

# A Holistic Parallel and Hierarchical Approach towards Design-For-Test

C. P. Ravikumar  
Texas Instrument (India)  
Wind Tunnel Road, Murugeshpalya  
Bangalore 560017, India  
ravikumar@ti.com

G. Hetherington  
Texas Instrument Ltd  
800 Pavilion Drive, Northampton  
NN4 7YL, United Kingdom  
g-hetherington@ti.com

## Abstract

*While design-for-test methods such as scan, ATPG, and memory BIST are now well established for ASIC products, their run-time for multi-million gate designs has become a problem. Too often, a tape-out is held up because pattern generation and verification are incomplete. This paper describes a holistic design-for-test approach which exploits both hierarchy and parallelism on every aspect of the DFT to minimize the run-time impact.*

**Topics:** *Experiments and Case Studies, Practical Test Engineering*

## 1 Introduction

Improvements in manufacturing technology continue silicon technologies shrinkage to devices sizes in the deep submicron regions to 90 nanometers and below. This leads to continuing growth in design sizes. In recent years typical ASIC design sizes have grown from 100's of thousands of gates to millions of gates. Meanwhile, the time available to design, manufacture and test the product is shrinking due to market pressures. This is especially true for low-volume ASIC products which have a limited market window.

These two vectors of design technology development lead to increased demands on manufacturing test [1,2,3,5,8, 17]. Traditional approaches of applying stuck-at and IDDQ automatic test pattern generation (ATPG) test patterns and built in self test (BIST) for memories are not sufficient to detect all manufacturing defects and marginal parts. Today's set of test patterns will include at-speed tests such as transition fault and path delay ATPG patterns. Additional test patterns are produced that stress voltage and/or temperature to detect reliability problems. As design size increases, so does the volume of test patterns. But, perhaps more importantly, the amount of *engineering effort, computational resources and run times* required for DFT work are also increasing.

DFT work must proceed without impinging on the rest of the design activity. Crunch occurs when, at tape-out, the ASIC manufacturer will demand a complete set of manufacturing tests that meet stringent fault coverage goals which have been verified on the final tape-out design data.

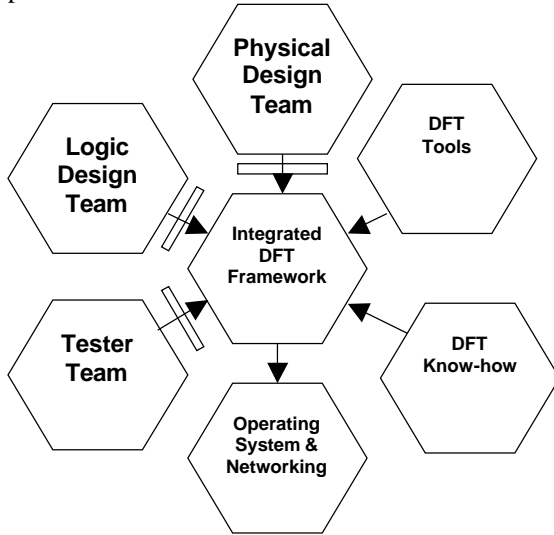
A good selection of commercial point tools is at hand to assist the DFT engineer -- tools which analyze RTL design descriptions, tools which insert scan and BIST and generate various types of ATPG patterns, tools to perform static timing analysis of the design and dynamic timing simulation verification of test patterns. Although these tools are good, they have to be aimed with custom scripts and command files at specific designs. Due to company-specific requirements for test pattern types, the tools have to be configured, again with scripts and do files, to generate the complete set of required test patterns. These test patterns must be compliant with the targeted tester. Finally, for convenience and throughput, the tools must be strung together, again with scripts, to run the whole DFT process non-stop. This engineering effort eats into the design cycle time and reduces the efficiency of the DFT engineer. The DFT process varies from one design group to another, even from one design engineer to another. The price to be paid for this is that the product engineer may receive "surprise" patterns that confuse and slow down the silicon ramp.

Agarwal expresses similar concerns in [2] and states, "as devices have become more complicated and incorporate hierarchical design methodologies, the rounds of translation and compiling test programs necessary to make ATPG work have increased drastically, squeezing the time-to-market window."

Based on these observations, there is a strong need for a DFT framework that can be a DFT engineer's assistant – an interface that speaks and understands the language of the DFT engineer, knows the DFT requirements of the company, has expert knowledge built in. A framework that relieves the DFT engineer of the mundane script generation and job running chores handling these to best practices and getting the most performance out of the point tools and compute hardware. This paper presents such a framework, one that has been used on real designs.

Figure 1 illustrates the concepts in a pictorial way. The Integrated DFT framework must interface with inputs from the logic design team and the physical design team. The tester team provides tester constraints to the framework, and the role of the framework is to return good, tester-ready patterns to the tester team. The framework encapsulates the following three types of knowledge – DFT tool know-how in order to generate the command files for DFT jobs, DFT know-how in order to decide the appropriate DFT architecture and the type of patterns to generate, and OS/Networking knowledge in

order to submit the jobs to a distributed computing platform.



**Figure 1: Integrated DFT Framework**

The paper is organized as follows. The next section describes the various tasks involved in ASIC DFT and emphasizes the need for improving the DFT cycle time through a framework. We also include a review of the previous work in the area of design frameworks. Section 3 describes our approach in developing a DFT framework. Section 4 provides details of the software we have developed to support this framework. Section 5 reports experimental results of using our framework on several designs. In Section 6, we summarize our contributions and indicate the scope of further work.

