in the verification stage, when code coverage is being investigated.

Static analysis enables functional verification to be started with a much higher quality of code, reducing iterations late in the verification cycle. In addition, static analysis typically also runs in tens of seconds to minutes compared to long running simulations. The Visual Verification Suite provides an FSM viewer and reports that help pinpoint any issues, up front, as you code.



FSMs:

FSM Name	Single Process?	Current State	Next State	Reset State	# of States	Language
unreachable(state)	Yes	state		IDLE	7	Verilog

States:

State Name	# Transitions	Reset?	Terminal?	Unreachable?	Missing Case Item?	Default State?
FLU4	1	No	No	No	No	No
FLU5	1	No	Yes	No	No	No
FLU6	1	No	No	Yes	No	No

Transitions:

Transition
FLU6 -> IDLE

☑ Enable Cross Probing to RTL?

```
                    end

            FLU6: begin
              I-BPS-0896: Module: 'unreachable' FSM 'unreachable.state' State Found: FLU6
              if (!if_stall && !id_freeze) begin
                state <= #1 IDLE;
                I-BPS-0656: Module: 'unreachable' State transition detected for state register 'unreachable.state' from FLU6 to IDLE.
                except_type <= #1 `OR1200_EXCEPT_NONE;
                extend_flush_last <= #1 1'b0;
              end
```

Figure 4 Visual Verification Suite FSM Viewer

Many of the suite's predefined rules focus upon the structural elements which may be incorrect in the design such as:

- Unnecessary events – These are unnecessary signals included in the sensitivity list. Such inclusion will lead to simulation mismatch and add complexity to achieving code coverage.
- If-Then-Else Depth – This will analyze the If-Then-Else structures to identify deep paths which may impact timing performance and throughput when implemented.
- Terminal State – This is a state in a state machine which once entered has no exit condition. Finding this prior to simulation can save wasted simulation time.

- Unreachable State – This is a state in a state machine which has no entrance condition. Finding this prior to simulation can again save considerable simulation time.
- Reset – This ensures each flip flop is reset and reset removal is synchronous to the clock domain of the reset. Several in-orbit issues have been detected relying upon the power-on status of registers and as such reset for all flip flops is best practice.
- Clocking – Clocking structures are also analyzed to ensure there is no clock gating or generation of internal clocks.
- Safe Counters – Checking of counters to ensure that terminal counts use greater than or equal to for up counters and less than or equal to for down counters will ensure that single event effects have a reduced impact on locking up counters.
- Dead or unused code – Analyzes and warns about unused or dead code in the design. This can be removed prior to functional simulation and reduces head scratching when code coverage cannot be achieved.

Multiple Clock Domains

Modern designs often contain several clock domains. Of course, information needs to be shared between these domains. Incorrect synchronization of data and signals between clock domains can result in metastability and corrupted data. In some systems this incorrect data can be catastrophic. The understanding of Clock Domain Crossing (CDC) origins boils down to a few simple truths. There is clock drift from separate sources.

At its most basic level, metastability is what happens within a register when data changes too soon before or after the active clock edge; that is, when setup or hold times are violated. A register in a metastable state is in between valid logic states, and the process of settling to a valid logic state takes longer than normal. It will eventually fall into a stable "1" or "0" state, but there is no way to predict which way it will fall or how long it will take.

Think of it as tossing a coin millions of times. There are actually three possibilities: heads, tails, or once in a great while the coin just might stick the landing and end up on its edge, if only for a while. The question is, will that while be longer than a clock cycle? That's metastability.

When data is transferred between two registers whose clocks are asynchronous, metastability will happen. There is no way to prevent it. All you can do is to minimize its impact by placing the two clocks in different clock domains and using a clock synchronization technique at the crossing point. Hence the name "clock domain crossing."

