Verification of Digital Designs (how to debug)

CSULB
CECS 360
John Tramel

References

- Xilinx documentation: Capabilities to Maximize Productivity for FPGA Debug and Verification
- Xilinx Design Reuse Methodology for ASIC and FPGA Designers
- Programmable Logic Design Quick Start Hand Book
- Debugging by David J. Agans

The Problem

- The more complex digital systems become it becomes equally complex or difficult to verify their correct operation

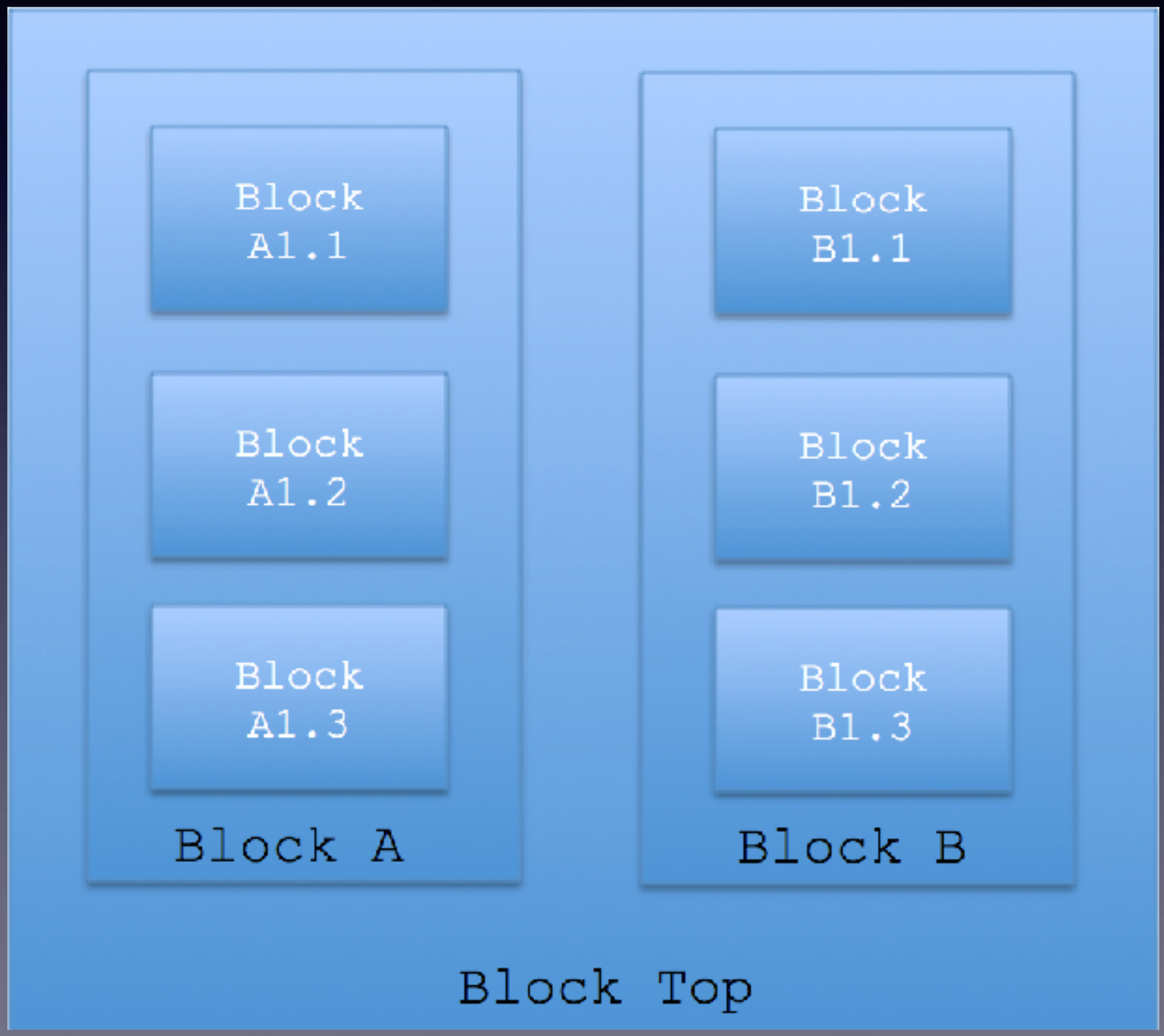
- The simplest and most time efficient approach is to utilize a test fixture (test bench)

- It is up to the designer to decide which modules require test benches and which may be assumed to work correctly - be careful here - a bad assumption here could cause much added debug time

