Building Design Process in a Startup Company

David A. Gates

ATI Research Silicon Valley, Inc., Santa Clara, CA, USA
gates@ati.com

Abstract

A startup company must build its design process up from nothing into a complete system in a very short period of time. However, a startup design team is confronted with several other challenges as well. The area of functional verification is one where complete design automation is essential to first-time design success. Built from scratch, the verification environment is gradually brought under fully automated control. DV, a verification flow manager, is the product of one such progression. Its software requirements and how it meets them are described.

1 Introduction

A startup ASIC design company has one big disadvantage and one big advantage when it comes to design process: the absence of one. Lacking a design process, a startup must build or acquire all the necessary pieces to complete a design. This can be an expensive and time-consuming proposition. On the other hand, the new company is not saddled with the compromises and tradeoffs made as a design process evolves to become the standard design environment of an established company. The startup starts with a clean slate, limited only by the experience and creativity of its early employees. The challenge during the startup phase is to build a design process that allows the company to get its first product out the door on time, while at the same time providing a solid basis for future design cycles.

In this paper, we describe the challenges facing a startup company that affect the way it goes about completing an ASIC design. These challenges place constraints on the way the design process evolves. In particular, focus is placed on the functional verification process, an area where comprehensive design automation is essential to project success. Evolution of the design process through various stages shows how an initially chaotic situation is gradually brought under the control of increasing automation. The end result of this evolution is a verification flow manager [1] that meets the requirements of a startup company. The architecture, implementation and use of such a tool called DV (for Design Verification) are described. Details and examples of how DV meets its requirements are presented.

2 Startup Challenges

Few, if any, startups choose to attack an entrenched competitor using commonly available means. Instead, a startup's goal is generally to introduce a new product or technology before any of its competitors. Such competitors can take the form of both other startups with similar goals and well-established companies working to improve their existing products.

2.1 Fluid Goals

A startup relies on its small size and lack of bureaucracy to adapt quickly to changing market conditions. It can also change goals quickly when faced with an insurmountable technical problem.

The design process must be able to tolerate large swings in the technical direction of the venture. For example, the team may be initially committed to a new technology node, only to realize that using the current node is the only way to hit a market window. A design process that takes weeks or months to reengineer could cause the startup to lose its competitive advantage.

2.2 Time Pressure

Quick reaction time is a response to the extreme time pressure startups face. If a competitor beats the startup to market, the most likely outcome is that the startup will cease to exist. Employees work long hours to compensate for the limited capital and personnel resources of the startup.

Time pressure impedes the development of a consistent and repeatable process. When a tradeoff is required between development time and process quality, time will generally win out. It is much easier for a weary engineer to focus on solving the problem at hand than to think about generalizing the solution. However, a design environment can mitigate this problem by capturing the design process as it develops. This captured form can then be reviewed and updated to establish standard practices.

2.3 Diverse Workforce

The engineering team has no doubt been recently assembled by hiring experienced individuals from different backgrounds. Each engineer brings their own training and experience to bear on the design process. Each will have strengths in some areas and weaknesses in

others. By drawing on the experiences of all engineers, the design process of the new organization can be built on the best practices of the group.

In some cases, two or more equally valid methods will be available. It may make sense to allow both to coexist, to choose one over the other outright, or to give each a trial to determine the preferred method. For example, in the first case, engineers working on different blocks of a design can use techniques most familiar to them to complete their designs. Forcing a common design process on everyone could introduce inefficiency that would lengthen overall development time. Instead, an extensible design environment is needed to accommodate process diversity.

2.4 The Leading Edge

Often the team will be facing technological hurdles on the leading edge of ASIC design. Innovative products have specifications that require the latest process node, a larger, faster ASIC, exotic process technology or all of the above.

In light of this, the design process needs to be able to adapt to handle these challenges. New tools and flows will be needed that no member of the team has dealt with before. It may be necessary to experiment with different solutions before an acceptable one is found. In addition, engineering teams working in parallel may end up solving the same problem in different ways. Only a flexible design environment will be able to cope with these issues.