## 2 ASIC DFT Cycle

ASIC designs are characterized by aggressive design cycle times. Therefore, it is common for DFT engineers to maintain a simple test architecture and use it repeatedly on several designs. Full-scan methodology and ATPG are normally used for testing the digital logic; built-in self-test (BIST) is used for testing the embedded memories. The ASIC DFT architecture depends on the size of the design. Smaller designs can use a flat design methodology and a “plain vanilla” scan DFT with a small number of scan chains pervading the entire design. Larger designs must exploit the hierarchy in the design and reuse sub-blocks that already have built-in DFT structures. There are several variations on the essential scan architecture, including Illinois Scan, Reconfigurable Scan, Partial Scan, and so on. The IEEE P1500 standard applies to system-on-chip designs based on IP cores [4, 17]. Logic BIST (or embedded Test) is seen as a way to address the problems of growing test data volume and test application time [2, 11, 17].

Several types of test patterns need to be generated for an ASIC. Embedded memories are self-tested using

marching algorithms and checkerboard patterns. On-chip PLLs are tested using BIST logic. Scan test patterns are applied to test the digital logic in the core of the chip. The single stuck-at fault model continues to be popular for logic test. Transition delay and path delay fault models are now being employed for at-speed testing. Parametric tests are useful to make measurements of DC parameters. IDDQ tests can reveal defects that cannot be modeled as stuck-at faults; IDDQ tests can also catch reliability problems in the chip. Burn-in testing is carried out to eliminate chips that have poor reliability. Low-voltage and high-voltage stress tests are used to catch delay defects and reliability problems. When excessive pin count, beyond the tester’s capacity, is a problem, loop-back testing is used. Since growing scan test volume is an issue, deterministic BIST and test compression schemes are becoming popular for large designs.

The planning of DFT work for an ASIC is done at a very early stage, and targeted DPM number and fundamental DFT methodology are documented. Early planning helps the design team in deciding the number of pins and silicon area to be dedicated for testing. Test Coverage targets for various types of tests are decided and any special types of testing, such as testing of special cells or IP cores, are listed and reviewed. When the design team owns the complete RTL-to-tape-out flow, DFT synthesis can be performed at RTL and timing closure can be achieved in one pass. For ASIC designs, it is common practice for design companies to contract Backend and DFT work to the semiconductor vendor. In this engagement model, the DFT engineers do not have access to RTL and must perform DFT insertion and ATPG on the netlist. Refer to Figure 2. The RTL designers will typically synthesize sub-blocks in the design and make the block-level netlists available to the DFT team, which, in turn, performs block-level insertion of DFT structures and coverage estimation through block-level ATPG. DFT insertion includes insertion of memory BIST collars and controllers, and scan chains. Using block-level timing constraints and approximate delay information, timing verification is performed using static timing analysis. Formal verification, in the form of equivalence checking, is used to verify that DFT insertion has not altered design functionality. Block-level test coverage reports, timing reports, and formal verification reports are reviewed and design iterations are recommended if the coverage is deemed inadequate, formal verification reports have failures, or significant timing violations are noticed. Chip-level DFT work includes deciding a top-level test access mechanism and hooking it up in the netlist. This is followed by the physical design, where some reordering of scan flops may be necessary to meet timing. Formal verification is performed at the chip-level to ensure functional equivalence after top-level DFT insertion. Chip-level ATPG patterns are then generated and validated. Delay information available from the physical design team may often be approximate, since the team

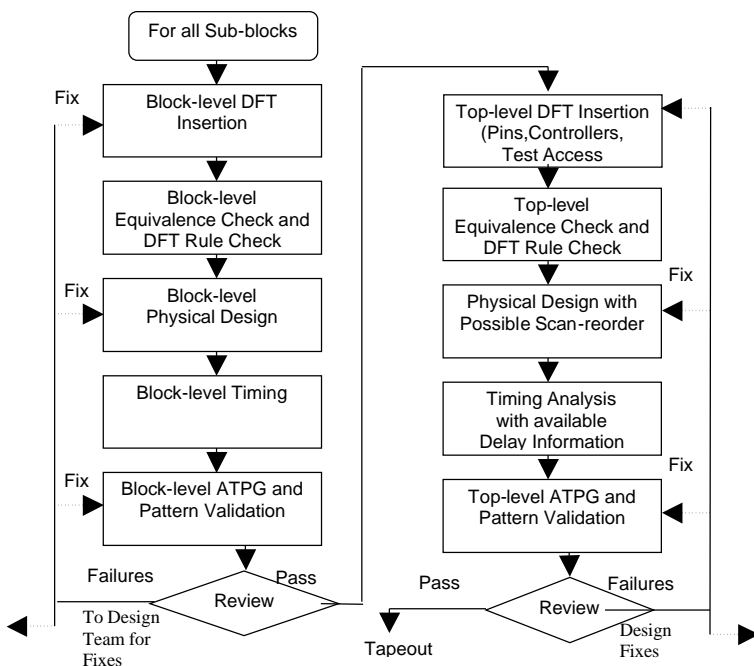
may continue to optimize the layout and fix timing; in nanometer technologies, design closure is complex due to issues such as crosstalk, IR drop, and reliability. Top-level ATPG coverage reports, timing reports, and formal verification reports are reviewed to ensure that coverage is adequate and there is no functional verification or timing closure issues. In addition to validation of test patterns for timing correctness, their tester-compliance is checked to ensure that constraints are met e.g. the tester has sufficient memory for the patterns. When final parasitic extraction is complete and accurate delay information is available, a re-iteration of the pattern validation and timing verification effort is necessary.

