Introduction

Failing tests; schedule slips; silicon re-spins; development tools that break with each new design. A growing number of test engineers are faced with these critical issues as they struggle with a test development process that has not kept pace with increasing device complexity. At issue is management of multiple test pattern sets at the core-level and preparing them for application at the chip-level for an SoC design. Simplistic test conversion tools, held together with ad-hoc scripts represent an outdated methodology that introduces test program errors, makes development schedules unpredictable, and can ultimately impact test quality. Many test teams have determined that a more formal approach must be applied to test pattern integration to achieve desired results in a more predictable fashion.

In this white paper we’ll explore an alternative to writing custom scripts to handle test pattern data. We’ll present an approach that utilizes a test data structure. It offers a process that is more immune to errors and achieves improved pattern readiness on a predictable schedule.

Managing Test Patterns

Creating manufacturing tests for the today’s complex devices is a significant undertaking with many stringent requirements. A typical test development process can be divided into 5 basic steps as shown by Figure 1.

![Diagram](image)

Figure 1: A traditional test development approach applied to more complex designs can rely too much on debugging first silicon on the ATE.

Typically, much attention is spent upfront creating high quality test pattern sets utilizing a multitude of different techniques. At the other end of the process, usually a significant amount of resources are put into debugging test program problems on the ATE with first silicon. However, the steps in between—merging, converting, ATE targeting, and verifying test patterns—may not be getting the attention they deserve. Instead, a process that incorporates ad-hoc scripts and disjointed tools is used instead of using a robust methodology. This can make the test development time unpredictable and foster hidden problems that will show up late in the process.

Multiple Pattern Sets and Formats

Very few devices today have just one pattern set. A test plan normally consists of several different types of tests to get high coverage. Many devices utilize structural scan tests to achieve thorough test coverage. Built-in self-test (BIST) methods are also used extensively, along with JTAG (IEEE 1149). Functional patterns still persist as a way to round out the test suite. Scan pattern formats are typically STIL (Standard Test Interface Language) or WGL (Waveform Generation Language). Functional patterns are typically created from a Verilog VCD (value change
dump) or EVCD (enhanced VCD) files. Some test vectors may even be generated by hand and represented by simple ASCII format.

Nearly all designs today rely upon design reuse, utilizing IP (intellectual property) from a variety of sources. Many times hard IP will come with existing test patterns. Not only do these patterns save test development effort, but they also lessen the burden of top-level pattern generation. Further, when properly isolated, these IP cores may be tested in parallel with test scheduling, saving substantial test time. Even with soft IP cores, a hierarchical approach to test pattern development may be the most efficient in terms of development and test execution time.

Managing the test scheduling for individual blocks can be a complex task. Several things must be considered:

- How many chip-level pins are available to access?
- Can the cores be isolated during concurrent testing?
- Which clock domains need to be active?
- Which power supplies need to be active, and will the test remain within power limits?
- Will a reset signal be needed?

After considering all these aspects and developing a schedule for the individual tests, the next step is combining test pattern sets that come from the multiple of sources mentioned earlier. This can be a difficult task and is the focus of the rest of the discussion.

**Existing Techniques for Managing Test Patterns**

Until recently, test pattern generation—be it structural or functional—was comprised of a test set created at the chip-level. This approach resulted in a single test pattern file that could be converted to a specific ATE format. Many test pattern conversion methods have been optimized for this type of approach. Unfortunately, extending these methods to handle multiple test sets can be problematic because the approach is not easily scaled.

**The STIL and Perl Approach**

Starting about 25 years ago, WGL became a de facto standard for describing cyclized test pattern vectors in a neutral format. More recently in 1999, STIL was developed through IEEE (STDS-1450) to provide a similar function. Today, both are used extensively—for example, ATPG tools typically provide output in either format and some ATE platforms will even process a strict subset of STIL or WGL.

![Figure 2: When ASCII file formats such as STIL or WGL are used as medium for the test conversion flow, manipulations are usually done with Perl scripts or similar.](image)

Though the purpose of STIL and WGL is to provide a consistent method of interchange between various tools, many test engineers are also using these ASCII-based formats for a test pattern database. In such situations, the manipulation of test data is typically done with general purpose scripting languages such as Perl, as illustrated by Figure 2. When employing this method, one tries to minimize changes to the original file to avoid injecting errors. Unfortunately even relatively simple changes such as renaming a signal can get quite involved as illustrated by Figure 3. Though this method may be expedient for relatively simple changes, it can quickly become unmanageable as the amount of data, the number of files, and the complexity of the operations increase when dealing with larger
designs. A tangle of long Perl scripts limits flexibility, error-checking, and capacity. Maintenance of such scripts can become a full-time job, which puts test development schedules at risk.

```plaintext
STIL 1.0;
-
Signals {
    core_input_01 In;
    -
SignalGroups {
        inputs_group1 = 'core_input_01 +
        -
Timing load {
            WaveformTable default_WFT_1 {
                Period '30ns';
                Waveforms {
                    core_input_01 { 01NZ { '0ns' D/U/N/Z; } }
                -
```

