Fossi-Foundation EDA Standards

libreChainEDA

Tool Smith Guide

Abstract

This manual is a guidebook for EDA tool designers to provide a firm understanding of how a modern design for reuse tool chain should function.

EDA Design Patterns.

Design Patterns are recurring problems in any field that are best handled by recognizing the problem and applying the optimal solution for every occurrence. This paper describes some patterns that we find in the EDA tools industry and provides some tactics to manage them.

There are four major design patterns that regularly occur with eda tools:

1) A Priori verses A Posteriori

2) Never copy data in a data base.

3) Never hard code constants in a program.

4) Hi volume Manufacturing.

A Priori verses A Posteriori

Most engineers will spend their entire career without ever encountering a problem that is so new that it has never been seen before now. Some of our problems are so old that their original names are in Latin.

A Priori refers to the case where you have complete knowledge of the situation beforehand. A Posteriori means that there are pieces to the puzzle that are missing that will not be known until some time in the future.

You can create an A Posteriori solution to an A Priori problem but under no circumstances should you try to solve an A Posteriori problem with an A Priori solution. It will work fine when you create it but it is a matter of when and not if it will fail.

The very first engineer to make that mistake ended his career passing out towels at the local vomitarium.

Never copy data in a database

Every piece of data is assigned to it's own storage location and anyone who needs that data will get it from that location. Any changes that are made to that data are made at that location.

This is simply common sense database management. If you let copies proliferate then there is an enormous problem with keeping everything in sync. Productivity suffers as different groups wind up making similar changes to different copies of a design. Fixes made on some copies are never passed to other groups that needed them.

No project ever turns into a total loss. Even if a project that you are on becomes a complete disaster , you can still salvage something by writing a book on what happened, why it happened and how it could have been prevented.

If you read any of those books then you probably already know this rule.

Never hard code constants in a program

No magic numbers. If your program need pi then create a variable and set its value to pi. That will make things a lot easier when the value of pi changes.

Everyone has heard this rule before yet for some unknown reason most verilog designers will routinely hard code things like module and file names in their code. In a large design for reuse environment those things will change a lot more ofter than pi will.

High Volume Manufacturing.

Prior to the industrial revolution, everything was hand crafted by artisans and craftsmen. Today we make tools that make tools that make tools that make things. Edison got it right. 1% inspiration and 99% perspiration. Engineers do the 1% and tools do the 99%. The EDA industry has grown into the area of Big Data. Any process that is not 100% automated will be buried under a tsunami of data.

is really BAD. All of these patterns are ignored or not understood and this results in huge productivity losses and mistakes when trying to design ICs.

We all wish that we worked in an A Priori world where things were stable and seldom change. But semiconductor design has never been stable. We work in an A Posteriori world but our tool smiths keep coming up with A Priori solutions that are guaranteed to break when we need them the most.

In 2009 Accellera released the IP-Xact standard (IEEE-1685). It contained a table of file types that listed every EDA file type known to man. It was great.

Five years later new languages like Chisel have been created and are now being used. What are designers supposed to do until the next revision of the standard? BTW: It takes about five years to revise a standard and another five before it is fully deployed through out the industry. We can't wait that long for solutions.

We copy data in this industry all the time. You need a uart for an IC that you are designing so you call a friend and ask him to send a copy of their latest uart. Of course you will pass it on to anyone who calls you and asks for it. But you will also add some new features and fix some bugs. Those changes will be passed downstream and should also get passed upstream but your friend has already released their chip so that doesn't happen.

IP modules in this industry go viral all the time. Copies are made, modified and recopied to all corners of the earth. That by itself is not a problem. The problem occurs when somebody is importing IP modules for a billion gate asic and they get the 4th great grand child twice removed of one of your modules when they already have it's ninth cousin in their design environment. That's when we see the train wreck.

There is no possible way that we can create robust designs in a timely manner when we have to draw our raw IP modules from a pond that is completely polluted with viral modules. The industry has to put an end to this practice.

The job of an R+D engineer is to create a single copy of the design and to make it perfect. The job of a manufacturing engineer is to make many copies of that design and to make each one adequate. R+D attracts engineers who want to be artisans and craftsmen. The problem is that art is a slow labor intensive process and our designs have grown to the point where you can no longer hand craft a design and even come close to meeting schedule. Our tool smiths need to adapt high volume manufacturing techniques to work on IC design tool flows. The problem is that most R+D engineers choose their careers because they did not want to spend their lives working on an assembly line and modern SOC design is starting to look more and more like an assembly line.

We know how to build widgets. Deming taught us how to do that. We need to apply his principles when designing our tool flows.

Tactics for managing Design Pattern Issues

Designate a single storage location for every piece or IP that you create that is backed up and under the control of Revision Control Software. In the corporate world this will be somewhere in the corporate cloud managed by the corporate IP Librarian. For individuals you should select a site like DropBox and keep archival physical backups in multiple separate locations.

