chiselWare

Developers Guide

Version 0.2.0

Warren Savage

November 23, 2025

Version	Date	Description
0.1.0	September 18, 2025	Working Draft
0.2.0	November 23, 2025	Update per new Dff template repo

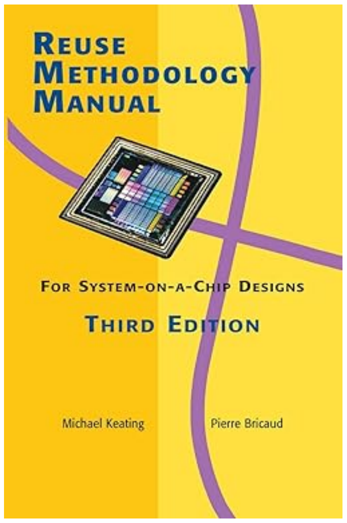
Contents

1 Introduction

Chip design is complex. Even the smallest designs involves thousands of decisions. In fact, one could argue that that chip design is mostly about making decisions. Having a methodology and guidelines dramatically reduces the number of decisions a designer must make and establishes a consistency and quality across all the work of many teams.

In the mid-1990s, Verilog was a relatively new language and there were no guidelines as to how to design semiconductor IP. As a result, the nascent IP market struggled in terms of quality which held back the widespread adoption of IP reuse for semiconductor design. Semiconductor companies were eager to develop their own IP portfolios for "internal reuse" yet lacked a methodology to achieve this.

In 1998, Synopsys and Mentor Graphics published the Reuse Methodology Manual (RMM) which largely came from Synopsys' internal strict guidelines for creating their DesignWare IP portfolio. The publication of the RMM was a seminal event that eventually led to the multi-billion dollar market for the IP that powers today's semiconductor industry.



The RMM didn't prescribe a Synopsys-way of doing things. Instead, it advocated for companies to establish their own methodology based on their needs and then enforce that discipline across their entire company. It provided many examples of things that should be standardized and how to standardize them, naming conventions for example. The naming convention itself was far less important than that there *was* a naming convention that everyone followed and that management would enforce.

The chiselWare Developers Guide (CWDG) is exactly this. An IP that claims to be chiselWare-compliant will follow the guidelines of the CWDG. That not only includes naming conventions but also guidelines on required deliverables, documentation, verification, and more. Users of chiselWare will be able to expect a certain standard of quality as they can see the rigor required to develop IP in this structured manner.

The CWDG is not static. It updated on a regular basis to accomodate improvements suggested by developers and users alike. Such updates however are not overly frequent to insure stability. There is a guiding principle to not mandate things that do not need it. The well-being of the user of chiselWare (User) is always at the forefront and that of the developer (Developer) is not far behind.

1.1 What's the big deal about Chisel?

The short answer is that Verilog a terrible language to describe hardware.

In fact, Verilog was never designed to be a language of design. It was invented in the 1980's as a simulation modeling language and quickly caught on. The primacy of Verilog arose in 1988 when Synopsys developed logic synthesis and needed a language to ingest for the synthesis of RTL to gates. Verilog was pervasive enough that the smart engineers at Synopsys realized that they could apply a synthesis semantic to Verilog and thus allowed for a single language to be used synthesis and simulation. If you were a digital designer in 1992, it was brilliant. If you are designer in 2025, you wonder "What were they thinking?!"

Since that time, very few advances have been made in the way we *describe* hardware. Software, on the other hand has made considerabled advances since that era with new languages like Haskel and Rust, containerization, and CI/CD methodogies. Hardware design remains mired in the legacy and limitations of the Verilog.[1]

[1]For the remainder of this document we will use simply refer to both SystemVerilog and Verilog as "Verilog" for brevity.

VHDL and SystemVerilog are hardly better and also suffer from not being designed for describing hardware. VHDL was originally a documentation language for describing systems and later adapted to include synthesis semantics following the Verilog precedent. SystemVerilog's primary impedus was to support advanced verification techniques such as Univerval Verification methodology or UVM.

The deficiencies in Verilog is evident by the existence of an entire EDA segment focused on addressing the weaknesses of Verilog, e.g. linting tools such as Synopsys' Spyglass. That there is a business motivation to perpetuate Verilog assures that if change is to come, it has to come from outside the commercial arena.