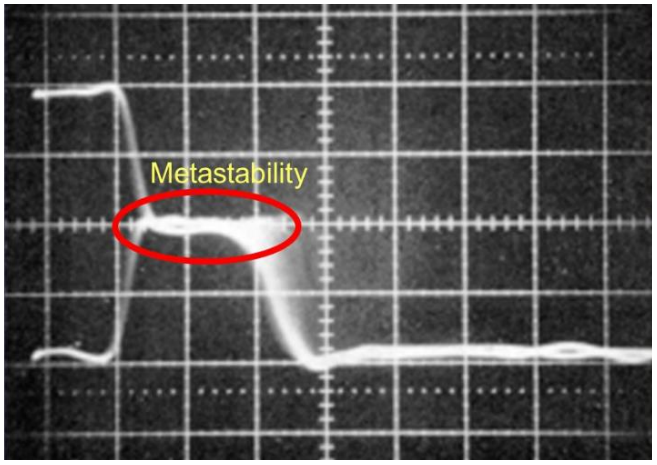


Figure 5 Data Metastability

Putting two clocks into the same clock domain is a declaration that these two clocks are synchronous to each other, and crossings between them do not need to be synchronized. If the clocks are from the same source, or one is derived from the other, then they are synchronous and can be placed into the same clock domain.

Clocks that are asynchronous to one another should always be placed in different clock domains, and any CDCs between them need to be synchronized. Even two clocks of the same frequency should be placed into different domains if they come from independent sources. Unfortunately, two independent clock sources of the same frequency will drift relative to one another over time and cause metastability problems.

Synchronizers

The simplest synchronization method for a single bit is to have two consecutive registers in the receiving domain. This is known as double-register synchronization. By requiring any metastable state that occurs to pass through two registers, it reduces the chance of metastability from $1/r$ to $1/r^2$, which is acceptable for most purposes. Data integrity is maintained only by coincidence. Since it's only one bit, the only two possibilities are that it will happen to match either the preceding clock cycle or the subsequent clock cycle.

One of the most popular methods of passing data between clock domains is to use a FIFO. A dual port memory is used for the FIFO storage. One port is controlled by the sender, which puts data into the memory as fast as one data word (or one data bit for serial applications) per write clock. The other port is controlled by the receiver, which pulls data out of memory; one data word per read clock.

Two control signals are used to indicate if the FIFO is empty, full, or partially full. Two additional control signals are frequently used to indicate if the FIFO is almost full or almost empty. In theory, placing data into a shared memory with one clock and removing the data from the shared memory with another clock seems like an easy and ideal solution to passing data between clock domains. For the most part it is, however generating accurate full and empty flags can be challenging.

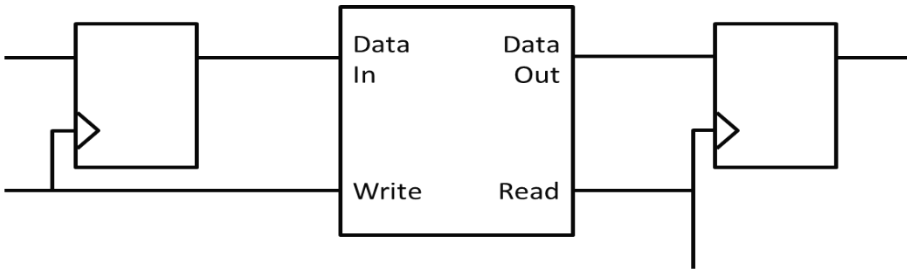


Figure 6 FIFO Bus Synchronization

Another CDC issue that must be addresses is that of data reconvergence, when two data signals are combined after being independently synchronized between the same two clock domains. This is a problem because synchronization is inherently an arbitration to avoid metastability. A new value will be correctly clocked, without metastability, on one of two successive receiving clock cycles. There's no way of knowing which. The two signals in question can be arbitrated differently and can end up being clocked into the receiving domain on different clock cycles when correct operation depends upon their remaining in step. Think again of the coin toss. With a single bit, it's all but certain that the coin will end up either heads or tails, but with multiple bits, you'd need either all heads or all tails. That's a losing bet.