Figure 3: STIL file snippet showing different sections that must be edited for renaming a signal

The ATPG Tool Approach

ATPG tool vendors have recognized that the patterns they generate for scan must typically be modified before they are ready for chip-level application. For this reason, ATPG tools are offering more options for modifying patterns after initial pattern generation. However, there some short-comings with relying exclusively upon ATPG tools to perform these pattern modifications. First, the ATPG tool may not be able to handle patterns that come from other tools. Second, it assumes that person doing the pattern integration has access to and knowledge of the ATPG tool. Many times this is not the case, where scan patterns are simply delivered to the Test Engineer, without the ability to go back to the ATPG tool.

The ATE Tool Approach

ATE vendors also offer tools that can help with integrating patterns from multiple sources. However, it requires that all patterns are going to only a single ATE platform. If a test team is targeting multiple ATE platforms, they will need to use a different process for each platform. Further, the ATE may not necessarily be available, as is the case if the ATE is being used in production, or if testing is done by another company.

Full-Chip Pattern Integration

It’s clear that test pattern integration for SoC designs needs a flexible approach that is scalable, easily maintained, and produces patterns within a predictable timeframe. It needs to be able to take test patterns from multiple sources and perform some common operations without risk of test data corruption. We’ll refer to this approach as **Full-Chip Pattern Integration**. This approach assumes that multiple pattern sets are gathered into and manipulated from within a controlled test data structure. The integrity of data from various input pattern files such as STIL, WGL, VCD, and EVCD will be maintained because they will not be edited directly. Instead these input files are converted to a test data structure before any operations are applied, as shown by Figure 4.
Figure 4: Editing of test patterns within a controlled data structure avoids corruption and syntax errors.

**STIL Reader**

An effective STIL reader is essential for this process. Analogous to spoken languages, STIL has many constructs and dialects. The STIL specification encompasses a wide range of test data aspects, including some that do not necessarily pertain to the application at hand. The STIL reader must be able to read and process the test pattern information that comes from the various test generation sources. This includes handling all STIL (and WGL) files that come from the major ATPG tools such as Mentor Graphics Tessent TestKompress®, Synopsys TetraMax™, and Cadence Encounter Test™. It’s best to avoid the need for preprocessing these files with scripts, whence create a loophole for data corruption.

Consistency checking of incoming STIL data is sound strategy. Obviously, bad input data will quickly doom the process. Sometimes data that was supposed to come directly from the ATPG tool may have been unknowingly hand-edited. Or the settings on the ATPG tool could have been improperly set. It may be possible to correct bad test data if it has been promptly flagged upon input. For example, if some input test data was left undefined, you may choose to still input the test data and create a process step that provides specific values once the test data has been converted to a test data structure.

**Test Pattern Manipulations**

Each SoC design presents a different set of test development challenges. Fortunately, most of the required operations on test patterns can be achieved by using some combination of a basic set of functions, as listed in Table 1. In the next sections we’ll step through these functions and how they may be used.

<table>
<thead>
<tr>
<th>Rename signals</th>
<th>Insert preamble before pattern set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add/clone/delete signals</td>
<td>Concatenate pattern sets</td>
</tr>
<tr>
<td>Invert signals</td>
<td>Add pipelining stages to scan change</td>
</tr>
<tr>
<td>Change to differential</td>
<td>Replace/mask bits</td>
</tr>
<tr>
<td>Change signal direction</td>
<td>Add/delete members to group</td>
</tr>
<tr>
<td>Add/clone/delete vectors</td>
<td>Add comment</td>
</tr>
</tbody>
</table>

Table 1: Basic set of pattern editing functions

**Rename signals**

Renaming signals can have any number of applications. One common application is to change core-level test pins to chip-level connections as shown by Figure 5. For many applications the same target names could be applied to
multiple cores because all core level pins need to be mapped to a common set of chip-level pins. In such a circumstance, the signals are likely to be multiplexed or similar. In the case of test pattern data (unlike Verilog), the multiplexing operation need not be done explicitly. Instead, the multiplexing can be accomplished by specifying a preamble, as described later.

![Diagram](image)

**Figure 5: Remapping core-level signals to the chip level**

### Add/clone/delete signals

Adding and deleting signals are common operations. For example, if logic is added to enable test access, signals will need to be added to the patterns. Of course, vector data and timing will also need to be added for these signals. In such cases, cloning can be more expedient if a similar signal already exists. Then slight changes to the bit patterns can be performed on the clones.