Verification Approach

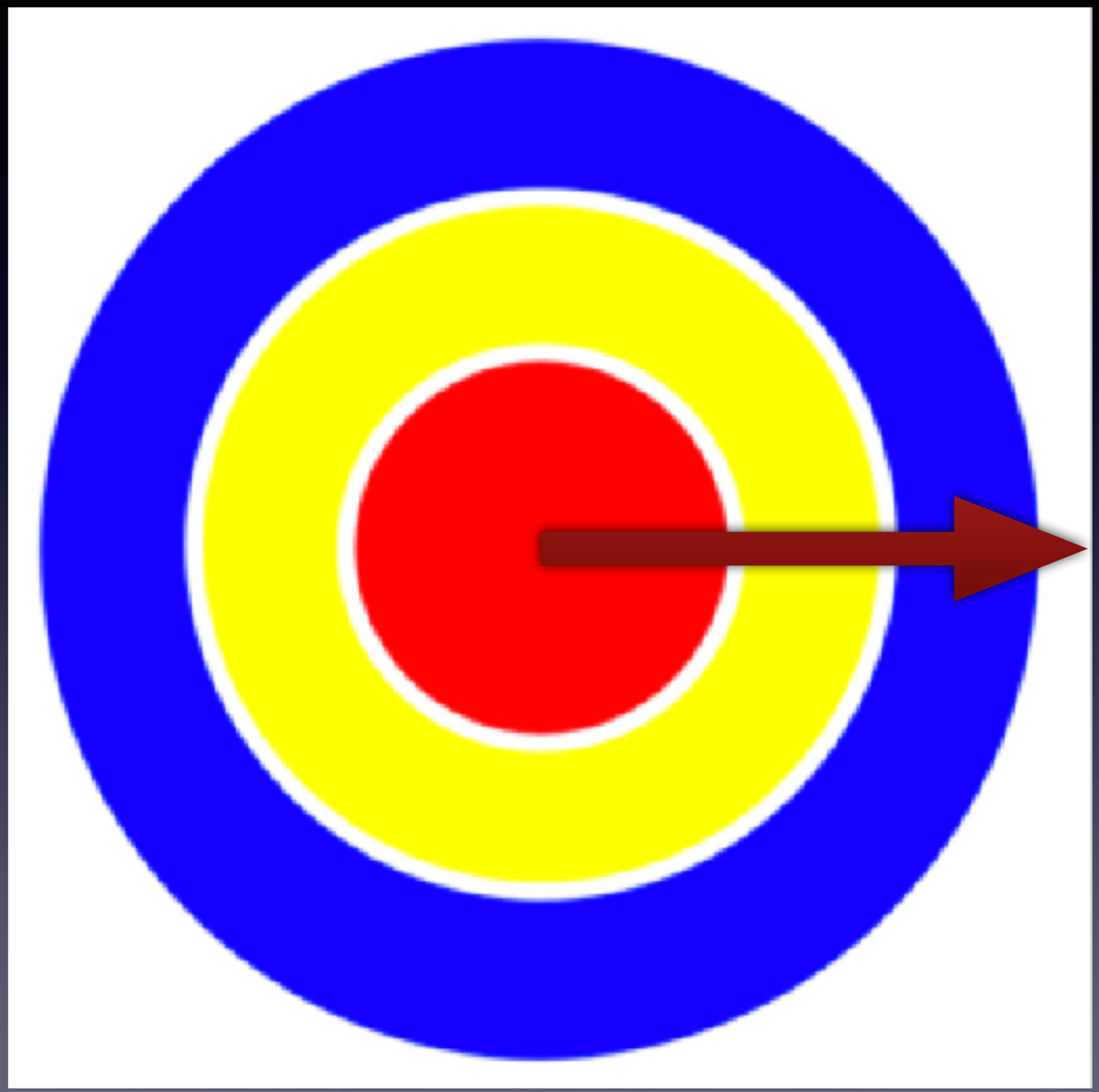
- A time proven approach to verification may be summarized by the phrase “divide and conquer”
- This translates to an approach that before blocks are integrated together into a more complex structure each of the blocks must have been proven to function properly
- This approach requires a patient approach where there is a progressive development of confidence in the proper operation of each component before the whole is verified

Divide and Conquer or Inside Outside Approach



- Blocks A1.1 thru A1.3 must be verified before attempting to verify Block A
- The same sequence for Blocks in B
- Block A /B must be verified before Block Top
- Once all component designs have been verified then Block Top should be verified