3 Functional Verification

One area of ASIC design where complete design automation is essential is functional verification. Full verification of a complex, multi-million gate ASIC requires thousands of directed and random tests to be run. The majority of these tests are aimed at verifying the RTL representation of an ASIC design.

3.1 Example Verification Flow

Figure 1 shows a simple flow for testing an RTL design. It is used for examples in later sections. Common infrastructure is compiled first for use by later steps. The RTL source is compiled into a simulator. In parallel, a test stimulus generator is compiled and run to generate input vectors for simulation. During simulation, the input vectors are fed in and output results are captured. These results are then checked for correctness against 'golden' results. The golden results are assumed to be correct and can be obtained in a variety of ways.

3.2 Automation of Verification

Verification is performed at both the individual block level and at the chip level where all blocks have been integrated together. The exact details of how to run tests at each of these levels are different and should be hidden

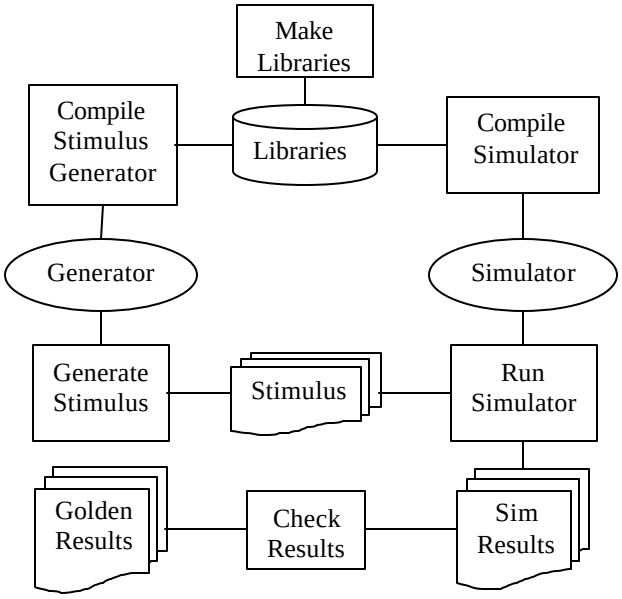

Figure 1: Example Flow

from the designer as much as possible. This makes it easier for the engineer to focus on design and debug of a block, yet still be able to test the block in context.

Abstracting the test process also helps when running test suite regressions. It is desirable to have a common environment that can run one test at a time interactively or entire test suites in the background. Batch queuing systems such as LSF [2] make the latter task feasible.

4 Design Process Evolution

When developing a new design environment, the design process evolves through several stages on a path towards comprehensive automation. In the following it is assumed that the team is already following certain common practices for good design: use of detailed specifications, well-documented code, source-code control with individual sandboxes for each engineer, and others [3].

4.1 Piecemeal Automation

Initially, individual pieces of the verification environment are developed in isolation. Engineers use commonly available tools such as Make, Perl, Python and TCL to automate building of the various components. The choice of which to use for each task is left up to the engineers, who make their decisions based on familiarity with a given tool or a desire to learn a new one. The net result is that the design environment is covered by a group of isolated areas of automation.

4.2 Manual Flows

In the next phase, engineers develop ad hoc flows to build multiple sections of the sandbox. For example, the following list might be used to run a simple flow:

```
# Make libraries.
1 > cd $TOP/lib && make
# Compile stimulus generator.
2 > cd $TOP/test/src && make gen
# Compile simulator.
3 > bld_sim.pl $TOP/sim/c1.conf
# Generate stimulus.
4 > cd $TOP/test && src/gen t1
# Run simulator.
5 > run_sim.pl –test t1 –conf c1
# Check results.
6 > chk_sim.csh –test t1 –conf c1
```

Note that this flow is a typical mix of different automation tools used at different steps. Initially, the flow is not fully automated because the complete list of steps needed is being discovered by trial and error. Recall that it is likely that no one person understands the complete system, since the pieces have been developed by several people working independently.