Deleting signals involves more than just getting rid of the port. It should also delete all test data and timing uniquely associated with that port. One circumstance where this can occur is if some ports of a core cannot be routed for chip-level access. Of course, these I/O should be “X’ed out” accordingly when generating the patterns. Since deleting signals can get rid of a significant amount of unnecessary test data, it may be best to perform the delete operation first to trim the test data files, enabling faster downstream execution.

One interesting application for cloning signals is enabling parallel testing of identical cores, as illustrated by Figure 6. By cloning the scan outputs, test patterns can be applied to multiple cores without increasing the number of inputs.

![Diagram](image)

**Figure 6: Example of concurrent test of identical cores**

### Invert signals

An efficient way to perform an inversion for test pattern data is to change all data bits to their complement. It’s a good idea to change the name of the signal to reflect the inversion.
**Change to differential**
Changing to differential can be accomplished by a combination of clone and invert.

**Change signal direction**
In some cases input ports may need to be changed to bidirectional. Or maybe an input and output ports need to be combined because of limited access pins.

**Add/clone/delete vectors**
It’s easy to envision many applications where vectors need to be added or deleted. Similar to operations done with signals, cloning a vector and then modifying particular bits may be more expedient than specifying each bit of the new vector.

**Insert preamble before pattern set**
Core-level pattern sets will typically need a set-up (preamble) before execution (see Figure 7). Sometime this can be fully specified during ATPG, but many times additional setup will be necessary after integrating the core into the chip level.

![Diagram showing pattern set set-up](image)

Figure 7: Patterns generated for a particular block will likely need a set-up sequence (preamble) before applying the patterns.

Preambles are essentially functional patterns. One way to generate a preamble is to perform a simulation, cyclize the VCD, and then prepend it to the pattern set. If the preamble is simple, maybe it is best done by simply specifying individual bits. But in such a case, pattern verification is especially important to flush out any mistakes made during this manual specification.

**Concatenate pattern sets**
Though concatenating pattern sets can usually be done on the ATE, it may be better to join them together earlier. The concatenation operation should avoid creating redundant timing existing between pattern sets.

**Add pipelining stages to scan chains**
Adding pipelining stages to scan chains is yet another operation that can many times be done with an ATPG tool, but it may also come up later during SoC integration—especially when dealing with hard cores. A simple illustration is shown in Figure 8. Adding pipelining to a test pattern set can be accomplished by padding the chain data.
**Change/mask bits**

As mentioned earlier, changing individual bits works well when used in conjunction with cloning signals or vectors. Masking failing bits comes up often. The question is usually at what stage to mask bits. Going back to ATPG is an option, but this is not always feasible. In some cases, masking on the ATE may be the most suitable approach. However, this can result in patterns that are less portable. In many circumstances, the best solution may be to perform the masking as part of the test conversion flow. This will enable the masking to be automated into the process.

**Add/delete members to group**

Groups are a useful constructs in STIL. Among other things, they provide a means of specifying timing for multiple signals. Having a way to add members to a group makes it easy to specify timing for new signals as they are added.

**Add comment**

Adding and preserving annotations and comments can be important in applications that are intended to be post-processed. Of course, they can also be valuable for documenting test data that is output back to STIL format.

**STIL Writer**

As Figure 4 shows, a reliable STIL writer is an important part of utilizing a test pattern data structure. Of course, the STIL produced by the writer must be syntactically true to the IEEE 1450 standard. On the other hand, it should not be needlessly complex. As mentioned before, STIL is a rich language and is open to being syntactically obtuse. A STIL writer that employs complex constructs can needlessly create parsing problems downstream.

**Pattern Verification**

Thorough verification is an important ingredient for all stages of the IC development process. Any change to the test patterns—regardless of how simple—may necessitate verification. An expedient method for verifying the test patterns is achieved by creating a relatively simple testbench that accesses the patterns directly during simulation through a Verilog PLI, as illustrated by Figure 9. Such a method avoids creating a large test bench which includes all the test vector data. By using this method, test data can be modified without recompiling the Verilog.
Figure 9: Accessing the test data structure through a Verilog PLI keeps the test bench simple

Converting Pattern to ATE-Specific Format

Looking back at Figure 4, up until this point we have assumed that test patterns eventually end up back as a STIL formatted file. However, it’s likely that this is not the final conversion of the data. Instead, in all likelihood the data will be converted to an ATE-specific format. Some ATE read STIL directly, but most require at least some special conversion. Therefore, it may be more feasible to go directly from the test data structure to ATE-specific format without writing out STIL. This avoids needless pattern conversion and creation of large STIL files.

Summary

Full-chip pattern integration is a systematic approach to SoC test pattern development. Instead of a patchwork of scripts manipulating ASCII and a heavy reliance on post-silicon debug, it utilizes a common test date structure and a reliable set of operations to methodically prepare SoC test. Effective error-checking and verification eliminate problems early, thus making the test development process more predictable and minimizing program debug on the ATE.