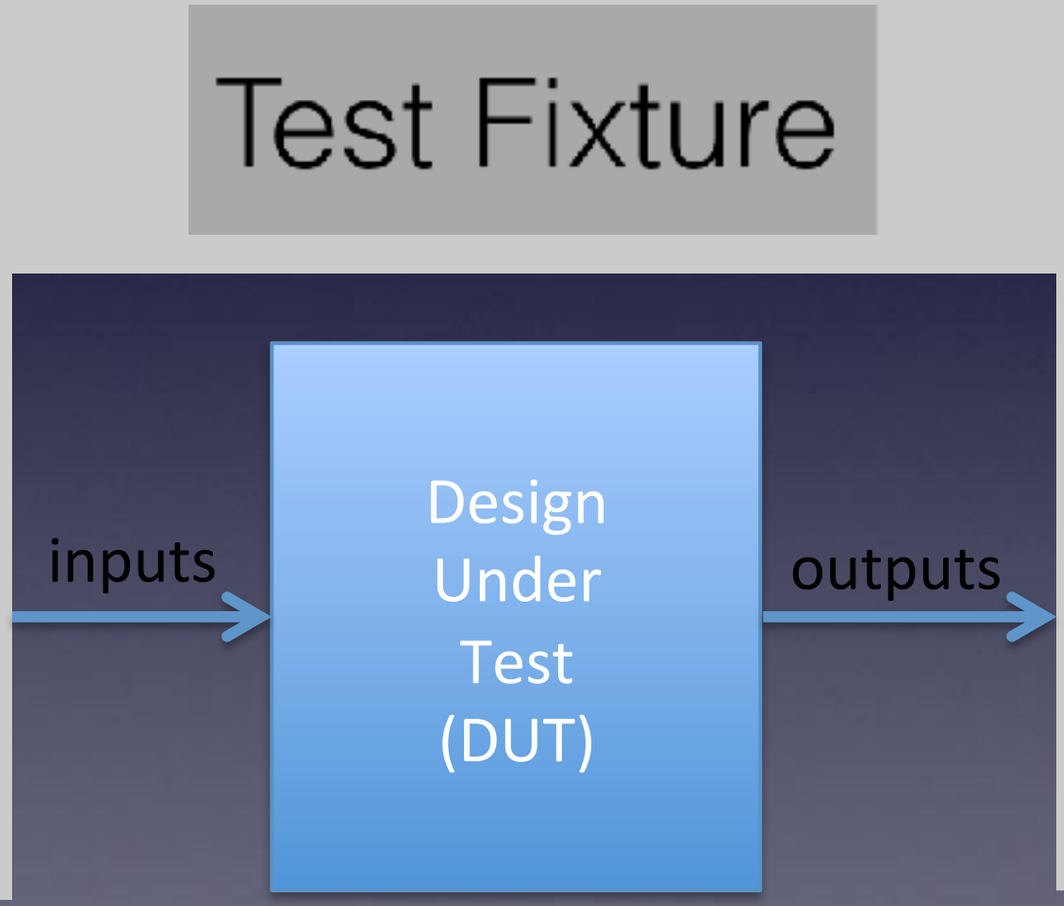
Another Perspective



- Debug from the “inside out”
- Start with what is essential to the operation of the design then move to the supplemental functions
  - Is there a clock?
  - Is reset correct?
  - Is the processor fetching / executing instructions?
- Build confidence step by step rather that assuming immediately that the whole design will function properly - have a method to your madness

Test Fixture Approach

It is the responsibility of the test bench to source all inputs to the DUT. This will include reset, clock, and all data inputs. The reset should be active from time 0 and deactivated after a certain time (say 100ns). The clock should be free running at 50MHz (20ns). The inputs should then be switched in a pattern that mimics the way they will switch in the final system.



The outputs will typically be viewed on the simulator's waveform display (inputs as well). The designer is then needed to determine based on the waveform whether the circuit is behaving correctly. It is possible to design self-checking test benches. This will be considered later in the presentation.