Always distribute your IP by cloning your repository over the interNet. Never use sneakerNet! Cloning maintains a link back to the source that can to used to update when future changes occur. If access is not possible then create a mirror on some host that is accessible such as github.com. Maintain your mirrors! Bit rot is real and IP quality relies on everybody keeping all IP up to date.

When creating a new module start first by creating its RCS repository and clone the empty repo into your design environment and begin adding files. The RCS status command will then warn you that you have local files that are out of sync with the master.

Support multiple design environments. Engineers on larger projects will be working on several different designs at the same time. Ensure that nothing done in one environment can ever cross contaminate any others.

Assign a unique name to every piece of IP. Back before IC designs relied heavily on reusing modules you would assemble a design team who would create every block used in the chip. A meeting was usually held where every block was assigned a unique name and every module in that block had to start with that name. That technique prevented any module name collision in the chip. That was an A Priori technique that worked because every module designer knew what their name was before they created their module.

Once we started reusing modules then the problem changed to A Posteriori. The designers had to pick a name long before they would know what other modules were in the chip. Fortunately the engineering profession had the answer. Every engineering school on earth teaches that you should never hard code constants in your programs. No magic numbers! If your program needs Pi then create a variable and assign it to the value of Pi. That makes things so much easier when the value of Pi changes.

Engineers who followed this rule would create `define BASENAME module_name and put that inside of a `ifndef BASENAME and would use `BASENAME through out their module. If the name they chose didn't work in a chip then the chip design team could easily override it with their own `define and everything worked.

Engineers who didn't follow that rule and hard coded the module names(AKA: Hackers) are now sitting on large repositories of code that will be harder and harder to reuse as the industry continues to grow. They should get busy and fix their code before their boss realizes what has happened.

The only way to pick a module name that is guaranteed to be unique is by following the VLNV process set forth in IEEE-1685 (IP-Xact). That part they got somewhat right. Any other scheme only reduces the odds of a name collision. It won't completely solve the problem.

Run every component through a preprocessor TWICE before loading it into the System on Chip design. The 'define namespace is global and you can't risk having one components `defines affecting any other components configuration. Run all the files for each component through a preprocessor and save the output in a library file. All of the `defines will be resolved so the library file can then be loaded into the SOC without any problem. Do this twice for each component. Once with the `define SYNTHESIS set and once without. Each tool can grab whichever version that they need.

Support all variants for each component. It is all too common to see RTL code using `defines for configurations that does not support multiple instances using different configurations. Create a different module name for each different configuration and set it using `define BASENAME.

Do not use search paths on SOC designs. You can use a search path on a single component because every filename within a component is unique. There is no guarantee of unique filenames between unrelated components.

Be careful with parameters. They can resize a port but they cannot make a port appear or disappear. You can use a parameter with a generate statement to change what submodules are loaded or not loaded. While some engineers may think that may be a neato thing to do it will turn the problem of figuring out all your dependencies from a A Priori to an A Posteriori problem. You will not know what modules are needed until after you know all the parameters but you need to deliver that list before parameters are set.

Keep your synchronous and asynchronous resets completely separate. Mixing them creates a mess of DFT and timing problems. Maintain this separation all the way to the PC Board.

Put every flip flop on both asynchronous and synchronous reset systems. Any flipflop that is not completely reset will require functional verification to ensure that the design will work from any startup state. It will cost more money to do that verification than it would cost to simply reset the flipflop.

Keep track of any circuits that must function during the power-on reset assertion and perform a formal design review for all startup states. Some circuits like clock dividers must run during reset and they are designed to self start. Be careful, there be dragons here.

Manage all four different types of IP.

Hardware modules

Software modules

Tools

Toolflows.

They all need unique names with IP-Xact VLNVs. Would be nice if IP-Xact actually handled all of them and not just the hardware modules. If I am checking on a code run and I see some new windows have popped up on my display that are running my machines “map” game then I know some fpga software designer thought that they could get away with using that name for their map program. They failed.

Automate everything. Apt-get -y install toolname is a lot easier than having to download a gzipped tarball, unpacking it, resolving all dependencies , configuring and running make. If you can't script it then don't do it.

Use a package manager. We have always assumed that the engineer would locate all the needed repositories and download all needed code into the design environment before running any toolflows. That worked fine up till about ten years ago. Back then you usually only used a hand full of repositories for your entire chip and they were easily downloaded. Today we have hundreds if not thousands of models in each chip and they all come in their own separate repositories. Our EDA tools must now locate and download their own IPs.

Do not download any code into your design environment that is not used in the chip. There are some pretty serious bugs that are only detectable because you downloaded some code but forgot to instantiate it anywhere in the chip. Taking a one line error message that can save you from having to turn a mask and burying inside of 100 false positive messages is how you miss messages. Do not create a library repository with 100 components. Create 100 separate component repositories and only download the ones that you are using.

Use an index file like an IP-Xact catalog file to keep track of all modules that are available to the designers. This can be downloaded from a repo and updated to add modules for the entire team at one time. Only download a component when a designer selects the component for addition.