It will be evident from the above discussion that the DFT cycle is a complex and iterative one, requiring interface with logic design team, physical design team, and the tester team. Sub-blocks in current-day ASICs are close to a million gates in size. Chip-level operations such as formal verification, physical design, ATPG, and pattern validation can therefore be very intensive in run-time. Memory requirement for some of these jobs is also very high, placing constraints on the type of machines on which they can be run. Physical designers and timing verification engineers use hierarchical methods to combat this problem. For example, by extracting timing models for circuit blocks, the effort involved in static timing analysis can be reduced, since timing paths that are entirely internal to a block need not be considered at the chip-level. A similar approach is needed to reduce the effort in ATPG and pattern validation.

Another observation on the DFT cycle is that it calls on the DFT engineers to acquire different types of skill sets, such as DFT insertion, formal verification, ATPG, simulation, and so on. The DFT engineer must also spend significant amount of time tracking the completion of jobs and deal with OS-related issues and optimizations. To give an idea of the complexity of this work, consider a chip with 5 million gates, consisting of 5 sub-blocks of 1 million gates each. Assume 10 types of scan ATPG patterns and 3 simulation corners on an average for each type of pattern. Assume that at the block-level, we obtain about 5,000 patterns for each block, which are split into 10 files. The block-level DFT work will generate 50 ATPG jobs and 1500 simulation jobs. A somewhat larger number of ATPG and simulation jobs are necessary at the top-level, assuming that there will be multiple test modes. Gathering the data and setting up the jobs can involve human errors, resulting in wasted effort and costly iterations. This brings out the need for a DFT framework to improve the DFT cycle time; since tools only address specific issues, only a framework can offer the much required holistic solution.

## 2.1 Previous Work

Several authors have written about the increased IC complexity and the associated growth in test cost [2, 6, 8, 16, 17]. The challenges posed by the deep submicron technologies to test engineers has been expounded by several authors – see [8] for a tutorial on this subject. Agarwal [2] emphasizes that hierarchical design methods are now growing in popularity and feels that ATPG is not the right solution for system-on-chip designs. Aizawa and Thummarakudy [1] discuss the benefits of hierarchical design techniques for chips larger than 3 million gates. Several EDA vendors now support hierarchical design techniques for logic design, physical design, and scan synthesis. Agarwal [2] does not recommend ATPG for hierarchical designs larger than 1 million gates, stating that “ATPG requires several rounds of manual translation, compiling test programs and substantial database management to make this approach work properly.” The author recommends hierarchical embedded test as an alternative. In this paper, we argue that ATPG has still plenty of juice left and a suitable DFT framework is what is required to keep this time-tested method going for several years more. In fact, we have effectively used the proposed methodology on large ASICs with over 8 million gates. This is not to down rate the importance of embedded or built-in self-test techniques. We believe that without a DFT framework, the best DFT techniques will not be easy to practice. Although the DFT framework described in this paper uses scan ATPG for logic testing, logic BIST must be a part of the framework for addressing the issue of test data volume. At the time of writing, we are extending the DFT framework to support Logic BIST techniques.



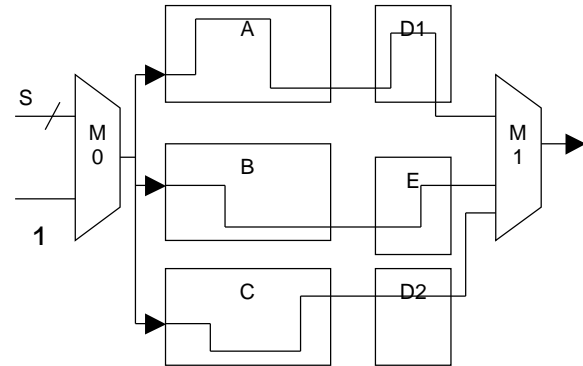
**Figure 2: ASIC DFT Cycle**

DFT flows have long been recognized as essential components in the chip design process [5,9,10]. In Birkel et al describe a DFT flow based on commercial EDA tools for a 3-million gate MP3 decoder chip [5]. Chickermane et al. describe a push-button flow based on in-house tools for inserting top-level I/O and test structures in an ASIC design [10]. The test structures include the test-related I/O ports, boundary scan cells, and the IEEE 1149.1 test access port. The user specifies an input file where the requirements of all I/O ports are captured. The aim of a DFT flow is to reduce the effort involved in the insertion of test structures and the associated timing closure, generation of good ATPG patterns, and validation of the patterns. The “test bringup time” on the tester is a metric to evaluate the effectiveness of a flow; after the first prototypes arrive, a test engineer must be able to bring up the tests in a few hours. When the DFT engineers have access to RTL code, it is possible to run early checks on DFT compliance as well as perform test synthesis at RTL. It is also common for RTL design team to compile the design with scan flops and pass on the netlist to DFT team, which in turn, is responsible for inserting DFT structures such as scan and BIST. Since no single DFT tool supports all the functionalities required in a DFT flow, it is common to employ a bevy of in-house tools and tools from different vendors. In the early nineties, there was a significant interest in the EDA community to create CAD frameworks [7]. The intention behind a CAD framework is to free a designer from being forced to acquire tool skills and design environments and offer a high-level view of design objects which permit the designer to focus on the design tasks. Our literature search did not reveal any published work on DFT framework. Work on design frameworks, based on expert systems, artificial intelligence, and neural networks, was reported late eighties and early nineties, but these techniques did not create much impact [15]. The fundamental reason for this is that design and test techniques are constantly evolving and designers feel the need to keep the control in their hands. Canned tools and flows that do not offer the flexibility of change will therefore be of limited value. The DFT framework described in this paper builds on this tenet and a key goal in the design of this framework was to offer complete flexibility to the DFT engineer in terms of tools, job control, and command files for DFT work.