File Creation Guidelines

- The test fixture will be a Verilog module that will have an instantiation of the DUT
- The test fixtures should be named as follows: if the DUT is *design.v*, then the test fixture should be *design_tf.v*
- All Verilog files have the extension .v
- All Verilog files should be named according to the name of the module (this implies that in this methodology there should only be one module per file)

Suggestions

- Build your designs incrementally, don’t try and bite off too much at a time
- Know how each module is supposed to behave before you code it - this makes is easier to build the test fixture
- Locate similar code in the same module - avoid the urge to create modules for simple structures; say a DFF or even a register
- In the simulator take time to arrange the signals in the display so you can follow what is happening. Usually resets and clocks at the top, static signals at the bottom (they don’t switch), then arrange the others by inputs / outputs
- Save the .wcfg file once you have arranged the signals so when you return you do not have to reformat the display

Suggestions

- It is in your test fixtures that you can utilize the high level constructs to make your job easier - here I use a for-loop to walk through all four options making sure to all a clock edge in between

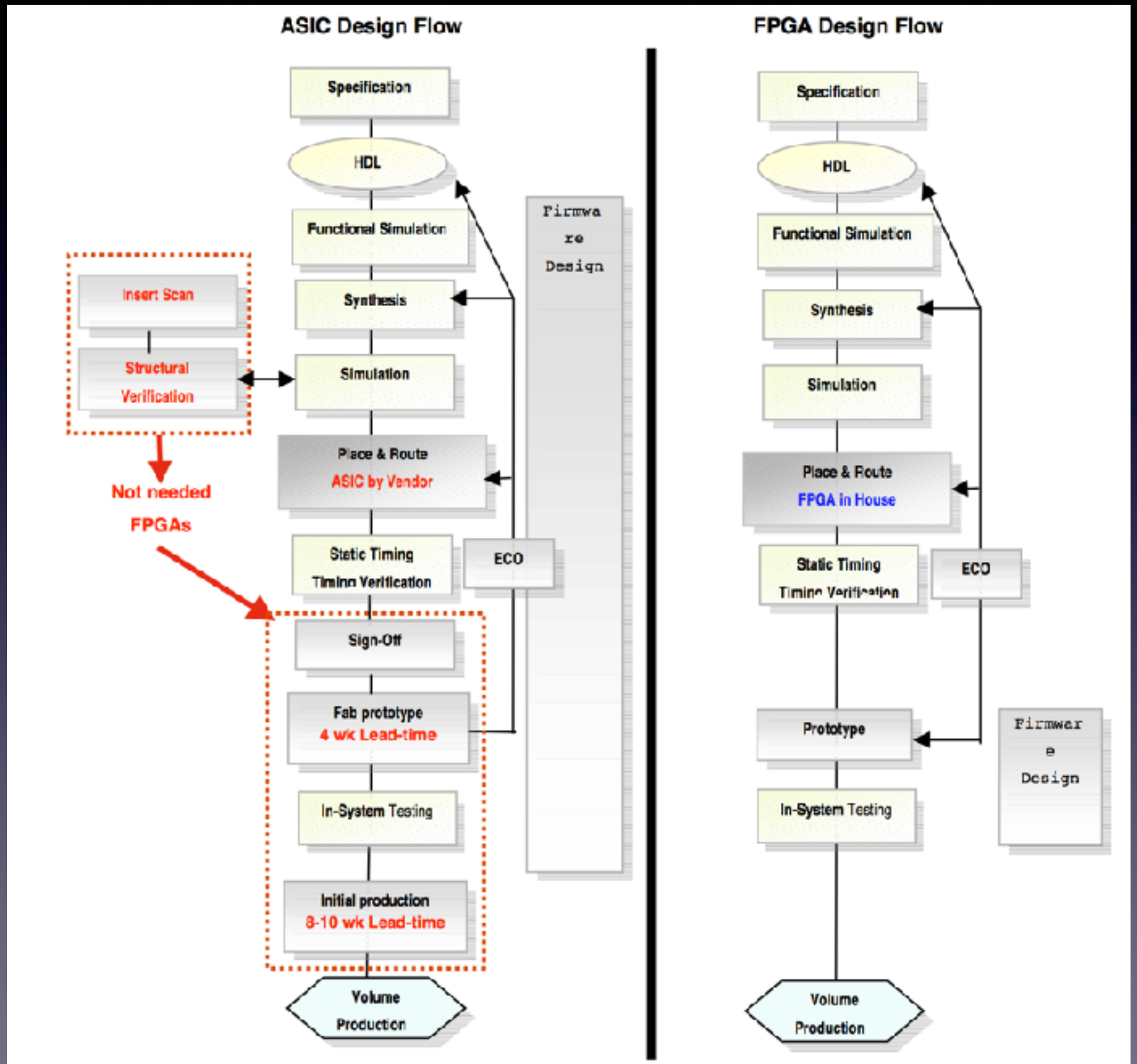
```
integer i;

initial begin
    reset = 1;
    clock = 0;
    din = 8'hAA;
    for (i = 0; i <= 4; i = i + 1)
      begin
      sel = i[3:0];
      @(posedge clock) ;
      end

end
```