In 2012, the University of California at Berkeley released Chisel, an expressive language, that *generates* Verilog.[2] Chisel inherits many features from Scala, a powerful new *software* language, that provides the designer with procedural, object-oriented, and functional programming paradigms. These capabilities in the hand of a hardware engineer provides a modern language capable of efficiently describing modern hardware designs. There are numerous on-line resources for learning Chisel[3].

1.2 What's the big idea behind chiselWare?

The RMM prescribed a set of rules and conventions for creating a reuse culture within a corporation. The chiselWare Developers Guide provides such conventions, not for a corporation, but for a *community.* Fortunately, the Chisel language itself obviates the need for many of the rules that targeted the dangers stemming from the semantic weaknesses of the Verilog language. So, in general, there are a lot less things to go wrong in Chisel.

The chiselWare Developers Guide provides a methodogy and framework to develop professional-grade IP using an open-source EDA tool chain. IP cores that have been developed in compliance with it will qualify those cores to be part of a curated library on GitHub and will entitle the authors to use the chiselWare logo as part of those cores' documentation.

There are a few overarching tenets associated with chiselWare, roughly in this order:

- Be kind to the user of the IP
- Be kind to the developer of the IP
- Allow automation whereever possible to ensure consistency and quality

1.2.1 Be Kind to the User

Commercial IP users are not hobbyists. They have no interest in reading your code. They are interested *in using your code*. Most commercial license agreement prohibiting copying IP code and some vendors go to great length to prevent users from even seeing the source code.

As with other high quality commercial IP, the user should not have to "read the code" in order to understand what the IP does.

So as part of that kindness, we provide customers with a consistent view of what they get when they use a chiselWare core. These include:

1. Consistent coding style
2. Consistent set of deliverales and in a consistent place
3. Consistent and thorough verification
4. Consistent, complete, and thorough documentation

[2]Chisel can generate either Verilog or SystemVerilog.
[3]See appendix for a list of some of these.

Coding Conventions If you have ever looked at a lot of high quality open source code you will see immediately that despite a large number of contributors the code looks extremely consistent, as if it was written by a single person.

chiselWare follows a coding style adopted from the Scala and Chisel coding style guides.

Directory Structures There is nothing more annoying about using another person's code than understanding how its all put together. Fortunately, Scala and Chisel are very structured in the way code is written and that discipline is followed whereever possible.

However, non-Scala/Chisel deliverables (i.e. documentation, synthesis scripts, etc.) are not defined. Therefore, all chiselWare cores follow a standardized list of required deliverables and where those deliverables are located.

Users of Core A will find the deliverables from Core B in exactly the same places.

Verification Completeness It is quite true that "verification is never complete", and that is true for chiselWare as well. However, its even more true that verification is often incomplete or inadequate. How is the user supposed to know?

chiselWare endeavors to prescribe a minimum level of verification that users can be assured that the cores have completed.

Documentation Completeness Developers often chafe at writing good documentation as it's, frankly, not nearly as fun as coding. However, it is as important deliverable as everything else, perhaps even the most important.

As such, chiselWare cores have minimum documentation requirements to insure that the user has the information required to use the core without needing to reverse-engineer the code by looking through the source code.

1.2.2 Be Kind to the Developer

Second only to being kind to Users is being kind to Developers.

If developing chiselWare cores is so onerus that no developer are willing to jump through the hoops to be chiselWare-compliant, then there will be no cores and no users.

So the administrators of chiselWare will try to abide by some tenets to make life as easy as possible for developers while keeping in mind our committment to users.

Will some people disagree? (Yes!) Is it subjective? (Yes!) But we expect that reasonable people can disagree and see the overall value despite what some may see as "warts."

Minimize the Rules Rules should not be arbitrary, there should be a clear reason for a rule that is related to producing quality cores that are easy to user. Therefore all rules should have a well-reasoned justification for its existence.

If a rule exists and there is an outcry from the developer community as it onerus or even stupid, then we should consider modifying it or even removing it entirely.

On the other hand, if we see problems arise from the user community about the same problem being seen again and again, then we have an obligation as a developer community to address it– though some may grit their teeth.

Don't Reinvent the Wheel If other people have put thought into certain conventions, then we should follow that convention. For example, both Scala and Chisel have excellent styleguides– so let's use that.

Sometimes these things may be ambiguous. In that case, let's take a stand and thoughtfully clarify them.

1.2.3 Consistency through Automation

There used to be a saying in the days following the publication of the RMM:

A rule without a tool is not a rule.

We cannot agree more! It is completely unreasonable in the days of extensive CI/CD tools that developers should have to memorize the rules contained in this guide and then manually check their code against those rules.