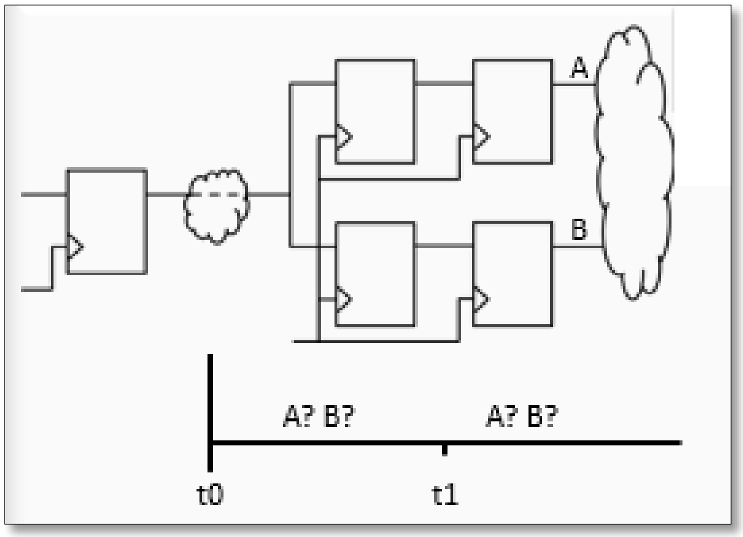


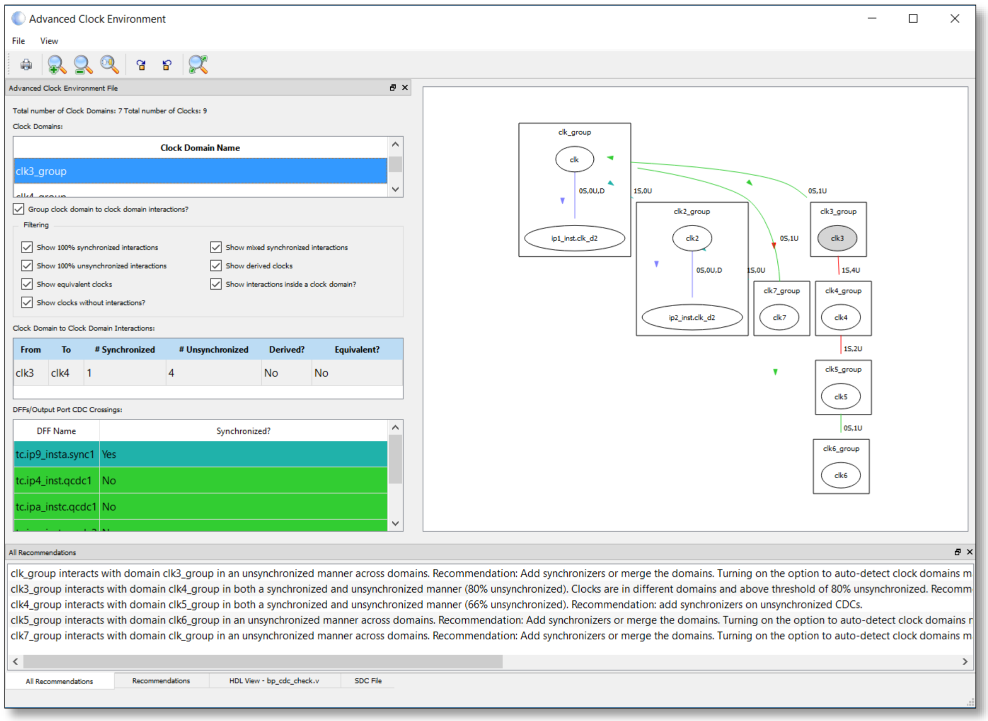
Figure 7 Signal Reconvergence

The implication is that, for a data bus that crosses clock domains, having individual synchronization on each of the bits will not work reliably. One solution is to generate a single bit “data valid” flag which indicates that the data is stable. Synchronize that flag across domains, and then use it to enable the clocking of the data bus into the new domain.