As an engineer makes changes to the sandbox such as modifying RTL source, fixing a library bug, or extending a test case, they must rerun parts of the flow to bring the sandbox up-to-date. In many cases, it may be so difficult to determine which pieces are out-of-date, that an engineer will choose to rerun all of them. The alternative is to risk working with an inconsistent sandbox which contains obscure, difficult-to-trace bugs.

One advantage manual flows have is being easy to debug, since the steps are run one at a time and the output of each step is immediately available for inspection.

4.3 Hard-coded Scripts

In short order, typical engineers become frustrated with the tedium of manually running through such flows over and over. In response, a hard-coded wrapper script is written that runs through the entire flow.

If the flow is simple enough, a target in a Makefile may suffice:

```
runtest1:
    # Make libraries.
    cd $TOP/lib && make
    # Compile stimulus generator.
    cd $TOP/test/test1 && make test1
    ...
```

However, in most situations, the dependency / update structure of Make interferes with efficient implementation, and a scripting language is used instead.

Running multiple steps automatically one after another can make it difficult to determine which output belongs to which step of the flow. For example, while errors will generally stop flow execution, mere warnings will only show up as messages embedded in the output stream. Working around this problem requires extra time and effort that might be skipped when developing many hard-coded scripts.

4.4 Generalized Scripts

Hard-coded scripts have the disadvantage of being very inflexible. Although they can be parameterized to run different instances of the same kind of test, a new script must be written for each major variation of a flow. Users must either learn the details for running each kind of test, or a wrapper must be written that calls the appropriate script as a subroutine.

What is desirable is a generalized form of scripting that combines the labor-saving of hard-coded scripts, the observability of manual execution and the flexibility to integrate new flows that neither provides. In the next section, we describe a tool called DV that has these three properties and more.

5 DV Architecture

The requirements for a functional verification tool meeting the special needs of a startup design team are summarized as follows:

- Flexible: can accommodate multiple flows and levels of design detail.
- Extensible: users can define new flows applicable to a particular part of the design.
- Formatted: test cases must be stored in a way that users can maintain.
- Automated: can run one or many tests interactively or as a batch.
- Observable: can easily follow the flow of control to spot problems.
- Modifiable: can be updated while in use as a project progresses.

DV (short for Design Verification) is a tool developed to meet these requirements. The architecture of DV is shown in Figure 2.

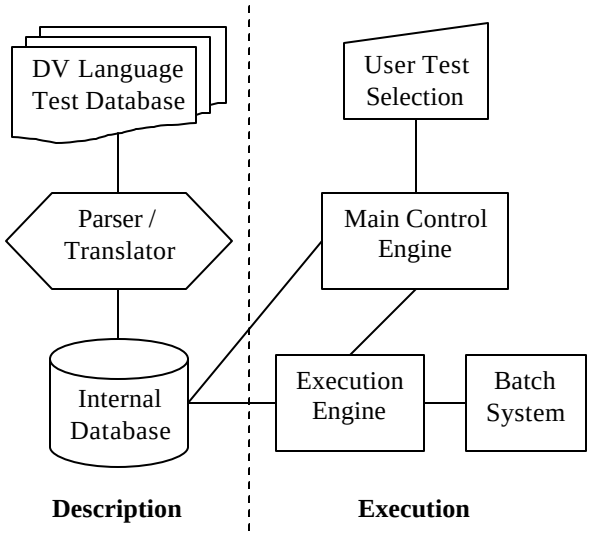

Figure 2: DV Architecture

The architecture is divided into two major pieces: one for describing tests and one for executing tests. Integral to the architecture is a database language (the DV language) that captures the testing process in an automatable way. This language is designed to make it easy for users to describe hundreds of tests for each block of a design and to account for the individual personalities of the blocks. A parser / translator converts the database for each block into a binary format that can be used during execution. Manual input from the user selects which tests to run and how to run them. The main control engine then extracts test information from the internal database and passes tests to the execution engine. The execution engine then interacts directly with the host system to run the flow or with an external batch system that manages resources throughout a network. The execution engine also interacts with the internal database to obtain parameters for the individual steps of a flow.

5.1 DV Language Descriptions

The DV language can describe four different kinds of things in the test database. They are configurations, tools, flows and tests.