So we will live in the modern world and dictate that 100% of the rules contained in this developers guide have automated checkers to help the developer find mistakes quickly.

In this way, we can enforce consistency across all chiselWare-compliant cores. Over time, this way of designing will become second nature to chiselWare developers.

2 Directory Structure

Chisel and Scala dictate a standard directory structure that enables the build tools to generate Verilog. In general, chiselWare will use all Scala and Chisel standards where possible and extend that to recognize a standard set of IP deliverables that are consistent with commercial practice.

Chisel and Scala offer some flexibility in regard to directory structure that are convenient for small projects but run into problems in regard to proper package management for more complex projects. Therefore chiselWare adopts a more resilient directory structure that can be used without trouble for all size projects.

Refering to the directory structure diagram on the next page, you will see the top the *modules* directory which will contain the top level core directory (e.g. *mycore*) for your project. Most projects will include only one project, but some complex projects may want to have multiple cores with independent environments and deliverables for the purpose of modularity. For example, a CPU with an optional floating point unit.

```
\modules\cpu
        \fpu
```

Underneath the *mycore* directory are the following directories which hold most of code for the core.

1. *docs* contains the user documentation
2. *sw* is an optional directory which may hold software drivers and other such deliverables associated with the core
3. *src* contains the source code for the code

Underneath the *src* directory are the familiar *main* and *test* subdirectories. It is here where the a deep hierarchy is required to ensure packages are properly handled. This at first, seems complicated and requires some explanation.

All chiselWare package names begin in the format:

```
org.chiselware.cores.o<org name>.t<team name>.<core name>
```

For example:

```
org.chiselware.cores.o1j.t05h.mycore
```

Each core is associated with a *organization* which is a two-digit alphanumeric code in the format of the Bash glob pattern [a-zA-Z0-9]. This provides nearly 4000 possible developer organizations.

Each core is also associated with a *team* which is a three-digit alphanumeric code in the same Bash glob pattern. This provides nearly 240 thousand possible teams per organization.

Organizations and teams are assigned automatically when a new core is registered via the chiseware.org web site.

At the bottom of the *main* and *test* hierarchy is where the real code for the core resides. Again, an example is helpful, see the tree on the next page.

A template project `https://github.com/chiselWare/00-000-dff` is provided to allow users to get started with a runnable project that can be modified with the details of their own core. This greatly simplifies getting chiselWare-compliant infrastructure in place so developers can focus on the design of their core. See Section 9 for details on how to modify the template for your own design.

```
./ Project root directory
├── build.sbt
├── project
│   ├── plugins.sbt
│   └── built.properties
└── modules
    ├── mycore
    │   ├── src
    │   │   ├── main
    │   │   │   ├── resources
    │   │   │   │   ├── vsrc
    │   │   │   │   └── stdcell.lib
    │   │   │   └── scala/org/chiselware/cores/o00/t000/
    │   │   │       └── mycore
    │   │   │           ├── MyCoreParams.scala
    │   │   │           └── MyCore.scala
    │   │   └── test
    │   │       └── scala/org/chiselware/cores/o00/t000/
    │   │           └── mycore
    │   │               └── MyCoreTest.scala
    │   ├── sw (optional)
    │   └── docs
    │       └── user-guide      User documentation
    │           ├── draw.io     Draw.io files
    │           ├── wavedrom    Wavedrom files
    │           ├── images      Image files used in document
    │           ├── pdf         Generated PDF files
    │           ├── html        Generated HTML files
    │           └── MyCore.tex  Top level LateX file
    ├── LICENSE.md
    ├── README.md
    └── Makefile    Makefile for running regressions
```

3 Versioning

chiselWare uses Semantic versioning (SemVer), for representing the version of each chiselWare core.

The SemVer for chiselWare is defined as follows the familiar x.y.z sequence:

The first number (x) represents the major version of the core. This is incremented when a new feature or capability is added to the core.

The second number (y) represents the minor version of the core. This is incremented when there there is any functional change to the core (RTL) that results in different gates being generated during synthesis.

The third number (z) represents any non-functional change to the core (RTL). Examples of this would be updated documentation, additional test cases, etc.

The purpose of this schema is to inform users of the core that the nature of the change to allow them to make the decision whether to take the update, or stay with the current version they are using.

Here are some examples:

0.9.4 to 1.0.0 (under development to first release)

1.1.1 to 1.1.2 (fixed typo in README.md)

1.5.4 to 1.6.0 (fixed a bug)

1.7.10 to 2.0.0 (added support for high-speed mode)

Notice how each time y is updated, the z position resets to 0. When x is updated, the y and z positions are reset to 0.