### 3 DFT Cycle Time Reduction

Adopting a small number of standardized DFT architectures makes the process of DFT insertion less of an art form and is a key to DFT cycle time reduction. What DFT architectures to support is a choice specific to the organization, but a guideline is that the architecture must support hierarchical ATPG and pattern validation. The divide-and-conquer (DAC) scan architecture we describe in this section is a hierarchical scan test architecture which is useful when the design has several

blocks, multiple instances of the same block, and the gate count in the design is more than a million gates [?]. Refer to Figure 3. Plain vanilla scan-insertion is used on individual blocks. The instances of sub-blocks are partitioned into *test groups*. Each test group *G* consists of one or more instances, and the total gate count in a group is a manageable number, say 1M. A test access mechanism (TAM) is inserted, which permits testing of each *test group* at the top-level. The TAM is based on scan multiplexers and can be synthesized from an auto-generated RTL. The resulting gate-level scan-mux logic is integrated into the original design along with the scan chains to obtain a final scan-inserted netlist.



**Figure 3: Divide-and-Conquer Scan**

Figure 3 illustrates DAC scan architecture on a chip with 13 instances of sub-blocks. Test Group TG1 includes the instances A, D1, F1, and G1. Similarly, TG2 includes B, D2, E1, F2, G3, and G2. Block instances C, E2, and G4 comprise the test group TG3. This partitioning into test groups was performed by roughly keeping the size of each test group nearly the same, so that the number of (stuck-at) faults in each group is nearly the same. The scan multiplexers are added to provide a test access mechanism. By setting M0 control to 0, the scan input SI can be fed to the scan input of block A. The scan chains of the block instances in a group are wired together as shown in the figure. Note that all blocks have the same number of scan chains. By setting the scan multiplexer M1 control to 01, and by selecting the scan multiplexer control M3 to 00, the scan output of block G1 can be observed at SO. When group TG1 is being scan-tested, constant 1's are scanned into the scan chains of the other two groups; this significantly reduces the scan test power dissipation in TG2 and TG3. In the “daisy chain” or “residual” mode, there is a path from SI to SO that spans all blocks.

There are  $g+1$  test modes in DAC scan, where  $g$  is the number of test groups. DAC scan permits us to apply scan patterns to each individual group and treat the group as a sub-circuit with its own scan-in and scan-out pins. Execution of an ATPG or simulation tool on a test group will take significantly less time than execution on the top-level netlist. This is because we *black box* the logic that is

not part of test group G when we perform ATPG for the corresponding test mode. Similarly, during simulation of test patterns, we can replace all the logic not part of the current test group by *empty boxes*. This reduces the effective netlist size and significantly reduces the effort involved in simulation. Since only one group is scanned at a time, the switching activity during scan shift is reduced. This reduces power dissipation during scan shift. The total scan volume and test application in a divide-and-conquer ATPG is smaller, since the effective scan length in test modes other than the “daisy chain” mode is much smaller than the scan length in the daisy chain mode.

Let U be the set of all stuck-at faults in the logic core, barring the faults in the black-boxed objects such as memories and macros. The stuck-at faults covered by group-level ATPG runs are saved, and let these be indicated by  $L_1, L_2, \dots, L_g$ . The list of faults that are left uncovered after the application of group-level tests is  $U - (L_1 \cup L_2 \cup \dots \cup L_g)$ . These faults are tested during “daisy chain” mode. The effort in generating patterns in the mop-up mode is thus reduced significantly, and the speedup is given by  $|U| / |U - (L_1 \cup L_2 \cup \dots \cup L_g)| + K$ , where K represents the overhead involved in saving the patterns in group-level testing and reading the fault lists during mop-up mode. This overhead becomes less significant as the size of the group is increased and there exists an optimal value for the size of individual groups and the number of groups when the overhead is minimum. The same divide-and-conquer ATPG procedure can be used for transition-delay and voltage stress testing also. The reliability test pattern types (burn-in, IDDQ, DC-parametric, and latch up tests) are generated for the “daisy chain” mode to maintain the robustness of the patterns.

Best practices of simulation are incorporated in the scripts generated by the framework e.g. the optimal setting of switches to the simulation tool. The auto-generated scripts include the appropriate simulation corners. The validation flow, which is based on simulation of vectors, consists of generation of a test bench, compilation of the netlist and back annotation of delay information, and actual event-driven simulation. It is possible to eliminate some of these tasks e.g. we can compile the netlist once and reuse the data structure multiple times during simulation.

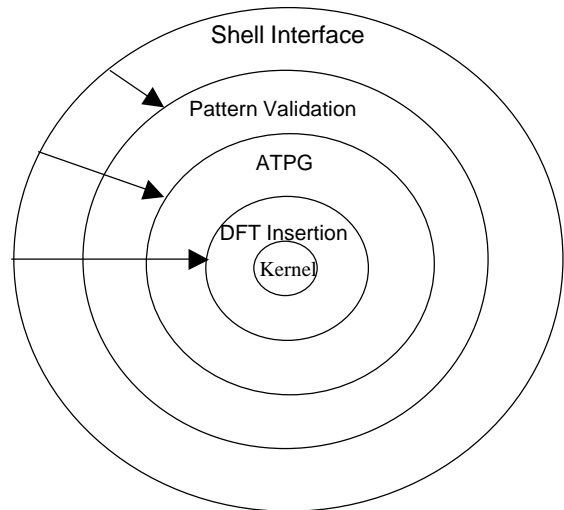
The DFT framework permits distributed execution of ATPG and simulation jobs on a network of computers (running different operating systems, Solaris or Linux, 32-bit or 64-bit) to achieve speedup. An in-house task scheduling tool was employed for this purpose, which uses the Load Sharing Facility (LSF) to distribute jobs on the network. The benefit of using a scheduler is that it offers a simple graphical interface to report the status of the jobs and a way to control the execution of tasks.