Configurations describe the structure of the design under test. They define which source to include for blocks and for special stimulus and monitoring code. The configuration description for the example might look like this:

```
// Describe structure.
conf c1
  block1 = rtl
  block1_mon = live
endconf
```

Each configuration has a name followed by a list of component / view pairs. Users can define multiple configurations in the test database. For example, during gatelevel testing, the original configuration can be modified to use a synthesized netlist:

```
// Derived configuration.
conf c2 : c1 +=
  block1 = netlist
endconf
```

Here a copy of the RTL configuration has been used as a template for the netlist configuration. The new configuration then redefines the view for component 'block1'. The ability to derive things from pre-existing ones is a powerful mechanism for organizing a test suite.

Tools and flows describe the actions that should be taken when running a test. A tool is a detailed description of a job defining where and how to run it. Each tool must specify a directory to run in and a command to run. In addition, most tools need arguments to control the command's behavior. The descriptions for two of the tools from the example would look like this:

```
// Describe tools.
tool make_lib
  dir = $TOP/lib
  cmd = make
endtool

tool chk_sim
  dir = $TOP/test
  cmd = chk_sim.csh
  args  = -test $test
  args += –conf $conf
endtool
```

Notice that the 'chk_sim' tool has been parameterized by using DV variables so that it can be used for different tests. The 'args' property employs one of several operators that can be used to edit an existing property value.

A flow is a list of tools defining the order to run them in. Sequential execution, parallel execution and a limited form of branching are allowed. A flow can fork and rejoin as needed, and can call other flows as subroutines. The complete flow for the example is:

```
// Describe flow.
flow default
  make_lib
  par
    seq
      make_gen
      run_gen
    endseq
    bld_sim
  endpar
  run_sim
  chk_sim
endflow
```

Nested parallel / sequential blocks are used to identify that the generator can be built and run while the simulator is also being compiled. Due to the complexity of its syntax, flows cannot be derived from templates or edited the same way configurations and tools can.

Finally, tests are used to tie structure to action to form a complete test case. The example test description looks like this:

```
// Describe test.
test t1
  conf = c1
  flow = default
  when = daily
endtest
```

A configuration provides the what; a flow and its tools provide the how and where. An additional property 'when' is used as a key for selecting tests to run as part of regressions. A user can define many such keys as a way of organizing the complete set of tests into related groups.

5.2 DV Execution

The DV tool reads and executes tests from the database as selected by the user. The user chooses which tests to run via command-line options. Tests are organized into blocks, with each block database typically containing the tests needed to validate the design block of the same name. The user specifies a block and then a list of tests to run for that block. Alternatively, the user can supply a selection expression which is used to query the contents of the database to find tests that satisfy the expression:

```
# Select one or two tests by name.
1 > dv block1 t1 t2
# Select test with an expression.
2 > dv block1 –where when=daily
```

The main control engine parses the command-line options, assembles a final list of tests to run by consulting the database, and then sends the tests one by one to the execution engine.

The execution engine reads the contents of a test's flow and executes the tools in the order encountered. Procedures for both interactive or batch execution of tools are provided. Special care is taken to avoid running the same tool more than once with the same context.

6 DV Implementation and Use Experience

The DV architecture has been implemented as a pair of cooperating programs running on Sun Solaris and Microsoft Windows NT. The user interface, control and execution engines and the internal database are written in approximately 4000 lines of Perl. A scripting language was chosen over a systems programming language such as C/C++ in order to enable rapid development and evolution of the tool. However, for speed and implementation efficiency reasons, the parser / translator was written as a 1000 line lex / yacc / C program. It converts the DV database to a Perl script that is then invoked by the control engine to set up the internal database. The language description is written in a loose way so that new properties can be attached to the database objects without having to change the language. This makes it easier to implement new features in the Perl program. On Solaris, the execution engine has been interfaced to the LSF batch queuing system to provide parallel execution support.

The original DV implementation was used to verify the 3D graphics sections of the Flipper ASIC, the SOC core of the Nintendo GAMECUBE console. While initially targeted at block-level RTL functional verification, the DV databases were extended by adding configurations and flows to support chip-level verification as well as gate-level verification.