4 Parameters

Nearly every IP core has parameters to maximize its reusability for its users. Chisel and Scala offer many powerful ways to use parameters. chiselWare preserves this flexibility except at the top level, where we use case classes to regularize the use of parameters at this level to ensure that every core is instantiated in the same way.

An example of a top-level case class is shown below. The class name is of the form: ⟨ module class⟩ Params.

```
case class DynamicFifoParams(
    dataWidth: Int = 8,
    fifoDepth: Int = 8,
    externalRam: Boolean = false,
    coverage: Boolean = false,
    bbFiles: List[String] = List("your_sram.v")
) {

  require(dataWidth >= 4, "Width must be greater than or equal 4")
  require((fifoDepth % 2) == 0, "Depth must be a power of 2")
}
```

All parameters that are used in Verilog generation or in the test bench are reflected here.

In addition, all parameters that have bounds should use a require statement to insure that the user doesn't misconfigure their instantiation. Use informative error messages to help the user correct their mistake.

The instantiation of a core using case class parameterization is shown below.

```
val myParams = DynamicFifoParams(
  dataWidth,
  fifoDepth,
  useExternalRam,
  bbFiles = List("dual_port_sync_sram.v"),
  coverage = true
)
val myFifo = new DynamicFifo(myParams)
```

In this example, we are customizing the default parameter settings of *bbFiles* and *coverage* with our desired parameter values. We then pass those parameters to our instantiation.

5 Coding Style

chiselWare required adherence to a reasonably strict set of coding guidelines to insure consistency across chiselWare cores. These are described in the following sections in order of precedence.

5.1 Scala Coding Style

All chiselWare cores shall be coded to be in compliance with the Scala Style Guide.

`http://docs.scala-lang.org/style/`

5.2 Chisel Coding Style

All chiselWare cores shall be coded to be in compliance with the Chisel Developer's Style Guide

`https://www.chisel-lang.org/docs/developers/style/`

5.3 chiselWare Coding Style

The following coding styles are required for compliance.

5.3.1 Coding Style Follows That of the Language

Scala and Chisel code is written according to those respective styleguides.

Other languages that appear in chiselWare cores follow the best practices of those languages, for example:

1. **Tcl** follows the Tcl styleguide.

   `https://wiki.tcl-lang.org/page/Tcl+Style+Guide`

2. **Bash** shall be the required shell and follows the Bash styleguide.

   `https://google.github.io/styleguide/shellguide.html`

3. **MakeFile** follows the Make styleguide.

   `https://www.gnu.org/prep/standards/html_node/Makefile-Conventions.html`

6 Verification

It goes without saying that all chiselWare cores must be verified to ensure correctness. The verification process is a critical part of the development cycle and should be approached with the same rigor as the design itself. The following sections outline the verification strategies that should be employed for chiselWare cores.

6.1 Directed Tests

Directed tests are essential for verifying the functionality of chiselWare cores. These tests should cover all the critical paths and functionalities of the core, ensuring that each component behaves as expected under various conditions. Directed tests should be written in a way that they can be easily understood and maintained.

They should also be comprehensive, covering both normal operation and error cases.

At a minimum, directed tests should include tests at the top-level module interface. Depending on the complexity of the core, unit-level tests may be desired, especially for testing of cornercases that are much harder to reach at the top-level.

Each directed test should be clearly documented in the comments, specifying the expected behavior and the conditions under which the test should pass or fail. This documentation is crucial for future maintainers and for understanding the intent of the test.

For configurable cores (i.e. those with parameters), directed tests should also include tests for each configuration option to ensure that the core behaves correctly under all supported configurations. A minimum of three configurations should be tested: the default configuration, a configuration with all parameters set to their maximum values, and a configuration with all parameters set to their minimum values. Additional configurations may be useful depending on the core's complexity and the number of parameters, at the discretion of the developer.

6.2 Constrained-Random Tests

Constrained-random tests are a powerful tool for verifying the functionality of chiselWare cores. These tests use random inputs within specified constraints to explore the design space and uncover potential issues that directed tests may not catch.

The "state-of-the-art" in modern verification involves the UVM (Universal Verification Methodology) and SystemVerilog. However, UVM is not currently supported in Chisel, and it also imposes an unnecessary complexity for most chiselWare cores. The goal of chiselWare is to keep the verification process simple and efficient, focusing on the core's functionality rather than the intricacies of a complex verification framework.