One of the early decisions during the design of the DFT framework was to provide support for more than one tool

for every step in the DFT flow. This was based on the observation that each EDA tool has its strengths and weaknesses. Also, when all the licenses of a specific tool are exhausted, using the DFT framework it is possible to reassign some of the jobs to alternate tools to save time. Similarly, the framework allows the user to select the operating system to run a job. For example, the “daisy chain” ATPG and simulation are memory intensive and may be run on 64-bit OS, while the group-level ATPG and simulation can be run on 32-bit OS. Similarly, the user can specify either Solaris or Linux to run a job.

## 4 An Integrated DFT Framework

The DFT framework software discussed in the previous sections was implemented in Perl. The structure of the software is shown in Figure 4.



**Figure 4: DFT Framework Software Organization**

The basic user interface to the framework is through a command shell. This decision is based on the fact that commercial DFT tools for insertion and ATPG are also driven by command shell interfaces. The other components of the framework include Perl modules for auto-generation of scripts for DFT insertion, ATPG, and simulation. The “kernel” of the framework consists of the DFT tools, the cell libraries, and the basic data structures used by the framework. The three essential types of flows that the flow supports are indicated by the arrows in Figure 4 and are (i) the complete flow, which includes DFT insertion, ATPG, and pattern validation, (ii) Only ATPG and Pattern validation, and (iii) Only Pattern validation. Each step in the flow may consist of several smaller steps e.g. DFT insertion includes insertion of scan structures, post-insertion DRC and formal verification. Through shell command interface, the user has the flexibility of running a “flow step”, which stops at the granularity of the sub-steps. To provide complete

flexibility, the user can also define additional flow steps such as some format conversion scripts that may be necessary in a specific design. The framework uses its own directory structure, which may pre-exist or which the shell can create. The directory structure is nested into four levels – flow steps, design objects ATPG types, and simulation corners. A “design object” refers to a sub-block or the full-chip for which the flow step must be executed. At the same level as “flow steps,” an “input” directory is provided where the user can load the netlist(s) and other inputs to the flow such as delay models. In each of the directories for insertion, ATPG type or a simulation corner, we provide sub-directories for the inputs, outputs, and log files to run the corresponding flow step. The input to a particular flow step may come from the output of another step, and the data movement is done by the framework automatically.

Although the framework can be used interactively, it is easiest to write shell scripts and run them through the framework. The shell compiles the script and gathers information about the design, the DFT environment, and the jobs that the user wishes to run. Commands can be classified into

- Commands to change basic kernel variables, such as the ATPG tool to be used, the location of the tool, the license file for the tool, and so on. An example is **set insert tool dftadvisor -type 32 -location /proj/dite/bin -block B\*** (Use the 32-bit version of DFTAdvisor tool for scan insertion on blocks whose names begin with B.)
- Commands to select the DFT configuration e.g. the scan architecture, scan methodology, number of scan chains, tester used, test groups, test clocks, etc. An example is, **set insert scan\_input -block A\* SCANIN** (For all blocks with names beginning with A, let SCANIN be the name of the bus for scan input.)  
  
**set atpg shift period 12.5 -group TG1 TG3 -atpgtype stuck** (Use scan shift period of 12.5 ns for groups TG1,TG3).  
  
**set simulation percentage 20 -block \* -atpgtype stuck trans -corner qcmin qcmax** (Simulate 20% of the block-level stuck and transition patterns at qcmax and qcmin corner conditions.)
- Commands to select the job configuration, e.g. **jobcontrol on -block B\* -step sim -atpgtype stuck -corner qc\_max** (Only verify stuck-at patterns at the QC-max corner for blocks with name beginning with a B.)

- Commands to override specific items, e.g. **set override atpgsetupoverride -step atpg -at setup -atpgtype burnin -text {<tool specific command>}** (At the start of the ATPG run, insert this tool specific command.)

A “job” refers to one or more flow steps. The framework is capable of automatically deriving all the flow sub-steps that must be run to complete the job and the associated dependencies. As a side benefit, the framework can perform *consistency checks* on the job configuration specified by the user. This reduces the chances of human error. After consistency checks pass and job steps have been analyzed, the framework writes out a “flow command file” which expresses the parallelism in the flow. The scheduling algorithm is embedded into the framework and is based on the “as soon as possible” strategy. The scheduling algorithm attempts to minimize the total execution time, which respecting constraints on the number of tool licenses that the user can use. The flow command file is read by the scheduler, which in turn pops up a graphical user interface that shows all the sub-steps to be executed. Even at this stage, the user has the option of running the entire flow or selectively running parts of the flow.

The shell runs in two modes, a "setup mode," in which the user can set the variables in the kernel. If the user does not specify some variables, the framework assumes default values and logs them. When the user switches to the "run" mode, the settings of the variables are considered "frozen". In the run mode, the user can generate the flow command file. At this point, the framework also generates all the scripts required to run the steps that are part of the job control.