Suggestions

- Remember that you can have as many procedural blocks in you test bench as you choose
- This means that you can code a block looking for a particular event if you need to confirm its occurrence

```
always @(posedge ready)
   $display("Saw the ready signal switch to 1");
```

ASIC/FPGA Flows



Debugging Rules
(c) David Agans www.debuggingrules.com

- Understand the system
- Make it fail
- Quit thinking and look
- Divide and conquer
- Change one thing at a time
- Keep an audit trail
- Check the plug
- Get a fresh view
- If you didn’t fix it, it ain’t fixed

Understand the System

- Read the manual - if there is one. This would include data sheets, anything that defines the behavior of the system you are debugging
- Make sure you don’t skip sections - read from cover to cover
- Know what is reasonable - know the system should work. This means you need to apply your understanding based on your background/training
- Know the “road map” - or the “data flow”. What information is flowing through the design and what algorithms are operating on that information
- Know your tools - for us that is the Xilinx ISE. Make sure you have a thorough understanding of the tools capabilities and how to use those capabilities to debug your design
- Look it up - don’t guess, don’t assume, make sure you confirm information you are relying on (FPGA pinout, etc.)

Make It Fail

- If your design does not function properly the identification of a failure is essential - this allows you to focus on the cause and confirm that you have fixed it

- When you are debugging work sequentially through the design by starting at the beginning and dealing with each anomaly one at a time

- Identifying the sequence of inputs that exposed the failure is a big step to resolving the failure

- Intermittent failures, works and then it doesn’t, are some of the most difficult to debug. Identifying the sequence of events causing the failure is essential to its fix

- Make sure you are collecting the proper information to allow you to debug the design

- Murphy’s law states that what should never happen - will. Be prepared.

Quit Thinking and Look

- This means that you should not just speculate what is causing the design to fail - you must understand the proper operation and then observe the failure in order to identify the cause of the failure

- Once you see an issue stay with it by observing as much information as necessary to identify the combination of events that lead to the failure

- There may be times when you need to include logic in your design to assist in the debug. A test mux is a common structure that once connected to a scope or a logic analyzer can provide tremendous insight

- Debugging in not a “clean” process - you must be willing to get your hands “dirty” in order to resolve issues

Divide and Conquer

- When debugging a design you need to narrow the search to one issue at a time. This is akin to “peeling an onion” - deal with one layer at a time
- Choosing your input patterns is very important - random data with random results is not intuitively obvious - choose stimulus that causes behavior that is easy to anticipate and therefore easier to debug
- As previously stated, as you uncover bugs stay with them until they are resolved - resist the urge to jump to the next when one becomes difficult

Change One Thing at a Time

- When introducing fixes to your design be careful to not change too much at a time - *Use a Rifle, not a Shotgun*
- This requires you to think through your proposed solution before you implement it - avoid *design thrashing* - i.e. what about this? what about that? …
- Always keep in mind there are two possible sources of a failure - the design under test and the test fixture. Keeping this straight is a major by-product of thoroughly understand the system being debugged
- If a failure crops up after once working do your best to keep track of changes made since it worked

Keep an Audit Trail

- Engineering is a difficult discipline - the amount of information/ data require to be keep track of may be overwhelming - help yourself by keeping a log book where you document what you are doing (both in debug as well as design)

- Write down what you did and in what order - then document what happened

- As previously stated this is difficult - keep in mind that *the devil is in the details*

- Do your best to correlate the data you have collected in order to form a conclusion

- *The shortest pencil is longer than the longest memory*

Check the Plug

- The essence of this point - learn to question your assumptions. When you are convinced that you have done everything correctly but it still fails - that is the time to consider what you missed

- Another common sense approach to debug is to begin at the beginning. All of our simulations begin at t=0 and the behavior should be correct from that point on. i.e. Verify reset works properly before moving into the operational debug

- A general rule of debug is "*trust your tools*" - this means that most of the time you have done something wrong. There remains a slim chance that the tool has an issue. Only travel down this path as a means of last resort.

Get a Fresh View

- Don’t be afraid to ask for help. It is remarkable the insight that can be gained when discussing your issues

- Speaking with others is helpful in that they will not enter the conversation with the same conclusions that you have already drawn - this may be a breath of fresh insight

- Don’t be afraid to ask an expert or one with more experience than you for help. They may or may not have the patience you need but seeking this avenue of assistance can be extremely insightful

- Do your best to practice “*egoless engineering*” - if all else fails read the manual

- Learn from your mistakes and let them adjust the way you do your design (process) so you don’t repeat it again in the future