Instead, chiselWare cores can be verified using Scala-based testbenches that leverage the Chisel testing framework. This allows for the creation of constrained-random tests that can effectively explore the design space while still being manageable and maintainable.

Unfortunately, ChiselTest is currently deprecated due to a lack of a developer to carry it forward. Its successor, svsim, lacks forking support, which is a significant limitation for constrained-random testing. Due to this, chiselWare cores will use the last working version of ChiselTest for constrained-random testing until a suitable replacement is available.

In addition to the directed tests, each chiselWare core should include a set of constrained-random tests that cover a wide range of configurations using ChiselTest. These tests should be designed to exercise the core's functionality in various ways, including edge cases and unexpected inputs.

Developers at their descretion, however, can supply a UVM testbench for their cores if they wish to do so to satisfy the chiselWare Constrained-Random requirements instead of ChiselTest. The requirement for ChiselTest for Directed Tests is still required.

6.3 Formal Tests

Formal verification is a powerful technique for ensuring the correctness of chiselWare cores. The newer versions of Chisel support formal verification through the use of the Chisel Formal library. This library allows developers to write formal properties that can be checked against the design, providing a high level of assurance that the core behaves correctly under all conditions.

Unfortunately, there is a conflict between the Chisel Formal library and the ChiselTest library, which prevents them from being used together in the same project. As a result, formal verification is not currently supported in chiselWare cores.

6.4 Scala Code Coverage

Scala code coverage is an important aspect of verifying the correctness of chiselWare cores. It provides insights into which parts of the code have been executed during testing, helping to identify untested areas and potential bugs. To ensure that all chiselWare cores are adequately tested, developers should use a Scala code coverage tool, such as Scoverage, to measure the code coverage of their tests. The goal is to achieve a minimum of 90% line coverage and 95% branch coverage for all chiselWare cores. This level of coverage helps to ensure that the core is thoroughly tested and that any potential issues are identified and addressed.

7 Documentation

Documentation is vital to any quality product. It is the key to ensuring that users and developers can understand and effectively use chiselWare cores. The following sections outline the documentation requirements for chiselWare cores.

Good documentation is evidence (to the user) that the developer has a good understanding of the core and its intended use. Together with the code, it is the primary means of communication between the developer and the user. Therefore, it is essential that all chiselWare cores are well-documented.

7.1 Code Comments

chiselWare should follow the Scaladoc documentation style for Scala code, which is a widely accepted standard in the Scala community. This includes using Scaladoc comments to document classes, methods, and parameters. Scaladoc comments should be clear, concise, and provide enough information for users to understand the purpose and usage of the code.

All public classes, methods, and parameters should be documented with Scaladoc comments. These comments should include a brief description of the purpose of the class or method, any parameters it takes, and the return value if applicable. Additionally, any exceptions that may be thrown should also be documented.

The developer should take care that future maintainers can easily understand the code and its intended functionality. The preferred style for conveyance of this knowledge is through a narrative preamble to the code, rather than through comments that simply restate what the code does line by line. This narrative should provide context for the code, explaining why certain design decisions were made and how the code fits into the overall architecture of the core. For example:

```
/** There is currently a limitation in Chisel with regard to performing
* individual bit manipulation on a Vec of UInts. This is due to the way
* that Chisel generates hardware for Vecs, which does not allow for
* individual bit manipulation on the elements of the Vec.
*
* The work around is to convert the Vec of Bools, perform the bit manipulation
* in Bool and then convert it back to a Vec of UInts.
*/

val siBool: Vec[Bool] = Wire(Vec(numScanChains, Bool()))

siBool(chainNum) := mySlices(chainNum).io.scanIn
dut.io.si := siBool.asUInt
```

Avoid inline comments that simply restate what the code does, as these can clutter the code and make it harder to read. For example:

```
def flushBuffer(): Unit = {
buffer.clear() // Clear the buffer to prevent overflow
}
```

Always use clear and descriptive variable names that convey the purpose of the variable to insure a future maintainer has the same understanding as the original developer. Don't be afraid to use longer variable names if they improve clarity. It is better use a longer, descriptive name than inventing an acronym that is not widely understood.

However there will be cases where a comment is necessary to explain a complex piece of code or a specific design decision on a line by line basis.

7.2 User Guide

A User Guide is an essential part of the documentation for chiselWare cores. It provides users with the information they need to effectively use the core, including how to integrate it into their designs, how to configure it, and how to use its features.