Figure 5 shows an example of a command file to insert a divide and conquer DFT into a design. Figure 6 shows an example command file for ATPG and pattern simulations. Figure 7 shows an example command file for performing the test pattern checks and handoff. Finally, Figure 8 shows the job scheduler interface whilst it is running the ATPG and pattern verification flow.

```

# DFT Insertion and verification for F001
# Flow Information
set top module F001

# Tool Information
set atpg tool tetramax \
-location /tools/synopsys/tmax_2002.12
set insert tool dftcompiler \
-location /apps/synopsys/U-2003.06

# General Dft insertion Information
set testgroup G1 -block BLOCK3 -instance BLOCK3_inst1 \
-block BLOCK4 -instance BLOCK4_inst1
set testgroup G2 -block BLOCK1 -instance BLOCK1_inst \
-block RAM -instance RAM_inst
set negedge on -block BLOCK4 RAM
set pin TEST_EN 1
set pin TM 1

# Clock definitions for all the blocks
set clock CLK -block BLOCK1 -offvalue 0 -period 100
set clock CLK -block BLOCK3 -offvalue 0 -period 100
set clock CLK -block BLOCK4 -offvalue 0 -period 100
set clock CLK -block RAM -offvalue 0 -period 100
set clock CLK -group F001 -offvalue 0 -period 100
set clock BIST_CLK -group F001 -offvalue 0 -period 100

# Scan insertion information for top block (F001)
set scan architecture divide_and_conquer
set scan chains 16
set test enable -pins TEST_EN -group F001
set scan group select -pins scan_select[0] scan_select[1]
set scan enable -pins USER_SCAN_EN -group F001
set scan input -prefix USER_SCAN_IN -group F001
set scan output -prefix USER_SCAN_OUT -group F001

# Scan insertion information for blocks BLOCK1, BLOCK3, BLOCK4
set insert resets disabled TEST_RESET1 -block BLOCK1 BLOCK3 BLOCK4
set scan enable -pins SCANENABLE -block BLOCK1 BLOCK3 BLOCK4
set scan input -prefix SCAN_IN -block BLOCK1 BLOCK3 BLOCK4
set scan output -prefix SCAN_OUT -block BLOCK1 BLOCK3 BLOCK4

# Scan insertion information for RAM
set insert resets disabled TEST_RESET2 -block RAM
set scan enable -pins SCANENABLE -block RAM
set scan input -prefix SCAN_IN -block RAM
set scan output -prefix SCAN_OUT -block RAM

# ATPG Verification Information
set atpg shift period 10 -clock CLK BIST_CLK -offvalue 0 0
set atpg fast period 2.5 -clock CLK BIST_CLK -offvalue 0 0

# Simulation Verification Information
set simulation timing check off
set simulation percentage 20

# Flow Control
set lsf run mode true -queue normal -platform sun
set user -id johndoe -jobs 20
jobcontrol on -step insert -block * -group F001
jobcontrol on -step atpg -group * -atpgtype stuck
jobcontrol on -step sim -group * -atpgtype stuck -corner qc_max

# Run the Flow
set mode run
run sequencer -batch true
exit

```

**Figure 5: Command File for Divide and Conquer DFT insertion**

```

# design F999 atpg and simulation
# Flow & Tool Information
set verbose on
set top module F999
set default platform sun
set simulation tool ncoverilog -use 50 \
-location /tools/cadence/ldv4.0_s007
set atpg tool fastscan -use 8 \
-location /tools/mentorg/8.2003_2.10

# scan ports
set scan enable -pins FPGA_TD_9
set scan input -pins FPGA_TD_0 FPGA_TD_1
set scan output -pins FPGA_RD_0 FPGA_RD_1
set test enable -pins DEBUG_MODE_2
set scan chains 2
set scan architecture vanilla

# Specifying clocks
set atpg shift period 20 -clock D_TCK NOISECLKP NOISECLKN \
-offvalue 0 0 1
set atpg fast period 2.5 -clock D_TCK NOISECLKP NOISECLKN \
-offvalue 0 0 1

# ATPG overrides
set atpg effort quick
set D_TRST 1
set D_TMS 1

# Simulation overrides
set simulation corners QC_MAX QC_MIN \
vbox_min vbox_max \
burnin_min burnin_max
set simulation mode parallel -svecs 0 -sbits 5
set simulation percentage 20

# Job Control
set lsf run mode true -queue large -platform sun
set user -id johndoe -jobs 50
jobcontrol on -step atpg \
-atpgtype stuck trans iddq latchup burnin vboxhi vboxlo depara
jobcontrol on -step sim \
-atpgtype stuck trans iddq depara -corner QC_MAX QC_MIN
jobcontrol on -step sim -atpgtype burnin -corner burnin_max burnin_min
jobcontrol on -step sim -atpgtype vboxhi -corner vbox_max
jobcontrol on -step sim -atpgtype vboxlo -corner vbox_min

# Run the Flow
set state run
create testflow
run sequencer
exit

```

**Figure 6: Command File for ATPG and Pattern simulation**

```

# Design F999 Test Pattern Handoff
# Flow & Tool Information

set verbose on
set top module F999
set default platform sun

#Test Pattern Handoff commands

set package name FCBGA_444
set ti part number F999
set padmap files /db/F999/FCBGA_444_padMap \
/db/F999/IoConnectivity.out
set corner table path /db/F999/cornerTableFile

# Flow Control

set lsf run mode true -queue normal -platform sun
set user -id johndoe -jobs 50

jobcontrol on -step singlemodetdchkr
jobcontrol on -step multimodetdchkr
jobcontrol on -step speccap
jobcontrol on -step pehandoff

# Run the Flow

set state run
create testflow
run sequencer
exit