Another solution is to ensure that the data itself is “gray” (only one bit changing on any given clock cycle) with respect to the receiving clock. This is easier when crossing from a slower to a faster domain because you can be sure there will not be multiple changes from the perspective of the receiving domain. The handshake synchronizers use two m-flip-flop synchronizers to generate request and acknowledge signals.

How to identify CDC Issues:

The Visual Verifications Suite’s Advanced Clock Environment (ACE) provides a graphical representation summarizing data paths between clocks and can make recommendations for grouping of clocks into clock domains. With ACE, designers can identify clocks to better understand how they interact with synchronizers in the design. This allows users to quickly identify improper synchronizers or clock domain groupings that cause CDC metastability.



Total number of Clock Domains: 7 Total number of Clocks: 9

Clock Domains:

Clock Domain Name
clk3_group

☑ Group clock domain to clock domain interactions?

Filtering

☑ Show 100% synchronized interactions
☑ Show 100% unsynchronized interactions
☑ Show equivalent clocks
☑ Show clocks without interactions?
☑ Show mixed synchronized interactions
☑ Show derived clocks
☑ Show interactions inside a clock domain?

Clock Domain to Clock Domain Interactions:

From	To	# Synchronized	# Unsynchronized	Derived?	Equivalent?
clk3	clk4	1	4	No	No

DFFs/Output Port CDC Crossings:

DFF Name	Synchronized?
tc.ip9_insta.sync1	Yes
tc.ip4_inst.qcdc1	No
tc.ipa_instc.qcdc1	No

All Recommendations

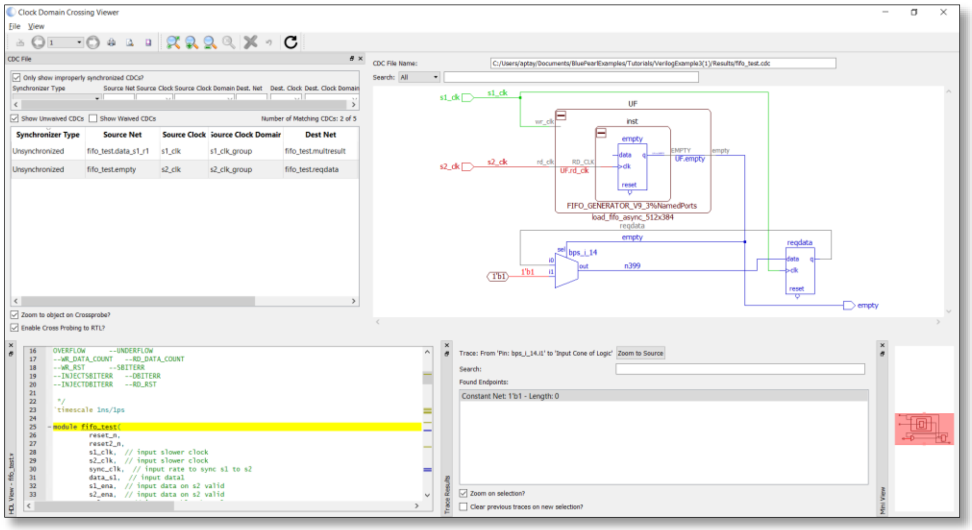
clk_group interacts with domain clk3_group in an unsynchronized manner across domains. Recommendation: Add synchronizers or merge the domains. Turning on the option to auto-detect clock domains m
clk3_group interacts with domain clk4_group in both a synchronized and unsynchronized manner (80% unsynchronized). Clocks are in different domains and above threshold of 80% unsynchronized. Recomm
clk4_group interacts with domain clk5_group in both a synchronized and unsynchronized manner (66% unsynchronized). Recommendation: add synchronizers on unsynchronized CDCs.
clk5_group interacts with domain clk6_group in an unsynchronized manner across domains. Recommendation: Add synchronizers or merge the domains. Turning on the option to auto-detect clock domains r
clk7_group interacts with domain clk_group in an unsynchronized manner across domains. Recommendation: Add synchronizers or merge the domains. Turning on the option to auto-detect clock domains m

Figure 8 Advanced Clock Environment

ACE addresses a fundamental chicken-and-egg problem with automated CDC analysis. To perform a CDC analysis, you first must properly define your clock domains, but in order to automatically define clock domains, you need to perform a CDC analysis. ACE does this by performing a quick-and-dirty CDC analysis that recognizes only double-register synchronization, and then by explicitly assuming that two clocks are in the same domain if a high percentage (80% by default) of CDCs are unsynchronized. Then the clock domains, whether defined automatically or by the user, are analyzed and graphically displayed.