6.1 Phased Introduction

Many of the features of the language and tool were not part of the initial deployment. The requirements for the tool were extended on an ongoing basis as the Flipper project progressed. The basic interactive execution capability was available at rollout, and engineers started using the tool. A major update to add batch execution mode was needed as more tests were written. Parallel execution was added in order to improve performance when using the batch system. In order to support multiple regressions, the expression-based database query system was introduced. Deployment of new features was relatively painless because of DV's plain-text database format and the Perl implementation of the engines.

6.2 Process Capture

One of the common drawbacks of verification scripts is a proliferation of arguments and environment variables that modify the behavior of individual tools. This makes it difficult to transfer verification to other engineers, since one must also pass along these custom settings. DV resists this approach by encouraging users to write a short

DV database that modifies the default behavior. Here is an example of this feature used to enable waveform dumping during simulation of the example test:

```
test t1 +=
  tool.bld_sim.args += -pli_waves
  tool.run_sim.args += -wave_dump
endtest
```

The details of how to enable dumping are encapsulated inside this short database file. The default test description is retrieved and modified by this script. Test-specific tool arguments are used by the execution engine to temporarily modify the tool's execution. It is also possible instead to modify the default tool descriptions so that a set of tool changes applies to all tests.

6.3 Finding Patterns

One of the benefits of capturing tests in a database is the ability to search for patterns in the database. Identifying patterns can influence future efforts in two ways. First, awkward patterns signal a deficiency in the scope of the database language. By analyzing such a pattern, modifications can be made to the language to make it simpler to express the pattern. For example, early DV databases contained the following pattern:

```
// Base test.
test t =
  conf = c1
  flow = default
endtest

// First derived test.
test t1 : t +=
  tool.test_run.args = "-obj 1"
endtest

// Second derived test.
test t2 : t +=
  tool.test_run.args = "-obj 2"
endtest
```

This method for defining a series of related tests was quite common. However, because each new test requires four lines in the database, it is cumbersome to add a new test or maintain large sets of tests. In response, the language and tool were modified to allow a single-line format for this pattern:

```
// Derived tests.
test t1 : t += args = "-obj 1"
test t2 : t += args = "-obj 2"
```

The matching 'end' keyword can be omitted when modifying a single property, and the alias 'args' has been introduced for the commonly occurring 'tool.test_run.args' property.

Second, good patterns can be documented and communicated to other members of the team. For example, the following pattern can be used to derive a series of tests of escalating functional complexity:

```
// Escalating difficulty series.
test t1 : t  += args =  -func 1
test t2 : t1 += args += -func 2
test t3 : t2 += args += -func 3
```

Each new test is derived by copying its predecessor and enabling a new function. When fed input from a random generator, this series can test the interactions of each new feature against all previous ones.

7 Conclusions

A verification flow management tool called DV has been developed as a natural extension of the methods used to bring a fragmented design environment under control. The main benefit of such a tool is the ability of verification engineers to automatically run the thousands of tests required to ensure that an ASIC design is bug-free. The DV tool is architected on an underlying model of design verification that encompasses both the design structure and the verification flow. At the same time, its implementation is guided by the constraints of a startup company in a way that allows it to evolve with the company. By capturing the verification process in an executable format, it has become a foundation tool for ASIC design projects.

8 Acknowledgments

The author would like to thank the ArtX/ATI Flipper ASIC development team, the first users of the DV verification tool described in this paper. Their contributions in the form of ideas and idioms were invaluable to its early development.

9 References

[1] J.B. Brockman, *A Schema-Based Approach to CAD Task Management*. Ph.D. Thesis, Carnegie Mellon University, Department of Electrical and Computer Engineering (Report CMUCAD-93-01), Jun. 1992.

[2] Platform Computing, Inc., *Platform LSF 5 Product Brochure*, URL: http://www.platform.com

[1] L. Bening and H. Foster, *Principles of Verifiable RTL Design*. Kluwer Academic Press, 2001.