```

**Figure 7: Command file for Test Pattern Handoff Flow**

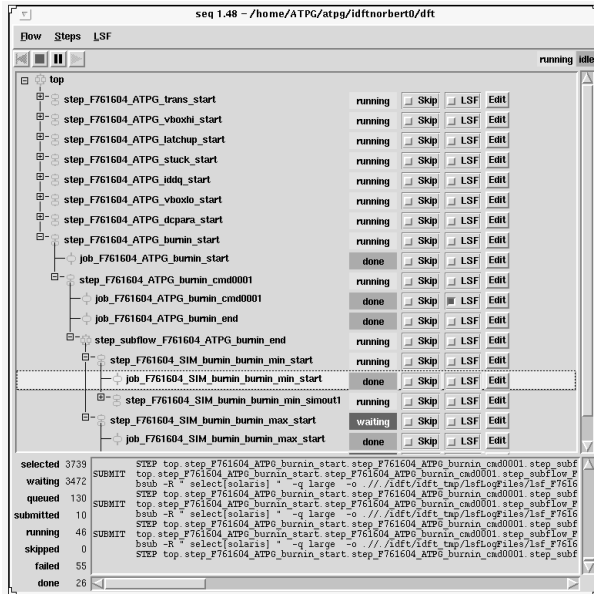
### 5 Results

The DFT framework described above has been used at several installations of our company on several designs. In Table1, the flow execution times obtained through the framework are compared against the classical, manual approach to DFT. The execution time includes the time to run the flow once from start to finish. The setup time, which includes the time to gather data, is not included in the table for either of the two approaches. The setup time for the framework will be smaller, given the consistency checks and help provided by the framework.

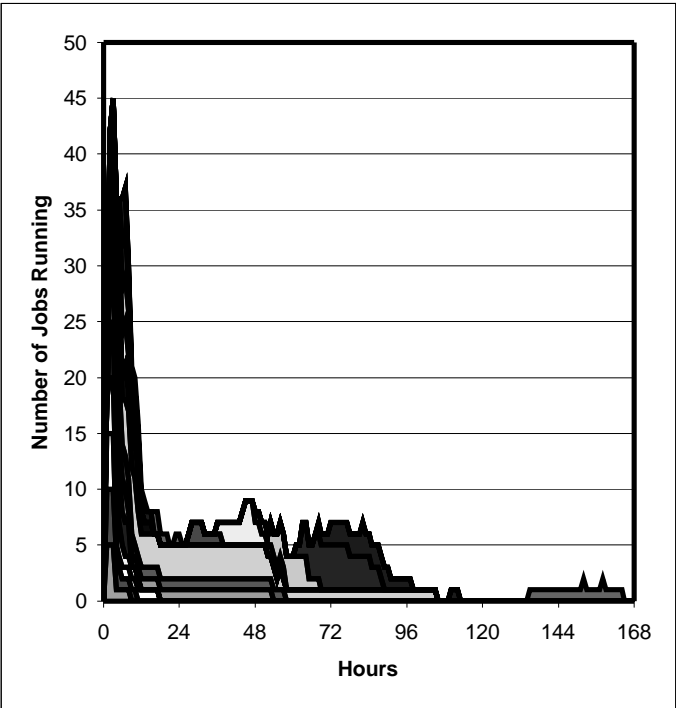
Design	Size (M gates)	Classical Approach	Framework
Design B	~ 5.5	Est 30 days	7 days
Design S	~ 4.8	Est 30 days	6 days
Design WL	~ 2	Est 15 days	3 days
Design TE	~ 1	Est 10 days	2 days
Design J	~ 8	Est 40 days	6 days

**Table 1: Speedup Results from Framework**

The run history for the ATPG and pattern simulation of Design B are shown in Figure 9. Here we see the bulk of the runs completing within about 3 days; however, there is a long tail of both ATPG and simulation jobs for the “daisy chain” mode.



**Figure 8: Scheduler GUI running ATPG and Simulation**



**Figure 9: Design B - Run history**

Detailed results on running ATPG and pattern verification for the design TE are shown in Table 2. This test chip used the plain vanilla scan architecture and 8 scan chains.

ATPG Type	CPU (hrs)	Test Cov %	Max Mem (MB)	Max Swap (MB)
stuck	2.90	95.46	770	1153
Burnin	1.06	81.90	886	1092
Dcpa	2.42	35.73	763	1090
Latchup	0.58	38.89	797	1080
Trans	1.53	75.71	1574	1932
vboxhi	0.56	82.88	880	1093
Vboxlo	0.56	82.88	865	1093
Iddq	5.02	96.37	2368	2722

**Table 2: Results of ATPG on Design TE**

On Design S, divide-and-conquer was employed, with 4 test groups. Some sample results are shown in Table 3 for Test Group 1. The total run time for ATPG, had it been run sequentially, would have been 58923 s. However, the stuck-at fault ATPG runs for 19269 s; the speedup for ATPG alone is therefore around 306%. Similarly, simulation of stuck-at ATPG patterns would yield a speedup of 541%. If we consider the total simulation time for all fault types, this would be 36348 s. The longest running simulation (Transition delay patterns) will dictate the speedup for the simulation subflow, and this can be computed to be 10.12.

Test Group 1	Stuck	Burnin	Vboxlo	Transition
Test Coverage	87.29%	79.28%	76.81%	75.20%
ATPG Time (hrs)	5.35 hrs	0.57 hr	0.58 hr	9.87 hrs
# pattern file simulated	10	2	2	9
Total Time (Verification)	4.59 hrs	0.61 hr	0.34 hr	4.55 hr
Maximum Time (Verification)	0.85 hr	0.30 hr	0.33 hr	0.99 hr
Speedup for simulation	5.41	2.03	1.02	4.57

**Table 3: Design S ATPG and simulation**

Table 4 compares run times on Solaris and Linux platforms. We indicate the results for Test Group 1. The bold face entries correspond to Sun Solaris platform, and the italicized entries correspond to Linux platform. We see that Linux run times are often better. Interestingly, we observed difference in test coverage when we run the tool on two different platforms.