No one should be expected to read the source code to understand how to use a chiselWare core.

The User Guide is written in LaTeX according to a template that is provided in the chiselWare repository. LaTeX provides a highly portable and professional-looking format for documentation, making it easy to create both PDF and HTML versions of the User Guide. Furthermore, it is widely used in the academic and technical communities, making it a familiar format for many developers and one that is not dependent on any commercial word processing software.

The minimum requirements for a chiselWare User Guide are as follows:

- Version History
- Errata and Known Issues
- Port Descriptions
- Parameter Descriptions
- Simulation
- Synthesis

8 Regressions

Regressions are a critical aspect of maintaining the quality of chiselWare. Once a core is officially released, it will be regularly regressed as part of a CI/CD infrastructure.

As chiselWare evolves, the requirements for a core to be part of chiselWare may change.[4] However, there is no obligation for a core to keep up with the latest version of chiselWare in order to remain part of chiselWare. A core that was successfully released under a prior version of chiselWare will always be part of chiselWare, even if it does not meet the requirements of the latest version of chiselWare.

However, with each new version of chiselWare, the regression tests will be run on older cores and issues will be reported to the core developer. If a core fails to meet the requirements of the latest version of chiselWare, it will be marked as such in the core registry.

It is up to the core developer to decide whether to update the core to meet the latest requirements. If the core is not updated, it will still be available to users, but it will be marked as being compatible with an older version of chiselWare. This allows users to make informed decisions about which version of a core to use based on their needs and the requirements of their projects.

Keeping your chiselWare core current is optional, but highly encouraged.

8.1 Configurations

Each core will be regressed using the three configurations described in the *Verification* section of this document.

Additonal configurations may be added at the discretion of the core developer.

8.2 Simulations

Each core will be regressed using the version of Chisel and ChiselTest defined by chiselWare.

8.3 Code Coverage

Each core will be regressed using the version of scoverage defined by chiselWare meet the minimum code coverage defined by chiselWare.

8.4 Synthesis

Each core will be regressed using the version of the synthesis tool defined by chiselWare to meet the minimum synthesis requirements defined by chiselWare.

At least three configurations will be synthesized:

- the default configuration
- a configuration with all parameters set to their maximum values
- a configuration with all parameters set to their minimum values.

Additional configurations may be synthesized at the discretion of the core developer.

8.5 Documentation

Each core will be regressed to ensure that it meets the minimum documentation requirements defined by chiselWare. This includes the User Guide and the code documentation written in Scaladoc style.

[4]Great care is taken to not make chiselWare a Tower of Babel, suffering from an ever escalating number of requirements.

8.6 Regression Harness

Make is used in conjunction with GitHub Actions to form a regression harness.

The regression flow follows this basic sequence:

- Generate all the documentation
- Run simulations on all configurations with code coverage enabled
- Run synthesis and perform static timing analysis
- Check logs for any errors

For the most part, execution is automatic. The developer only has to make some customizations in the Dff template. The required customizations are described in the *DEVELOPERS.md* file in the Dff template repo.

9 Getting Started

To insure quality and consistency, chiselWare cores are fully productized to commercial-grade standards.

In order to make that easy for developers, a fully-compliant chiselWare core is provided as a template as the starting point for new core projects.

A template repository is provided to chiselWare developers that is a very simple design (a d flip-flop) that is fully chiselWare-compliant core.

`https://github.com/chiselWare/00-000-dff`

This code is automatically populated in new chiselWare core repos as part of the core registration process. The following are the main components of the core.

1. **Scala setup** A standard *build.sbt* and other setup files are provided to allow the core developer to work in the same environment as other chiselWare developers.
2. **Directory Structure** The standard directory structure is provided to allow the develop an out-of-the-box correct example of how to structure their code.
3. **Documentation Structure** The standard documentation structure is provided with all the required sections to be populated. This includes the README and LICENSE markdown files.
4. **Regression Harness** A standard (across all chiselWare cores) is provided to allow the chiselWare CI/CD infrastructure to automatically run the full complement of tests as part of the release environment. A Makefile is used to hold the environment together.
5. **Synthesis Framework** A structured environment is provided to automatically generated synthesis tests and static timing analysis on the core.
6. **Testing Framework** Like synthesis, simulation tests are automatically run and a framework is provided to allow multiple configurations of the core to be simulated.

Within the Dff repo, there is a *DEVELOPERS.md* file that contains instructions on how to modify the repo for your own core.