The overall goal of ACE is to enable engineers to find metastability issues in designs by properly grouping clocks into clock domains. Design and Verification engineers use ACE to ensure the clock domains are properly specified before running a CDC analysis. ACE will quickly find errors in clock domain groupings or find/recommend appropriate clock domain groupings for a circuit that is synchronized. Only then can a correct and comprehensive CDC analysis be performed.

Next, the suite's CDC Analysis understands FPGA vendor clocking schemes, saving enormous resources to set up designs. The CDC analysis has built-in intelligence that helps set up the CDC run and rapidly debug issue found using the built-in cross-probing and schematic display.



☑ Only show improperly synchronized CDCs?

Synchronizer Type | Source Net | Source Clock | Source Clock Domain | Dest. Net | Dest. Clock | Dest. Clock Domain

☑ Show Unwaived CDCs ☐ Show Waived CDCs Number of Matching CDCs: 2 of 5

Synchronizer Type	Source Net	Source Clock	Source Clock Domain	Dest Net
Unsynchronized	fifo_test.data_s1_r1	s1_clk	s1_clk_group	fifo_test.multresult
Unsynchronized	fifo_test.empty	s2_clk	s2_clk_group	fifo_test.reqdata

☑ Zoom to object on Crossprobe?

☑ Enable Cross Probing to RTL?

CDC File Name: C:/Users/aptay/Documents/BluePearlExamples/Tutorials/VerilogExample3(1)/Results/fifo_test.cdc

Search: All

```
OVERFLOW      --UNDERFLOW
--WR_DATA_COUNT   --RD_DATA_COUNT
--WR_RST      --SBITERR
--INJECTSBITERR   --DBITERR
--INJECTDBITERR   --RD_RST

 */
`timescale 1ns/1ps

module fifo_test(
          reset_n,
          reset2_n,
          s1_clk,  // input slower clock
          s2_clk,  // input slower clock
          sync_clk,  // input rate to sync s1 to s2
          data_s1,  // input data1
          s1_ena,  // input data on s2 valid
          s2_ena,  // input data on s2 valid
```

Trace: From 'Pin: bps_i_14.i1' to 'Input Cone of Logic' Zoom to Source

Search:

Found Endpoints:

Constant Net: 1'b1 - Length: 0

☑ Zoom on selection?

☐ Clear previous traces on new selection?

Figure 2 Visual Verification Suite CDC Analysis

One of the strengths of the suite's CDC analysis is that it flags all CDCs, whether unsynchronized, properly synchronized, or improperly synchronized. For example, using a double-register scheme on a single bit is perfectly appropriate, but a multi-bit bus requires a more robust synchronization technique. The user even has the option to find what we call "Clock Equivalent" crossings, which are clock-to-clock interactions within the same clock domain.

Visual Verification Suite, used early and often in the design process as opposed to as an end of design/sign-off only tool, significantly contributes to design security, efficiency, and quality, while minimizing chances of field vulnerabilities and failures.

About Blue Pearl Software

Blue Pearl Software, Inc. is a leading provider of DO-254 compatible design automation software for ASIC, FPGA, and IP RTL verification. Our customers are RTL managers and developers, in military, aerospace, semiconductor, medical, communications and safety critical design companies. The Visual Verification™ Suite speeds block and project level verification with advanced integrated RTL structural and formal linting, constraint generation and clock domain crossing analysis. Our usability is unmatched in the industry and can help your design team accelerate development and produce high reliability designs. The Visual Verification Suite is designed, tested and supported in the United States of America. To learn more, request a demonstration today.