	Stuck	Burnin	Vboxlo	Trans.
<b>Coverage</b>	<b>87.29%</b>	<b>79.28%</b>	<b>76.81%</b>	<b>75.20%</b>
<b>ATPG Time</b>	<b>5.35 hrs</b>	<b>0.57 hr</b>	<b>0.58 hr</b>	<b>9.87 hrs</b>
<b>Sim Time</b>	<b>4.59 hrs</b>	<b>0.61 hr</b>	<b>0.34 hr</b>	<b>4.55 hrs</b>
<b>Max Time</b>	<b>0.85 hr</b>	<b>0.30 hr</b>	<b>0.33 hr</b>	<b>0.99 hr</b>
<i>Coverage</i>	<i>87.29%</i>	<i>73.52%</i>	<i>70.79%</i>	<i>75.04%</i>
<i>ATPG time</i>	<i>1.77 hrs</i>	<i>0.15 hr</i>	<i>0.21 hr</i>	<i>5.86 hrs</i>
<i>Sim time</i>	<i>4.59 hrs</i>	<i>0.64 hr</i>	<i>0.34 hr</i>	<i>3.30 hrs</i>
<i>Max sim time</i>	<i>1.44 hrs</i>	<i>0.32 hr</i>	<i>0.34 hr</i>	<i>0.94 hr</i>

**Table 4: Run-time Comparison on Solaris and Linux on Design S, Test Group 1**

Bold font in the table corresponds to Sun/Solaris, Italics correspond to Linux. Other than run-time differences, the user can notice difference in coverage, which can be attributed to several factors, including a different random number generator.

## 6 Conclusions and Further Work

In this paper, we described a DFT framework that is holistic, in the sense that it addresses the entire flow from insertion to pattern verification and handoff. Because of this approach, we are able to extract the maximum parallelism in the DFT tasks and reduce the total DFT cycle time. The goal behind building such a framework is to provide uniformity in DFT work across designs and design organizations within the same company. Best practices from different design teams were integrated in the insertion, ATPG, and pattern verification. The pattern handoff, which includes running in-house checkers that verify patterns for tester rule compliance and bundling together design and test data, is also a part of the framework. The user can use one or more of the flows supported by the framework. The framework has an easy-to-use command interface to set up the DFT work and generate the flow description file and the command files required to run the tools; the job sequencer has a convenient graphical user interface. The parallelization of jobs on a CPU farm offers a speedup advantage that can cut down the total DFT time to about 7 days for a 10M gate design. The side benefit of the framework is its support for multiple tools and operating systems, which makes exploration possible. The framework development is an ongoing activity with several enhancements planned. Some important enhancements included support for reconfigurable scan, support for logic BIST, and smarter ATPG & pattern verification for multiple instances of the same sub-block.

## Acknowledgements

We acknowledge the contributions to this framework from Nirmalya Haldar, Kamal Kiran, V.R.Devanathan, Rajanikanth Dandamudi, R. Raghuraman and Senthil Arasu. Thanks are also due to David Thomas for developing the task scheduler. We acknowledge the guidance provided by Mike Ales and Ken Butler throughout this work. Fastscan and DFTAdvisor are trademarks of Mentor Graphics Corporation. TetraMax and DFTCompiler are trademarks of Synopsys Inc.. NCVerilog is a trademark of Cadence.

## References

1. T. Aizawa and R. Thummarakudy. *Hierarchical ASIC Design at Three Million Gates and Above*. [www.gdatech.com/articles.htm](http://www.gdatech.com/articles.htm)

2. V.K. Agarwal. To ATPG or not to ATPG? Electronic News. 2002.
3. T.L. Anderson. *A Designer's View of Chip Test*. International Test Conference. 1995. Page 292.
4. B. Bennetts. *IEEE P1500 Embedded Core Test Standard*. [www.dft.co.uk](http://www.dft.co.uk). February 2001.
5. B. Birkel, B. Hooser, M. Janssens, F. Lenke and V. Vorisek. *Design integration, DFT, and verification methodology for an MPEG 1/2 audio layer 3 (MP3) SoC device*. Proceedings of the IEEE , 12-15. May 2002 Pages:303 - 306
6. M. Bushnell and V.D. Agrawal. *Essentials of Electronic Testing*. Kluwer Academic Publishers. 2000.
7. J. Camara and H. Sarmiento. Tool Management in an Electronic CAD Framework. Proceedings of ISCAS 1995. Pages 928-932.
8. K.T. Cheng, S. Dey, M. Rodgers and K. Roy. *Test Challenges for Deep Sub-micron Technologies*. Design Automation Conference, 2000.
9. V. Chickermane and K. Zarrinch. *Addressing Early Design-for-Test Synthesis in a Production Environment*. International Test Conference. 1997. Pages 246-255.
10. V. Chickermane, D. Lackey, D. Litten and L. Smudde. *Automated Chip-level I/O and Test Insertion Using IBM Design-for-Test Synthesis*. IBM Microelectronics Journal, Second Quarter, 2000.
11. G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan and J. Rajski. *Logic BIST for large industrial designs: real issues and case studies*. ITC 1999: 358-367
12. R.H. Kerr. *Designing for Test with Multimillion-Gate ICs*. IBM TestBench Design-for-Test and Automatic Test Pattern Generation. IBM MicroNews, Second Quarter, 1999. Vol. 5, No.2
13. K.P. Parker. *The Boundary-Scan Handbook*. Kluwer Academic Publishers. 1995.
14. C. P. Ravikumar and Rahul Kumar. *Divide-and-Conquer IDDQ Testing for Core-Based System Chips*. International Conference on VLSI Design 2002: 761-766.
15. A. Sangiovanni-Vincentelli. *The Tides of EDA*. IEEE Design & Test. Nov-Dec, 2003. Pages 59-75.
16. R.S. Tupuri, J.A. Abraham and D.G. Saab. *Hierarchical Test Generation for Systems on a Chip*. Manuscript available through WWW. 2003.
17. Y. Zorian and E.J. Marinissen. *System Chip Test: How will it impact your design?* Design Automation Conference, 2000.